UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

NVIDIA CORPORATION,
Petitioner,

v.

NEURAL AI, LLC,
Patent Owner.

Case No. IPR2025-00609
U.S. Patent No. RE48,438

DECLARATION OF DR. MARY K. BOLIN IN SUPPORT OF PETITION FOR *INTER PARTES* REVIEW OF U.S. PATENT NO. RE48,438

I, Mary K. Bolin, declare as follows:

1. I am Emeritus Professor at the University of Nebraska, Lincoln Libraries. I have held the following positions: Professor and Catalog and Metadata Librarian, Digital Initiatives and Special Collections Department, University of Nebraska—Lincoln Libraries (2015–2020); Professor and Chair, Discovery and Resource Management and Coordinator of Collection Development, University of Nebraska, Lincoln Libraries (2004–2015); San José State University School of Information: Lecturer (2008–present); Head of the Technical Services Department, University of Idaho Library (1993–2003; promoted to Professor in 1999); Head of the Cataloging Department, University of Idaho Library (1986–1993; promoted to Associate Professor and tenured in 1991); and Head of Records Maintenance Section, Cataloging Department, University of Georgia Libraries (1981–1986).

2. Over the course of my work as a librarian, professor, researcher, student research supervisor, and author of numerous scholarly papers and other publications, I have had extensive experience with cataloging records and online library management systems built around Machine-Readable Cataloging (MARC) standards. I also have substantial experience in authenticating printed documents and establishing the date when they were accessible to ordinarily skilled researchers.

3. Attachment A is my full resumé.

NVIDIA Ex1023

4. I have been retained by counsel for NVIDIA Corporation ("Petitioner") to evaluate, authenticate, and establish the date of public availability of a document relevant to the above-captioned Petition for *Inter Partes* Review. For my time performing this service, I am being paid my usual hourly fee. I have no financial interest in this action, or affiliation with Petitioner. My compensation is not dependent upon the outcome of, or my testimony in, this proceeding or any other proceeding, including before the U.S. Patent Office.

A. Document of Interest

5. The document that is the subject of this Declaration is the following:

Document 1. Luo, Z., Liu, H., & Wu, X. (2005, July). Artificial neural network computation on graphic process unit. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* (Vol. 1, pp. 622–626). IEEE.

Document 2. Pharr, Matt, and Randima Fernando. *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional, 2005.

B. Methodology

6. I searched for this document in the OCLC database, Google Scholar, and one or more subscription databases such as IEEE Xplore and ACM Digital Library. I also searched the library catalogs of institutions such as the University of

Nebraska—Lincoln and other large research libraries. My goal was to find a record in one of these databases that contained a date indicating when the document was available to the public.

7. *Use of MARC Records*. The "Machine Readable Cataloging" standard, also known as "MARC," is an industry-wide standard method of organizing library catalog information. The Library of Congress first developed MARC in the 1960s. A MARC-compatible library is one that has a catalog consisting of individual MARC records for each of its works. Today, MARC is the primary communication protocol for the transfer and storage of bibliographic metadata in libraries.

8. OCLC is a database of MARC records used by libraries worldwide. When an OCLC-participating institution acquires a work, it creates a MARC record for this work in its computer catalog system, and OCLC automatically supplies the date of creation. The MARC record creation date reflects the date on which (or shortly after) the item was first acquired and cataloged. Once the MARC record is created by a cataloger at an OCLC-participating member institution, it is instantaneously available to any OCLC-participating members and, therefore, available to the public.

9. A MARC record comprises several fields, each of which contains specific data about the work. Each field is identified by a standardized, unique, three-digit code corresponding to the type of data that follows. For example, a work's title

NVIDIA Ex1023

is recorded in field 245, the primary author of the work is recorded in field 100, an item's International Standard Book Number ("ISBN") is recorded in field 020, an item's Library of Congress call number is recorded in field 050, and the publication date is recorded in field 260 or 264 under the subfield "c." If a work is a periodical, then its publication frequency is recorded in field 310, and the publication dates (*e.g.*, the first and last publication) are recorded in field 362, which is also referred to as the enumeration/chronology field. The library that created the record is recorded in field 040 in subfield "a" with a unique library code. When viewing the MARC record online via OCLC's Worldshare Record Manager database, hovering over this code with the mouse reveals the name of the library. I used this method of "mousing over" the library codes in the OCLC database to identify the originating library for all of the MARC records in this Declaration, as well as the "display holdings" command to see a list of libraries that own the item. In the MARC records in the OCLC Worldshare Record Manager database, MARC field 040 contains the OCLC symbols of institutions transcribing, creating, and modifying the record.

10. OCLC was created "to establish, maintain and operate a computerized library network and to promote the evolution of library use, of libraries themselves, and of librarianship, and to provide processes and products for the benefit of library users and libraries, including such objectives as increasing availability of library resources to individual library patrons and reducing the rate of rise of library per-

unit costs, all for the fundamental public purpose of furthering ease of access to and use of the ever-expanding body of worldwide scientific, literary and educational knowledge and information."[1] Among other services, OCLC and its members are responsible for maintaining the WorldCat database (http://www.worldcat.org/), which is used by independent and institutional libraries throughout the world. OCLC is the international cooperative to which libraries catalog records and others pay to access MARC records. The MARC records referenced in this Declaration, extracted from the OCLC database, were generated in the ordinary course of business. Further, I personally located and downloaded each of these records associated with the document on which this Declaration focuses.

11. OCLC provides its members online access to MARC records through its OCLC Worldshare Record Manager database. When an OCLC member institution acquires a work, it creates a MARC record for this work in its computer catalog system in the ordinary course of its business. MARC records created at the Library of Congress are batch-loaded into the OCLC database through a subscription to MARC Distribution Services daily or weekly. Once the MARC record is created

[1] Third Article, Amended Articles of Incorporation of OCLC Online Computer Library Center, Incorporated (available at http://www.oclc.org/en-US/membership/councils.html).

NVIDIA Ex1023

by a cataloger at an OCLC member institution or is tape-loaded from the Library of Congress, it is then made available to any other OCLC members online, and, therefore, made available to the public.

12. When an OCLC member institution creates a new record, OCLC automatically supplies the date of creation for that record. The MARC record creation date reflects the date on which, or shortly after which, the item was first acquired or cataloged.

13. The date of creation for the MARC record appears in the fixed field (008), characters 00 through 05. Field 005 is a 16 digit field that records the date characters yyyymmdd including 4 for the year, 2 for the month, and 2 for the day. The date entered on file never changes. These dates are automatically supplied by the OCLC system and cannot be changed by the user. When the MARC record is created in OCLC, it is available throughout the world to other libraries and in WorldCat, the public version of OCLC Connexion. Thus, the date of creation of the MARC record by a cataloger at an OCLC member institution reflects when the underlying item is accessible to the public.

14. MARC records also include several fields that include subject matter classification information. An overview of MARC record fields is available through

NVIDIA Ex1023

the Library of Congress.[2] For example, 6XX fields are termed "Subject Access Fields."[3] Among these, for example, is the 650 field, which is the "Subject Added Entry – Topical Term" field.[4] The 650 field is a "[s]ubject added entry in which the entry element is a topical term." *Id.* The 650 field entries "are assigned to a bibliographic record to provide access according to generally accepted thesaurus-building rules (e.g., *Library of Congress Subject Headings* (LCSH), *Medical Subject Headings* (MeSH))." *Id.* Thus, a researcher can easily discover material relevant a topic of interest with a search using the terms employed in the MARC Fields 6XX.

15. For Documents 1 and 2, I performed a search of OCLC (www.oclc.org), which is a large shared database used by libraries around the world to create and exchange cataloging data, using MARC format, which is a universal communications format for that data.

16. Using the records from OCLC, I determined when the record was created, and which libraries own the item. I then visited the website of the library or libraries that own the item, searched their library catalog, and determined when the item was added to the library's collection.

---

[2] *See* http://www.loc.gov/marc/bibliographic/.

[3] *See* http://www.loc.gov/marc/bibliographic/bd6xx.html.

[4] *See* http://www.loc.gov/marc/bibliographic/bd650.html.

17. I am familiar with many online catalogs, databases, and search engines. In preparing for this Declaration, I used authoritative information systems such as the above-mentioned WorldCat (https://www.worldcat.org), the online catalog of the Library of Congress (https://catalog.loc.gov), shared research databases such as ResearchGate and CiteSeerX, the online catalog of a number of large research libraries such as the University of Nebraska—Lincoln, IEEE Xplore, and Google Scholar search for records. These records are identified and discussed in this Declaration.

18. The IEEE Xplore[5] research database is a vital resource for undergraduate and graduate students, researchers, scientists, and the public, offering access to high-quality, peer-reviewed publications. IEEE Xplore serves as a repository for IEEE's ebooks, standards, conference proceedings, technical reports, and journal articles, with content dating back to 1988 and selectively covering materials from 1913 to 1987. The database currently includes over 5.5 million full-text documents and is recognized for its rigorous curation and accurate bibliographic records. IEEE Xplore ensures precise reporting of publication dates and event locations in alignment with industry standards for major scholarly and technical journals. Approximately 20,000 new documents are added each month. While

[5] *See* https://ieeexplore.ieee.org/Xplore/home.jsp.

abstracts are freely available, full-text access requires a subscription or institutional login.

19. I am familiar with the Internet Archive, a digital library formally certified by the State of California as a public library. Among other services that the Internet Archive makes available to the general public, is the Wayback Machine, an online archive. The Internet Archive's Wayback Machine service archives webpages as of a certain capture date to track changes in the web over time. The Internet Archive has been in operation as a nonprofit library since 1996 and has hosted the Wayback Machine service since its inception in 2001. I consider the Internet Archive's records keeping to be as rigorous and detailed as other formal library records keeping practices such as MARC records and OCLC records.

C. Opinions Regarding Document 1

20. The following are my opinions with respect to the authenticity and public availability of Document 1—Luo, Z., Liu, H. & Wu, X., *Artificial neural network computation on graphic process unit* (July 2005).

1. Authentication of Document 1

21. Attachment B is a true and correct copy of Document 1 obtained from IEEE Xplore. This attachment is identical to Exhibit 1005 to the Petition for *Inter Partes* Review.

22. Attachment C is a PDF of the record of Document 1 on IEEE Xplore, which includes the full text of Document 1. According to IEEE Xplore, Document 1 was published in the Proceedings of the 2005 IEEE International Conference on Neural Networks, held from July 31 to August 4, 2005, in Montreal, QC, Canada. I compared the text of Document 1 from IEEE Xplore, and the context is identical to that of Exhibit 1005 to the Petition for *Inter Partes* Review.

23. It is therefore my opinion that Exhibit 1005 to the Petition for *Inter Partes* Review is an authentic copy of Document 1.

2. Public Availability of Document 1

24. Document 1, on its face, states that it is from "Proceedings of International Joint Conference on Neural Networks, Montreal, Canada, July 31 - August 4, 2005." *See* Att. B at 1. It also bears other indicia of publication, such as "©2005 IEEE" and the print ISBN "0-7803-9048-2." *Id.*

25. Document 1 was published in the International Joint Conference on Neural Networks (IJCNN) 2005 Conference Proceedings. Attachments D and E are the cover and copyright page of the "2005 IEEE International Joint Conference on Neural Networks," respectively, which I obtained from IEEE Xplore.[6] Both identify the IEEE Catalog No. (05CH37662) and ISBN (0-7803-9048-2) of the publication.

[6] *See* https://ieeexplore.ieee.org/xpl/conhome/10421/proceeding.

26. Attachment F is the Table of Contents of the conference proceedings. According to the Table of Contents, Document 1 was part of the "Plenary Poster Session: Hardware." Att. F at xxxiv. Attachment G is the "IJCNN 2005: Session Grid," also obtained from IEEE Xplore. According to the Session Grid, the Plenary Poster Sessions took place on Monday, August 1 and Tuesday, August 2, 2005, from 7:00 p.m. to 11:00 p.m. Att. G at iv–v.

27. Attachment H is a copy of the article Luo, Z., Liu, H. & Wu, X., *Artificial neural network computation on graphic process unit*, obtained through ResearchGate.[7] The content of this version of the article is identical to that of Document 1 obtained through IEEE Xplore, except this version (1) contains a cover page added by ResearchGate, (2) does not have the IEEE markings in the header and footer on the first page, and (3) does not have the pagination added by IEEE. Attachment I is a screenshot of the PDF properties for the version of the document obtained through ResearchGate. It shows that the PDF was created and last modified

[7] *See* https://www.researchgate.net/profile/Hongzhi-Liu-7/publication/4202315_Artificial_neural_network_computation_on_graphic_process_unit/links/53f5b7940cf2fceacc6f56a6/Artificial-neural-network-computation-on-graphic-process-unit.pdf.

on April 27, 2005, which is consistent with it having been presented and made available at the IJCNN conference in August 2005.

28. In addition to having been presented at the conference, Document 1 is available in the IEEE Xplore database.[8] According to Attachment C (a PDF of the record of Document 1 on IEEE Xplore), Document 1 was added to IEEE Xplore on December 27, 2005. Att. C at 1 ("Date Added to IEEE *Xplore*: 27 December 2005").

29. Attachment J is a screenshot of the PDF properties for Document 1. It shows that the PDF was last modified on December 21, 2005, which is consistent with it having been made available on IEEE Xplore on December 27, 2005.

30. Attachment K is a PDF of the MARC record for the conference proceedings that includes Document 1 from OCLC. This MARC record shows a creation date of January 4, 2006.

31. Attachment L is a PDF of the MARC record for the conference proceedings that includes Document 1 from the University of Arizona's library catalog. This MARC record shows a creation date of January 4, 2006.

32. Based on the facts that Document 1 is found in IEEE Xplore with a "date added" of December 27, 2005, that a MARC record for the conference proceedings that include this paper is found in OCLC with a creation date of January

[8] *See* https://ieeexplore.ieee.org/abstract/document/1555903.

4, 2006, and that a MARC record for the conference proceedings that include this paper is found in the catalog of the University of Arizona library with a creation date of January 4, 2006, it is my opinion that Document 1 was publicly available by December 27, 2005, and no later than around January 4, 2006.

3. Locating Document 1

33. Field 245 (Title Statement) of the MARC records indicates that Document 1 was included in "Proceedings of the International Joint Conference on Neural Networks (IJCNN) 2005," which took place "July 31-August 4, 2005, Hilton Montréal Bonaventure Hotel, Montréal, Québec, Canada" and was "co-organized by the International Neural Network Society (INNS), IEEE Computational Intelligence Society." Att. K at 2. Field 650 (Topical Term) of the MARC records indicates that this publication relates to, among other things, "Neural networks (Computer science)." Att. K at 2; Att. L at 1.

34. It is my opinion that someone searching for the subject matter of Document 1 in an academic library (such as the University of Arizona's library), including by terms such as "neural networks" would have been able to locate Document 1.

35. In the IEEE Xplore record for Document 1, IEEE included keywords (which IEEE explains are meant to "assist with retrieval of results") such as "[a]rtificial neural networks," "[n]eural networks," "[g]raphics," "[p]arallel

algorithms," and "[c]entral processing unit." *See* Att. M at 1. Document 1 was further cataloged using index terms such as "Neural Network," "Graphics Processing Unit," "Artificial Neural Network Computation," and "Artificial Neural Network Model." *Id.* at 2.

36. It is my opinion that someone searching for any of the above terms on IEEE Xplore would have been able to locate Document 1. Indeed, and as an example, I searched IEEE Xplore to confirm that by merely searching using keywords, a skilled researcher would be able to retrieve Document 1. By searching articles from 2006 and before using the Boolean search ("artificial neural networks" AND "graphics processing unit") on IEEE Xplore, I was able to locate Document 1. *See* Att. N. Search tools on IEEE Xplore existed in 2005 and 2006. For example, Attachment O is the IEEE *Xplore* 2.0 User Guide from 2005, which includes a section on "Searching IEEE Xplore," including example searches. Att. O at 25–50. At that time, IEEE Xplore allowed searching by title, author, abstract, index terms, and full text. *Id.* at 26. Attachment P is a printout from the Internet Archive of the home page of IEEE Xplore (http://ieeexplore.ieee.org/Xplore/guesthome.jsp) from March 29, 2005. It confirms that searching functionality was available. Att. P at 1.

37. Thus, it is my opinion that a skilled researcher searching the subject-matter of Document 1 would have been able to locate this reference by around December 27, 2005, and no later than around January 4, 2006.

D. Opinions Regarding Document 2

38. The following are my opinions with respect to the authenticity and public availability of Document 2—Pharr, Matt, and Randima Fernando. *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional, 2005 (also referred to below as "GPU Gems 2").

4. Authentication of Document 2

39. Attachment Q is a true and correct copy of Document 2, which was scanned from a copy of GPU Gems 2 that I understand was purchased by counsel for Petitioner (*see* Attachment R). I compared the pages of Attachment Q to the pages of the physical copy of GPU Gems 2, and I confirmed that Attachment Q is an accurate and complete scan of GPU Gems 2.

40. I have been informed by counsel and understand that Exhibit 1032 submitted by Petitioner in the *IPR* proceedings is the same true and correct copy of Attachment Q that I attach to my declaration.

41. It is therefore my opinion that Attachment Q (Exhibit 1032 to the Petition for *Inter Partes* Review) is an authentic copy of Document 2.

5. Public Availability of Document 2

42. Attachment Q shows a copyright date of 2005 and a "First printing" date of March 2005. It was published by Addison-Wesley, a known and reputable book publisher.

43. Attachment S is a PDF of the MARC record for Document 2 from the UT-Austin library catalog. The creation date is listed as July 18, 2005.

44. Attachment T is a PDF of the MARC record for Document 2 from the Brigham Young University library catalog. The creation date is listed as July 28, 2005.

45. Based on the facts that Document 2 lists a "First printing" data of March 2005 (Att. Q) and MARC records for Document 2 with creation dates of July 18, 2005 (Att. S) and July 28, 2005 (Att. T), it is my opinion that Document 2 was available in libraries around July 2005.

6. Locating Document 2

46. In my professional experience as a librarian, academic libraries possessed some form of digital catalog by 2005. A skilled researcher could have accessed the digital catalogs of libraries such as those at UT-Austin or Brigham Young University, where Document 2 was located.

47. For example, Document 2 could have been located by searching the above libraries by some portion of the title (*GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation*) or by searching the subject headings (from Field 650 of the MARC records) "Computer graphics" and "Real-time programming."

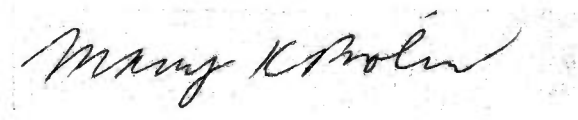
NVIDIA Ex1023

48. Thus, it is my opinion that someone searching for the title or subject matter of Document 2 would have been able to locate Document 2 around July 2005.

* * *

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements are made with the knowledge that willful false statements and the like are punishable by fine, imprisonment, or both, under Section 1001 of Title 18 of the U.S. Code. I state under penalty of perjury under the laws of the United States of America that the foregoing is true and correct.

Executed this April 22, 2025, in Lincoln, Nebraska.



Mary K. Bolin

NVIDIA Ex1023

ATTACHMENT A

Mary K. Bolin, PhD
2433 U Street
Lincoln, Nebraska 68503
mary@marykbolin.com

Mary K. Bolin, PhD

Professor Emeritus
University Libraries
University of Nebraska—Lincoln
mbolin2@unl.edu

Lecturer,
School of Information
San José State University
mary.bolin@sjsu.edu

EDUCATION

PhD, Educational Studies (Educational Leadership and Higher Education), 2007, University of Nebraska

Dissertation: *A Typology and Discourse Analysis of the Status and Appointments of Librarians at Land Grant Universities* (available at digitalcommons.unl.edu)

MA, English, 1999, University of Idaho
Thesis: *Grace: A Contrastive Analysis of a Biblical Semantic Field* (available at digitalcommons.unl.edu)

MSLS, 1981, University of Kentucky

BA, Linguistics, 1976, University of Nebraska—Lincoln

WORK EXPERIENCE

Center for Digital Research in the Humanities. Fellow, 2019- Emeritus, 2020-

University of Nebraska--Lincoln Libraries: Professor and Catalog and Metadata Librarian, Digital Initiatives and Special Collections, 2015-2020. Emeritus, 2020-

University of Nebraska—Lincoln Libraries: Professor and Chair, Discovery and Resource Management, 2004-2015; Coordinator of Collection Development, 2012-2015

University of Idaho Library: Head, Technical Services Department, 1993-2003. Promoted to Professor in 1999

University of Idaho Library: Head, Cataloging Department, 1986-1993. Promoted to Associate Professor and tenured, 1991

University of Georgia Libraries: Head, Records Maintenance Section, Cataloging Department, 1981-1986

Teaching:
San José State University School of Information:

LIBR 281, Metadata. 2009-
LIBR 248, Introduction to Cataloging and Classification. 2013-
LIBR 202, Information Retrieval. 2008-2013

Associate Coordinator, Gateway PhD program, San Jose State University, 2021-2022

American Library Association

Cataloging for Non-Catalogers, Online Continuing Education Course, 2017-
Intermediate Cataloging for Non-Catalogers, Online Continuing Education Course, 2018-

Lyrasis

Webinar on BibFrame, taught twice a year, 2020-
Quick and Clean Cataloging, Parts 1, 2, and 3. 2022-

Independent Study in Idaho

Library Science C415, Technical Services for Small Libraries, 2004-2013
Library Science C416, Library Services to Rural Libraries, 2004-2013
Library Science C422, Use of the School Library, 1999-2013

Library Science C425/525, Organization and Management of Small Libraries, 1988-2013
Library Science C427, Library or Media Center Practicum, 1995-2013

Gateway PhD Program (San Jose State University and Queensland University of Technology). SJSU supervisor for two PhD students, 2015-

Gateway PhD Program (San Jose State University and Manchester Metropolitan University). Associate Coordinator, 2021-2022

Consulting
Expert witness for multiple attorneys employed by Wolf, Greenfield & Sacks, P.C., Hughes Hubbard & Reed LLP, and other firms. 2017-

Editing
Language and academic editing for EditSprings, a service for international scholars, 2020-

SCHOLARSHIP

Refereed Publications:

(Publications available from digitalcommons.unl.edu)

Pytlik Zillig, Brian, and Mary K. Bolin. (Forthcoming 2023).“Visualization of Semantic Primes in Texts from Early Modern English Drama.” *Journal of the Japanese Association for the Digital Humanities.*

Bolin, Mary K. (2022). *Refocusing Academic Libraries through Learning and Discourse: The Idea of a Library.* Chandos/Elsevier.

Bolin, Mary K. (2022). “Academic Libraries and Writing Centers: Collaborations at US Public Research Universities.” *School of Information Student Research Journal, 11*(2). Retrieved from https://scholarworks.sjsu.edu/ischoolsrj/vol11/iss2/2

Bolin, Mary K. (2022). *Refocusing Academic Libraries through Learning and Discourse: The Idea of a Library*. Oxford, UK: Chandos/Elsevier.

Bolin, Mary K. (2022). "Metadata, Catalog Systems, Linked Data, and the Evolving ILS." In: Sandra Hirsh (Ed.), *Information Services Today: An Introduction.* 3rd ed. New York: Rowman and Littlefield.

Wintermute, Harriet E., and Bolin, Mary K. (2018). "Metadata Generators for Backlog Reduction: Metadata Maker Streamlines Cataloging and Facilitates Transition. *Technical Services Quarterly 35*(2) doi: https://doi.org/10.1080/07317131.2018.1422879

Bolin, Mary K. (2018). "Metadata, Catalog Systems, Linked Data, and the Evolving ILS." In: Sandra Hirsh (Ed.), *Information Services Today: An Introduction.* 2nd ed. New York: Rowman and Littlefield.

Bolin Mary K. (2017). *The 21-century Academic Library: Global Patterns of Organization and Discourse.* Oxford: Chandos Publishing. (Chandos Information Professional Series)

Bolin, Mary K. (2017). Linguistics and LIS: A Research Agenda." *SLIS Student Research Journal,* 7:1.
Retrieved from http://scholarworks.sjsu.edu/slissrj/vol7/iss1/2

Bolin, Mary K. (2014). "Open Access Web Resources for Library Continuing Education and Training." In S. Hines (Ed.), *Revolutionizing the Development of Library and Information Professionals: Planning for the Future* (pp. 107- 124). Hershey, PA: . doi:10.4018/978-1-4666-4675-9.ch008.

Bolin, Mary K. (2014). "The Language of Academic Librarianship: The Discourse of Promotion and Tenure. "*Advances in Library Administration and Organization 32*.

Bolin, Mary K. "Librarian Status at US Research Universities: Extending the Typology." *Journal of Academic Librarianship* 34:4 (September 2008)

Bolin, Mary K. "A Typology of Librarian Status at Land Grant Universities." *Journal of Academic Librarianship*, 34: 3 (May 2008)220:230.

Bolin, Mary K. "The Library and the Computer Center: Organizational Patterns at Land Grant Universities." *Journal of Academic Librarianship*, 31:1 (2005)

Eckwright, Gail Z. and Mary K. Bolin. "The Hybrid Librarian: The Affinity of Collection Management with Technical Services and the Organizational Benefits of an Individualized Assignment." *Journal of Academic Librarianship*, 27:6 (2001)452:457.

Bolin, Mary K. "Catalog Design, Catalog Maintenance, Catalog Governance," L*ibrary Collections, Acquisitions, and Technical Services.* 24: (2000)53-63.

Bolin, Mary K. "The Collegial Environment and the Functional Organization," *Journal of Library Administration*. 29:2(1999):49-61.

Bolin, Mary K. "Cataloging Environmental Impact Statements and Related Documents," *Resource Sharing and Library Networks*. 14:1(1999): 21-40.

Bolin, Mary K. "Waiting for the Electrician or Someone like Him: Client-Centered Technical Services" Proceedings of the Association of College and Research Libraries National Conference, Nashville, April 1997.
< http://www.ala.org/Content/ContentGroups/ACRL1/Nashville_1997_Papers/Bolin.htm >

Grants:

Electronic Theses and Dissertations. Seed Grant from University of Idaho Office of Research, 2002. Co PI with Gail Z. Eckwright. $6500.

Interoperability of Metadata for Thematic Research Collections. Grant to the Center for Digital Research in the Humanities from the Institute of Museum and Library Services 2005-2008. Project Team Member.
http://unlcms.unl.edu/cas/center-for-digital-research-in-the-humanities/projects/pages/iom_files/IOM_Final_Narrative_Report_July_2008.pdf

Other Publications:

Library Philosophy & Practice. Peer-reviewed electronic library science journal. Mary K. Bolin and Gail Z. Eckwright, founders and editors. http://digitalcommons.unl.edu/libphilprac

Library Science C415, Technical Services for Small Libraries, through Independent Study in Idaho. 2004.

Library Science C416, Library Services to Rural Libraries, through Independent Study in Idaho. 2004.

Library Science C422, Use of the School Library, through Independent Study in Idaho. Revised 1999.

Library Science C425/525, Organization and Management of Small Libraries through Independent Study in Idaho. Revised 2005.

Library Science C427, Library or Media Center Practicum, Independent Study in Idaho. Revised 1999.

Presentations:

Bolin, Mary K. “BibFrame and the MARC Cataloging Environment: an Introduction.” Lyrasis Webinar, June 16, 2020, August 17, 2021.. Recording available http://lyrasis.adobeconnect.com/paufo9e0dm30/

Pytlik Zillig, Brian, and Mary K. Bolin. “A Little More than Kin, a Little Less than Kind”: the Semantic Field Grace in Early Modern English" Poster Session, Text Encoding Initiative Meeting, Graz, Austria, September 18, 2019

Pytlik Zillig, Brian, and Mary K. Bolin. “Shakespeare and the Enumeration of Semantic Primes.” Research Paper Presentation, Text Encoding Initiative Meeting, Tokyo, Japan, September 13, 2018

"Library Philosophy and Practice: Creating and Maintaining an Open-Access E-Journal," Presentation to UNCL meeting (libraries at campuses of the University of Nebraska), Lincoln, Nebraska, March 20, 2008.

Presentation to "Thinking outside the Borders" participants on the open-access electronic journal Library Philosophy and Practice and its role in the international library community. University of Nebraska—Lincoln Libraries, October 24, 2007

Panel discussion on open access electronic journals. GWLA directors' meeting. Linda K. Hall Library, Kansas City, Missouri. April 2007.

"Waiting for the Electrician or Someone Like Him: Client-Centered Technical Services." Association of College and Research Libraries National Conference, Nashville, April 5, 1997.

" 'Some Problems of an Editor': the Idaho Librarian from Manuscript to Published Issue," Joint Annual Conference of the Idaho Library Association and the Washington Library Association, Spokane, Washington, April 6, 1995.

"Automating Idaho's Libraries: a Historical Perspective." Idaho Library Association Annual Conference, Idaho Falls, October 4, 1990.

"Small Library Management." Idaho Library Association Spring Conference, Lewiston, Idaho, May 13, 1989.

"No More 'Good Workers': Excellence and Academic Library Classified Staff." Association of College and Research Libraries 5th Annual Conference, Cincinnati, April 7, 1989.

"Communication to and from Cataloging Departments." Heads of Cataloging Discussion Group, American Library Association Midwinter Meeting, San Antonio, January 1988.

Discussion Leader at the LAMA (Library Administration and Management) Management Exchange. American Library Association Annual Meeting, San Francisco, June 1987.

"Staff Turnover." Heads of Cataloging Discussion Group, American Library Association Midwinter Meeting, Chicago, January 1987.

SELECTED PROFESSIONAL SERVICE

Reviewer. *Resources for College Libraries*. Linguistics bibliography, Spring 2020.

Peer reviewer for ALISE National Conference presentations, Spring 2020.

Peer reviewer for *Journal of Academic Librarianship,* 2012-

Peer reviewer for refereed journal, *Western Nigerian Journal of Medical Sciences* (http://www.wnjms.org/), a publication of Ladoke Akintola University of Technology (LAUTECH) Teaching Hospital, 2008-

PhD Reviewer for students from Aligarh Muslim University, India, 2010-

Outside reviewer for numerous library faculty promotion and tenure cases in US universities.

University of Nebraska--Lincoln

Chair, University Libraries Faculty, 2018

Faculty Senate Academic Rights and Responsibilities Committee, 2016- Chair, 2018-19

Faculty Senate – 2004-2008; 2008-2011

Faculty Senate Executive Committee 2004-2008

Academic Planning Committee (APC), 2005-2008; Vice Chair, 2006-2007; Chair, 2007-2008

Aesthetic Review Committee 2006-2007

APC monitor for academic program reviews: Department of Political Science, 2005; Department of Educational Psychology 2006

Pacific Northwest Library Association:

Editor, *PNLA Quarterly*, December 2002-December 2013

Idaho Library Association:

Editor, *The Idaho Librarian*, September 1990-January 1996

ATTACHMENT B

Artificial Neural Network Computation on Graphic Process Unit

Zhongwen Luo, Hongzhi Liu and Xincai Wu

Faculty of Information, China University of Geoscience(Wuhan) , Wuhan 430074,China
E-mail : luozw@cug.edu.cn,liuhz87.student@sina.com, xcwu@cug.edu.cn

Abstract

Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given.

Keywords: Graphic Process Unit; ANN; SOM; MLP

I. INTRODUCTION

Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation.

MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors.

For relative works, Kyoung-Su Oh et al.[2] implement an GPU based MLP for classify the text area in a image, and give an almost 20 time speed up over CPU. Thomas Rolfes[3] gives an artificial neural network implementation using a GPU-based BLAS level 3 style single-precisions general matrix-matrix product. Bohn[4] describes an SOM calculation method based on OpenGL hardware speed-up on SGI workstation, which inspired our work to further deploy the possibility to implement ANN calculation based on PC commodity graphic hardware.

In recent years, the graphic hardware performance is doubled every 12 months which is much faster than CPU's performance increase speed which is doubled every 18 months. And GPU vendors had make programmability on GPU, which make it possible for implement general-purpose computation.

In this paper, two kinds of ANN computation on GPU are given. In section 2, SOM computation model and implementation on graphic hardware is discussed. In section 3, MLP computation model and implementation on GPU is discussed. In section 4, the computation result and comparison are given for both CPU and GPU. In section 5, some of the design details and lessons we learned during implementation are given. In section 6, conclusion and some future works are given.

II. COMPUTATIONAL MODEL AND IMPLEMENT METHOD FOR SOM

A. The SOM Computational Model

The SOM takes a two-step computation: search for the best matching unit (BMU) and modify the map according to a distance function of the lateral connections. Usually, the Euclid distance is chosen as similarity measure method. The calculation formula takes as follow:

$$\| W_b - \xi \| < \| W_i - \xi \| \quad \text{for any i;} \qquad [1]$$

Modify the unit value regarding a distance function of the lateral connections as follow.

$$W_i^{new} = W_i^{old} - \varepsilon\Omega(r_b, r_i) * (W_i^{old} - \xi) \qquad [2]$$

NVIDIA Ex1023
Page B-2

As Bohn described, three OpenGL[5] extension functions are needed, they are blending, glColorMatrix and glminmax. These functions are fully support by SGI workstation, but only partially supported by PC graphic hardware. So it is difficult to fully implement SOM on OpenGL. Fortunately, current GPU provide more programmability, which makes it possible for implement the SOM calculation on GPU.

B. SOM Implementation on Programmable GPU

As discussed in the previous part, not all OpenGL functions supported by SGI workstation are supported by PC commodity graphic hardware. But recently, graphic hardware vendors provide programmability and some high-level program language[6,7]. This kind of programmability is at a lower level than OpenGL, and is more powerful than fixed OpenGL function. In this implementation, Cg [8] (C for graphic) is chosen as developing environment.

As described in previous part, the calculation contains two steps. The first is finding the best matching unit, and the second is adjusting the value according to the distance from BMU.

For finding the BMU, two steps are needed. In the first step, similarity measurement is calculated; in the second step, the minimum values which represent the BMU are found and located. The similarity computation code for GPU takes as follow:

```
Half 4 temp= tex2D(texture,coords) -intrain;
c.x = dot(temp,temp);
c.yz= coords.xy;
```

The first two sentences calculate the square of Euclid distance of two vectors, the third sentence save the unit coordinate, which will be used late for locating the best match unit.

The main difficulty in Cg computation comes from finding the minimum value and determination its location, for there is no global variable in Cg environment. We use a multipass method to calculate it. Our scheme shows as Fig 1. In each pass we find the minimum value of four units show as colored and save the result in a small size texture. After some step, the size decrease to 1, and we can get the minimum value and its location.

After get BMU, we can adjust the self-organize map according to equation [2].

III. MLP FOR REAL TIME BALL RECOGNIZING IN FIRA SOCCER ROBOT

A. Background

Currently, there are two main world soccer-robot competition, one is RoboCup[9], the other is FIRA[10]. In both case, Object are identified by their unique color. But in the near future, these color cues may be removed, so new vision algorithms based on model are needed to cope with the situation. The model based algorithm for finding ball

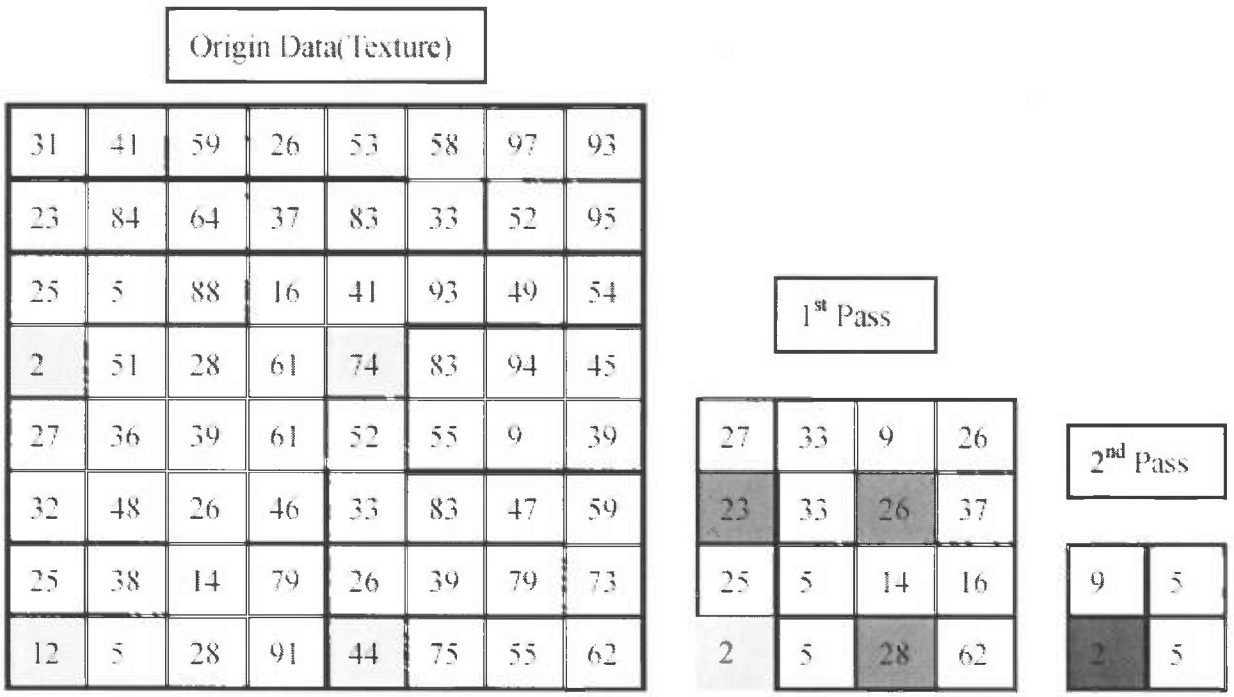


Fig. 1: Scheme for find the minimum value

NVIDIA Ex1023
Page B-3

and non-ball location will discussed below.

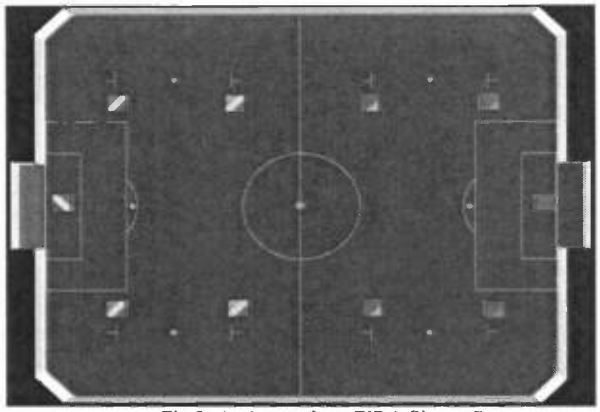


Fig 2. An image from FIRA SimuroSot

Fig2 shows an image taken from FIRA robot soccer 5vs5 simulator. Our task is to develop a model to recognize the ball, and then using this model to recognize the ball in real time.

For each location, the characters are calculated considering the color value at a small area around the location. The area radius takes as 7, which is shown in Fig 3,

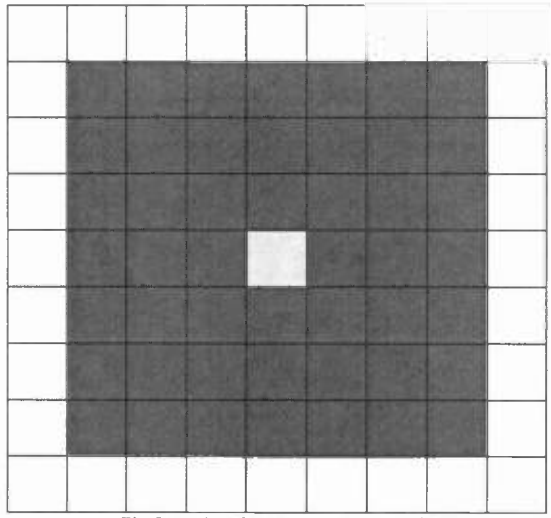


Fig 3. region for character calculation

Seven parameters are selected as characters of the ball. There are 3 average value of the colors red, green and blue around the concerning position:

$$mean_{x,y} = \sum_{i=-3}^{3}\sum_{j=-3}^{3} c_{x+i,y+j} \Big/ 49$$

3 standard deviation of the colors red green and blue:

$$delta_{x,y} = \sqrt{\sum_{i=-3}^{3}\sum_{j=-3}^{3}(c_{x+i,y+j} - mean_{x,y})^2 \Big/ 48}$$

And the luminance:

$$luminance = r + g + b$$

B.Computational Model of MLP

Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network.

The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set.

The trained MLP is used to recognize and trace the ball in real time.

There are two main computation steps for MLP used as the classification machine. The first one is matrix multiplication:

$$net = w \bullet x + b \qquad [3]$$

The second one is sigmoid function calculation:

$$\sigma(net) = \frac{1}{1 + e^{-net}} \qquad [4]$$

As for the determination of ball on robot soccer, the MLP calculation was applied on each point in the play ground. And MLP was used to distinguish between ball and non-ball position.

C. MLP Implementation on Graphic Hardware

Current GPU has a limited instruction length and a limited number of temporary variables for calculation at each location. So multipass is needed for complex problem. For MLP discussed in this section, three passes are performed. In the first pass, average values of three color and luminance are calculated. In the second pass, the standard deviations of the three colors are calculated. In the third pass, classification result is got from MLP calculation on the characteristic.

NVIDIA Ex1023
Page B-4

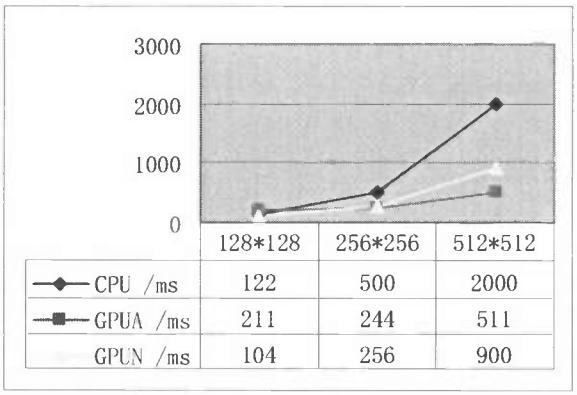
In this calculation, we use a new Nvidia's GF6000 serious graphic hardware. The reason is that the ATI's GPU supports very few numbers of operations in each pass; so more passes are need for MLP calculation. Old Nvidia's GPU does not support fully float texture, which is crucial for the precision of result.

IV. RESULTS AND DISCUSSION

The environment for CPU computing is INTEL P4 2.4G. Based on this PC, ATI 9550 and Nvidia GF5700 GPU are used for SOM computing, Nvidia GF6200 GPU is used for MLP computing.

A. CPU and GPU Train Time for SOM and Discussion

Usually the train process for self organize map is time consuming when the map is large enough. So there is a need to speed up the training procedure. For our test problem, 80 data are chosen for training the SOM and average time for the computation on CPU and GPU can be reached. The result is shown in Fig 4.



	128*128	256*256	512*512
CPU /ms	122	500	2000
GPUA /ms	211	244	511
GPUN /ms	104	256	900

Fig. 4: SOM train computing time on GPU and CPU

The result shows that GPU based implementation is faster than CPU, especially for large self organize map. As map size increase, the computation time on GPU increase slowly than that on CPU. And different GPU had different result, for Nvidia's GPU, it takes the least time for small size SOM like 128*128, but for ATI's GPU it takes the least time for larger size SOM like 256*256. The difference may come from vendor's hardware implementation. The result shows that ATI's graphic card takes more time for the compile of program and the code is better optimized so computation time decrease with more data, but Nvidia's graphic card takes less time for compile and takes more time to computing.

B. MLP Computation Time for CPU and GPU

The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table I.

The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time.

TABLE I: MLP COMPUTATION ON CPU AND GPU.

CPU /ms	11328
GPUN /ms	46

V. SOME IMPLEMENTATION DETAILS AND LESSONS

To increase the efficiency, some basic rule should obey. The main rules are given below.

First, create the GPU hardware program only once and enable it when it is needed. The reason is that when a new program is created, it will be compiled by Cg, which is time consuming.

Second, if possible, do one's best to decrease the calculation passes. For computation scheme described in section 2.2, the main bottleneck is at finding minimum procedure. The test shows that the bottleneck comes from multi-pass used in finding minimum procedure. Table 2 shows result for different scheme of finding the minimum. To decrease the pass, we make two changes, the first is to combine the value calculation with one pass of find minimum, and the second is to combine two pass of find minimum computation into one pass. Instead of calculate 4 units, 16 units are calculated. GPUA represent calculation on ATI graphic card and GPUN represent the calculation on NVIDIA graphic card. The performance increase is clear, especially for Nvidia's graphic card.

TABLE II: COMPARISON BETWEEN MORE AND FEW PASS

KFM size	128*128	256*256	512*512
GPUA more pass /ms	366	400	533
GPUA few pass /ms	211	244	511
GPUN more pass /ms	190	640	2889
GPUN few pass /ms	104	256	900

Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL's PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called "Render to Texture" to easy the use of PBuffer. We had used this class in our program.

Fourth, hardware from different venders usually has different property. So if one implementation is not work at one kind of hardware, try another implementation. For

NVIDIA Ex1023
Page B-5

example, in the calculation of minimum value, firstly we use the following code:

```
c=tex2D(texture,coords).x<tex2D(texture,coords+half
2(0,offset)).x? tex2D(texture,coords) :
tex2D(texture,coords+half2(0,offset));
```

Which can give the correct result in ATI card, but can not get correct result in an Nvidia's card, we think that the inner parallel schemes makes the difference. To work around it, the above sentence is changed to the following logical equivalent one:

```
half4 c=tex2D(tex,coords);
if (c.x>tex2D(tex,coords+half2(0,half_side)).x)
    c=tex2D(tex,coords+half2(0,half_side));
```

Then the result is correct for both graphic cards.

VI. CONCLUSION

In this paper, implementation for two ANN models on graphic hardware is given. Inherent parallelism of commodity graphic hardware is used to accelerate the computation of ANN. The result shows that GPU is capable for some of ANN calculation, the graphic hardware make it possible for an increasing performance/cost ratio on the area of large size ANN computation.

Compared to Bohn's initial computation on SGI workstation, our implementation has two benefits. One is our calculation is more precise, for we had use the float point computing. The other is that we only use a commodity available graphic card, which is much more easily available than SGI workstation so can be widely used.

The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have "video in" function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don't need to transfer data between CPU and GPU, which will speed the process.

We can also do other general ANN computation on GPU, because GPU provide almost all arithmetic operation, logic operation and some important mathematic function. And for the application of neural network on images, it is more naive to make such computing on a graphic hardware.

For future works, one is to make other kinds of ANN computation on GPU. The other is to further deploy the MLP on more real situation of robot soccer, which include select better parameter and use faster algorithm on GPU.

REFERENCES

[1] Teuvo Kohonen. Self-organizing maps. Springer Verlag, New York, 1997.

[2] Kyoung-Su Oh, Keechul Jung, GPU implementation of neural networks, Pattern Recognition 37, 6, 1311-1314, 2004

[3] Thomas Rolfes. Artificial Neural Networks on Programmable Graphics Hardware, in Game Programming Gems 4,pp373-378, 2004

[4] Bohn, C.A. Kohonen Feature Mapping Through Graphics Hardware. In Proceedings of 3rd Int. Conference on Computational Intelligence and Neurosciences 1998. 1998.

[5] Woo, M., Neider, J., Davis, T., Shreiner, D. OPENGL Programming Guide,Addison-Wisley,1999

[6] Femando R. and Killgard, M.J. The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics Addison-Wisley, 2003

[7] Harris, M. http://www.gpgpu.org/developer/, 2003

[8] Mark, W.R., Glanville, R. S., Akeley, K. and Killgard, M.J. 2003. Cg : A system for programming graphics hardware in a C-like language. ACM Trans. Graph. 22, 3, 896-907

[9] http://www.robocup.com

[10] http://www.fira.net

NVIDIA Ex1023
Page B-6

ATTACHMENT C

Artificial neural network computation on graphic process unit

Publisher: IEEE Cite This PDF

Zhongwen Luo ; Hongzhi Liu ; Xincai Wu All Authors

13 Cites in Papers
1 Cites in Patent
921 Full Text Views

Abstract
Document Sections
I. INTRODUCTION
II. COMPUTATIONAL MODEL AND IMPLEMENT METHOD FOR SOM
III. MLP FOR REAL TIME BALL RECOGNIZING IN FIRA SOCCER ROBOT
IV. RESULTS AND DISCUSSION
V. SOME IMPLEMENTATION DETAILS AND LESSONS
Show Full Outline
Authors
Figures
References
Citations
Keywords
Metrics
More Like This

Abstract:
Artificial neural network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given.

Published in: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.

Date of Conference: 31 July 2005 - 04 August 2005

DOI: 10.1109/IJCNN.2005.1555903

Date Added to IEEE *Xplore*: 27 December 2005

Publisher: IEEE

Print ISBN:0-7803-9048-2

Conference Location: Montreal, QC, Canada

ISSN Information:

SECTION I.
INTRODUCTION

Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM)[1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation.

MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation(BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors.

For relative works, Kyoung-Su Oh et al.[2] implement an GPU based MLP for classify the text area in a image, and give an almost 20 time speed up over CPU. Thomas Rolfes[3] gives an artificial neural network implementation using a GPU-based BLAS level 3 style single-precisions general matrix-matrix product. Bohn[4] describes an SOM calculation method based on OpenGL hardware speed-up on SGI workstation, which inspired our work to further deploy the possibility to implement ANN calculation based on PC commodity graphic hardware.

In recent years, the graphic hardware performance is doubled every 12 months which is much faster than CPU's performance increase speed which is doubled every 18 months. And GPU vendors had make programmability on GPU, which make it possible for implement general-purpose computation.

PDF Help

NVIDIA Ex1023
Page C-2

In this paper, two kinds of ANN computation on GPU are given. In section 2, SOM computation model and implementation on graphic hardware is discussed. In section 3, MLP computation model and implementation on GPU is discussed. In section 4, the computation result and comparison are given for both CPU and GPU. In section 5, some of the design details and lessons we learned during implementation are given. In section 6, conclusion and some future works are given.

SECTION II.
COMPUTATIONAL MODEL AND IMPLEMENT METHOD FOR SOM

A. the SOM Computational Model

The SOM takes a two-step computation: search for the best matching unit (BMU) and modify the map according to a distance function of the lateral connections. Usually, the Euclid distance is chosen as similarity measure method. The calculation formula takes as follow:

$$\|W_b - \xi\| < \|W_i - \xi\| \text{ for any i;} \quad [1]$$

View Source

Modify the unit value regarding a distance function of the lateral connections as follow.

$$W_i^{new} = W_i^{old}\text{-}\varepsilon\Omega(r_b, r_i)^*(W_i^{old}\text{-}\xi) \quad [2]$$

View Source

As Bohn described, three OpenGL[5] extension functions are needed, they are blending, gIColorMatrix and glminmax. These functions are fully support by SGI workstation, but only partially supported by PC graphic hardware. So it is difficult to fully implement SOM on OpenGL. Fortunately, current GPU provide more programmability, which makes it possible for implement the SOM calculation on GPU.

B. SOM Implementation on Programmable GPU

As discussed in the previous part, not all OpenGL functions supported by SGI workstation are supported by PC commodity graphic hardware. But recently, graphic hardware vendors provide programmability and some high-level program language[6], [7]. This kind of programmability is at a lower level than OpenGL, and is more powerful than fixed OpenGL function. In this implementation, Cg [8] (C for graphic) is chosen as developing environment.

As described in previous part, the calculation contains two steps. The first is finding the best matching unit, and the second is adjusting the value according to the distance from BMU.

For finding the BMU, two steps are needed. In the first step, similarity measurement is calculated; in the second step, the minimum values which represent the BMU are found and located. The similarity computation code for GPU takes as follow:

$$\begin{aligned} &\text{Half 4 temp} = \text{tex2}D(\text{texture}, \text{coords})\text{-intrain}; \\ &\text{c.x} = \text{dot}(\text{temp}, \text{temp}); \end{aligned} \qquad \text{c.yz} = \text{coords.xy};$$

View Source

The first two sentences calculate the square of Euclid distance of two vectors, the third sentence save the unit coordinate, which will be used late for locating the best match unit.f

The main difficulty in Cg computation comes from finding the minimum value and determination its location, for there is no global variable in Cg environment. We use a multipass method to calculate it. Our scheme

NVIDIA Ex1023
Page C-3

shows as Fig 1. In each pass we find the minimum value of four units show as colored and save the result in a small size texture. After some step, the size decrease to 1, and we can get the minimum value and its location.

After get BMU, we can adjust the self-organize map according to equation [2].

SECTION III.
MLP FOR REAL TIME BALL RECOGNIZING IN FIRA SOCCER ROBOT

A. Background

Currently, there are two main world soccer-robot competition, one is RoboCup[9], the other is FIRA[10]. In both case, Object are identified by their unique color. But in the near future, these color cues may be removed, so new vision algorithms based on model are needed to cope with the situation. The model based algorithm for finding ball and non-ball location will discussed below.

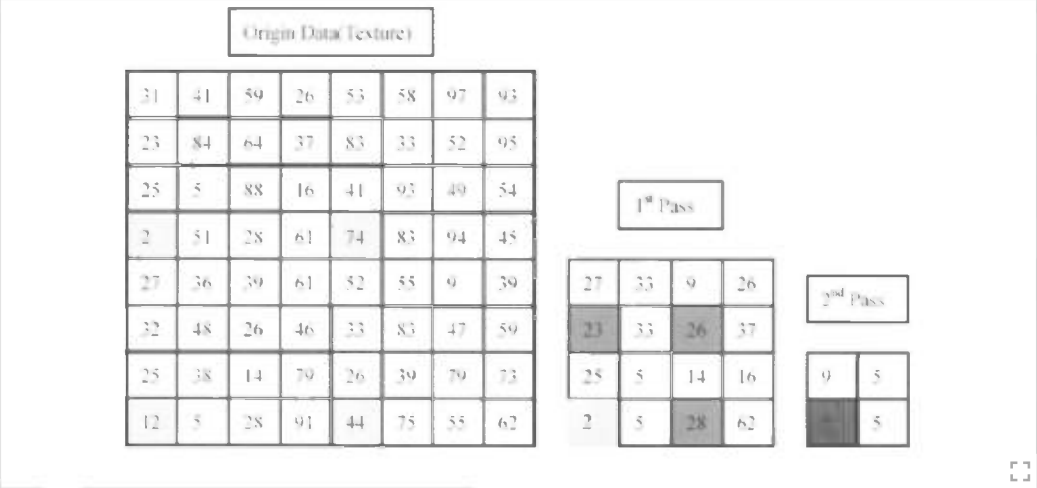


Fig. 1: Scheme for find the minimum value

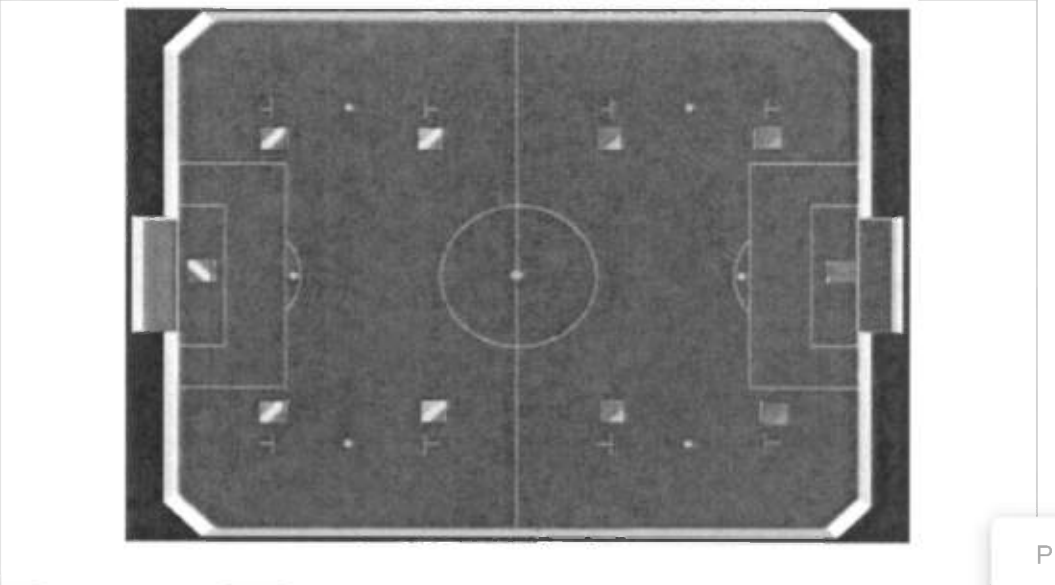


Fig 2. An image from FIRA SimuroSot

Fig2 shows an image taken from FIRA robot soccer 5vs5 simulator. Our task is to develop a model to recognize the ball, and then using this model to recognize the ball in real time.

NVIDIA Ex1023
Page C-4

For each location, the characters are calculated considering the color value at a small area around the location. The area radius takes as 7, which is shown in Fig 3,

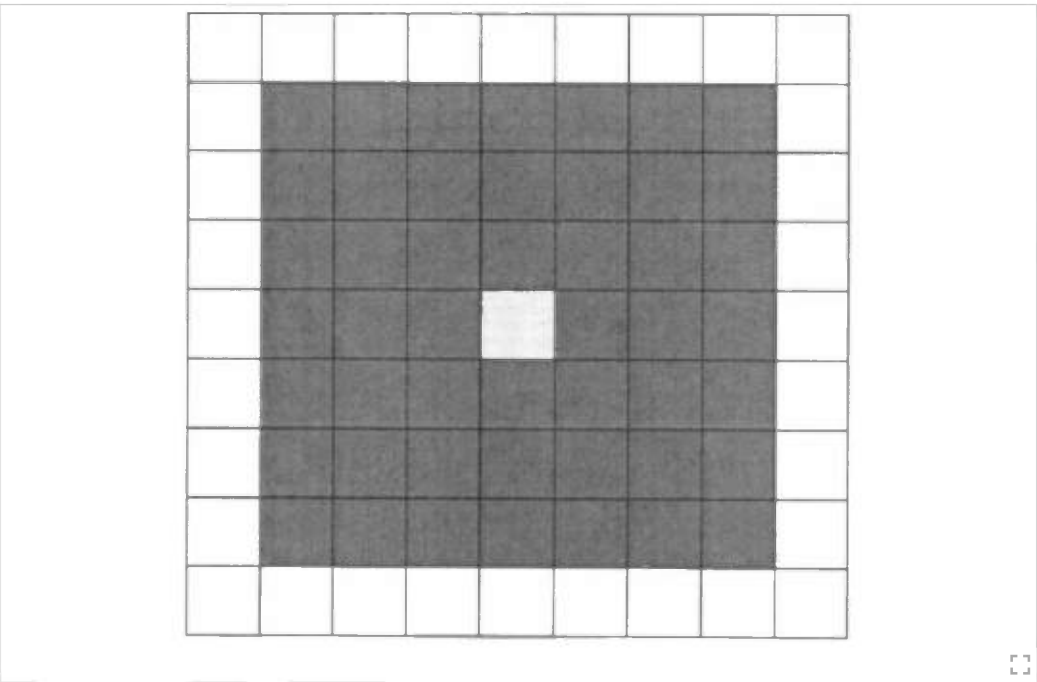


Fig 3. region for character calculation

Seven parameters are selected as characters of the ball. There are 3 average value of the colors red, green and blue around the concerning position:

$$mean_{x,y} = \sum_{i=-3}^{3}\sum_{j=-3}^{3} c_{x+i,y+j}/49$$

View Source

3 standard deviation of the colors red green and blue:

$$delta_{x,y} = \sqrt{\sum_{i=-3}^{3}\sum_{j=-3}^{3}(c_{x+i,y+j} - mean_{x,y})^2/48}$$

View Source

And the luminance:

$$luminance = r + g + b$$

View Source

B. Computational Modelof MLP

Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network.

The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set.

The trained MLP is used to recognize and trace the ball in real time.

NVIDIA Ex1023
Page C-5

There are two main computation steps for MLP used as the classification machine. The first one is matrix multiplication:

$$net = w \bullet x + b \quad [3]$$

View Source

The second one is sigmoid function calculation:

$$\sigma(net) = \frac{1}{1 + e^{-net}} \quad [4]$$

View Source

As for the determination of ball on robot soccer, the MLP calculation was applied on each point in the play ground. And MLP was used to distinguish between ball and non-ball position.

C. MLP Implementation on Graphic Hardware

Current GPU has a limited instruction length and a limited number of temporary variables for calculation at each location. So multipass is needed for complex problem. For MLP discussed in this section, three passes are performed. In the first pass, average values of three color and luminance are calculated. In the second pass, the standard deviations of the three colors are calculated. In the third pass, classification result is got from MLP calculation on the characteristic.

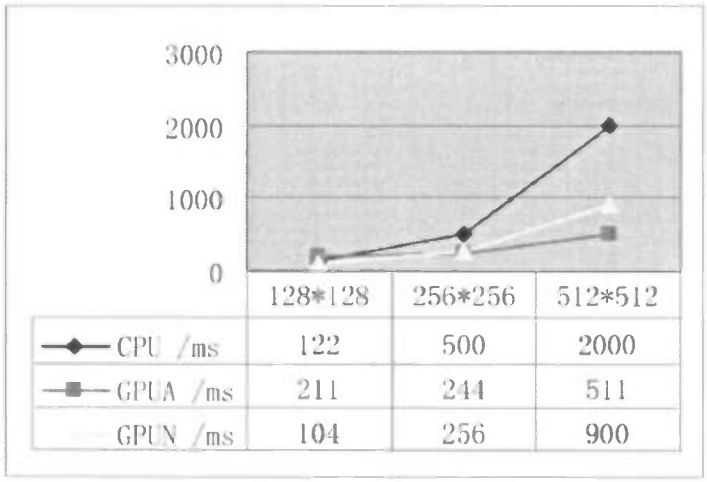
In this calculation, we use a new Nvidia's GF6000 serious graphic hardware. The reason is that the ATI's GPU supports very few numbers of operations in each pass; so more passes are need for MLP calculation. Old Nvidia's GPU does not support fully float texture, which is crucial for the precision of result.

SECTION IV.
RESULTS AND DISCUSSION

The environment for CPU computing is INTEL P4 2.4G. Based on this PC, ATI 9550 and Nvidia GF5700 GPU are used for SOM computing, Nvidia GF6200 GPU is used for MLP computing.

A. CPU andGPU Train Time for SOM and Discussion

Usually the train process for self organize map is time consuming when the map is large enough. So there is a need to speed up the training procedure. For our test problem, 80 data are chosen for training the SOM and average time for the computation on CPU and GPU can be reached. The result is shown in Fig 4.



	128*128	256*256	512*512
CPU /ms	122	500	2000
GPUA /ms	211	244	511
GPUN /ms	104	256	900

PDF
Help

NVIDIA Ex1023
Page C-6

Fig. 4: SOM train computing time on GPU and CPU

The result shows that GPU based implementation is faster than CPU, especially for large self organize map. As map size increase, the computation time on GPU increase slowly than that on CPU. And different GPU had different result, for Nvidia's GPU, it takes the least time for small size SOM like 128*128, but for ATI's GPU it takes the least time for larger size SOM like 256*256. The difference may come from vendor's hardware implementation. The result shows that ATI's graphic card takes more time for the compile of program and the code is better optimized so computation time decrease with more data, but Nvidia's graphic card takes less time for compile and takes more time to computing.

B. MLP ComputationTime for CPU and GPU

The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1.

The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time.

TABLE 1: MLP COMPUTATION ON CPU AND GPU.

CPU /ms	11328
GPUN /ms	46

SECTION V.
SOME IMPLEMENTATION DETAILS AND LESSONS

To increase the efficiency, some basic rule should obey. The main rules are given below.

First, create the GPU hardware program only once and enable it when it is needed. The reason is that when a new program is created, it will be compiled by Cg, which is time consuming.

Second, if possible, do one's best to decrease the calculation passes. For computation scheme described in section 2.2, the main bottleneck is at finding minimum procedure. The test shows that the bottleneck comes from multi-pass used in finding minimum procedure. Table 2 shows result for different scheme of finding the minimum. To decrease the pass, we make two changes, the first is to combine the value calculation with one pass of find minimum, and the second is to combine two pass of find minimum computation into one pass. Instead of calculate 4 units, 16 units are calculated. GPUA represent calculation on ATI graphic card and GPUN represent the calculation on NVIDIA graphic card. The performance increase is clear, especially for Nvidia's graphic card.

TABLE I: COMPARISON BETWEEN MORE AND FEW PASS

KFM size	128*128	256*256	512*512
GPUA more pass /ms	366	400	533
GPUA few pass /ms	211	244	511
GPUN more pass /ms	190	640	2889
GPUN few pass /ms	104	256	900

PDF
Help

NVIDIA Ex1023
Page C-7

Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL's PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called "Render to Texture" to easy the use of PBuffer. We had used this class in our program.

Fourth, hardware from different venders usually has different property. So if one implementation is not work at one kind of hardware, try another implementation. For example, in the calculation of minimum value, firstly we use the following code:

```
c = tex2D(texture, coords).x<tex2D(texture, coords+half
2(0, offset)).x? tex2D(texture, coords) :
tex2D(texture, coords+half2(0, offset));
```

View Source

Which can give the correct result in ATI card, but can not get correct result in an Nvidia's card, we think that the inner parallel schemes makes the difference. To work around it, the above sentence is changed to the following logical equivalent one:

```
half4c=tex2D(tex, coords);
if(c.x>tex2D(tex, coords + half2(0, half_side)).x)
   c=tex2D(tex, coords+half2(0, half_side));
```

View Source

Then the result is correct for both graphic cards.

SECTION VI.
CONCLUSION

In this paper, implementation for two ANN models on graphic hardware is given. Inherent parallelism of commodity graphic hardware is used to accelerate the computation of ANN. The result shows that GPU is capable for some of ANN calculation, the graphic hardware make it possible for an increasing performance/cost ratio on the area of large size ANN computation.

Compared to Bohn's initial computation on SGI workstation, our implementation has two benefits. One is our calculation is more precise, for we had use the float point computing. The other is that we only use a commodity available graphic card, which is much more easily available than SGI workstation so can be widely used.

The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have "video in" function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don't need to transfer data between CPU and GPU, which will speed the process.

We can also do other general ANN computation on GPU, because GPU provide almost all arithmetic operation, logic operation and some important mathematic function. And for the application of neural network on images, it is more naive to make such computing on a graphic hardware.

For future works, one is to make other kinds of ANN computation on GPU. The other is to further deploy the MLP on more real situation of robot soccer, which include select better parameter and use faster algorithm on GPU.

PDF
Help

NVIDIA Ex1023
Page C-8

Authors

Figures

References

Citations

Keywords

Metrics

IEEE Personal Account

CHANGE USERNAME/PASSWORD

Purchase Details

PAYMENT OPTIONS

VIEW PURCHASED DOCUMENTS

Profile Information

COMMUNICATIONS PREFERENCES

PROFESSION AND EDUCATION

TECHNICAL INTERESTS

Need Help?

US & CANADA: +1 800 678 4333

WORLDWIDE: +1 732 981 0060

CONTACT & SUPPORT

Follow

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | IEEE Ethics Reporting | Sitemap | IEEE Privacy Policy

A public charity, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2025 IEEE - All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies.

PDF

Help

ATTACHMENT D

2005 IEEE International Joint Conference on Neural Networks (IJCNN)

Montreal, QC, Canada
July 31 - August 4, 2005

Volume 5 of 5



IEEE Catalog Number 05CH37662
ISBN 0-7803-9048-2

NVIDIA Ex1023
Page D-2

ATTACHMENT E

IJCNN 2005 Conference Proceedings

Copyright and Reprint Permission: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limit of U.S. copyright law for private use of patrons those articles in this volume that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923. For other copying, reprint or republication permission, write to IEEE Copyrights Manager, IEEE Operations Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331USA. All right reserved. Copyright ©2005 by the Institute of Electrical and Electronics Engineers, Inc.

Papers are printed as received from authors.

All opinions expressed in the Proceedings are those of the authors and are not binding on The Institute of Electrical and Electronics Engineers, Inc.

Additional copies may be ordered from:
IEEE Order Dept.
445 Hoes Lane / PO Box 1331
Piscataway, NJ 08855-1331 USA

Phone: (Toll Free) +1-800-678-4333
Email: customer-service@ieee.org
Web: shop.ieee.org

Bound Edition:
IEEE Catalog No. 05CH37662
ISBN: 0-7803-9048-2
ISSN: 1098-7576

CD Edition:
IEEE Catalog No. 05CH37662C
ISBN: 0-7803-9049-0

ATTACHMENT F

Table of Contents

Schedule Grids iii

Message from the General Chair viii

Organizing, Program and Review Committees xi

INNS Officers and Board of Governors xiv

INNS President's Welcome xv

IEEE EXCOM and ADCOM xvii

IEEE CIS President's Welcome xviii

Cooperating Societies and Sponsors xx

Conference Information xxi

Exhibit Directory xxii

Conference Meeting Rooms xxv

Schedule-at-a-Glance xxvi

Post-Conference Workshops xxviii

Program

Sunday, July 31, 2005

Tutorials

9:00 a.m.-11:00 a.m.
Tutorial T01: Evolutionary Robotics

9:00 a.m.-11:00 a.m.
Tutorial T03: Bioinfomatics and Machine Learning: the Prediction of Protein Structures on a Genomic Scale

11:15 a.m.-1:15 p.m.
Tutorial T05: Neural Networks That Actually Work In Prediction and Decision/Control: Common Misconceptions Versus Real-World Success

11:15 a.m.-1:15 p.m.
Tutorial T06: Feature Extraction in Computational Intelligence

11:15 a.m.-1:15 p.m.
Tutorial T07: Integrating Language and Cognition: New Results in Computational Intelligence

NVIDIA Ex1023
Page F-2

11:15 a.m.-1:15 p.m.
Tutorial T08: Evolving Connectionist Systems for Adaptive Learning and Knowledge Discovery: Principles, Models and Applications

2:30 p.m.-4:30 p.m.
Tutorial T09: Neural Networks for Dynamic Systems Feedback

2:30 p.m.-4:30 p.m.
Tutorial T10: Unsupervised Learning

2:30 p.m.-4:30 p.m.
Tutorial T11: New Formulations for Predictive Learning

2:30 p.m.-4:30 p.m.
Tutorial T12: Data Visualization of High Dimensional Scientific Data

4:45 p.m.-6:45 p.m.
Tutorial T13: Cognitive Memory

4:45 p.m.-6:45 p.m.
Tutorial T14: Nonlinear Manifolds in Pattern Recognition and Image Analysis

4:45 p.m.-6:45 p.m.
Tutorial T15: Support Vector Machines and Kernel Based Learning

4:45 p.m.-6:45 p.m.
Tutorial T16: Biologically Plausible Artificial Neural Networks

Monday, August 1, 2005

8:00 a.m.-9:00 a.m.
Plenary Talk
Exploring Chemical Space with Computers: Challenges and Opportunities 83

9:30 a.m.-11:30 a.m.
Pattern Recognition

- *Automatic Target Recognition Using New Support Vector Machine*.....*84*
- *Training with Heterogeneous Data* *90*
- *Tracking the States of a Nonlinear System in the Weight-Space of a Feed-Forward Neural Network* *96*
- *Data-Dependent Kernels for High-Dimensional Data Classification* *102*
- *Generalized 2D Principal Component Analysis* *108*
- *Optimal Gradient-Based Learning Using Importance Weights*.....*114*

NVIDIA Ex1023
Page F-3

9:30 a.m.-11:30 a.m.
Special Session: Neurodynamics and Intentional Dynamic Systems

- *Cinematographic Construction by Brains of Knowledge from Information* *120*
- *Analysis of Phase Transitions in KIV with Amygdala during Simulated Navigation Control* *125*
- *Mixture Segmentation and Background Suppression in Chemosensor Arrays with a Model of Olfactory Bulb-Cortex interaction* *131*
- *The language of cortical dynamics (no file)*
- *Evolving Neurodynamic Controllers for Autonomous Robots* *137*

9:30 a.m.-11:30 a.m.
Special Session: Neural Networks Applications to Bioinformatics

- *Data Visualization Methodologies for Data Mining Systems in Bioinformatics* *143*
- *Random projections for assessing gene expression cluster stability* *149*
- *A New Approach to Hierarchical Clustering for the Analysis of Genomic Data* *155*
- *Inferring Protein-Protein Interactions Using Interaction Network Topologies* *161*
- *Predicting sugar regulation in Arabidopsis thaliana using kernel learning methods* *167*
- *Feedback Linearization Using Neural Networks Applied to Advanced Pharmacodynamic and Pharmacogenomic Systems* *173*

9:30 a.m.-11:30 a.m.
Information-Theoretic and Bayesian Learning

- *Improved Spam e-Mail Filtering Based on Committee Machines and Information Theoretic Feature Extraction* *179*
- *A Variational Bayesian Method for Rectified Factor Analysis* *185*
- *An Information Theoretic Approach to Adaptive System Training Using Unlabeled Data* *191*
- *A Study of AdaBoost with SVM Based Weak Learners* *196*
- *Information maximization and cost minimization in information-theoretic competitive learning* *202*
- *Maximally Discriminative Spectral Feature Projections Using Mutual Information* *208*

1:00 p.m.-3:00 p.m.
ICA and PCA

- *Extending Kernel Principal Component Analysis to General Underlying Loss Functions* *214*
- *Nonlinearity and Optimal Component Analysis* *220*
- *Relation between Kernel CCA and Kernel FDA* *226*
- *Extraction of Frame-Difference Features based on PCA and ICA for Lip-Reading* *232*
- *Multinomial PCA for Extracting Major Latent Topics from Document Streams* *238*
- *Post Nonlinear Blind Source Separation by Geometric Linearization* *244*

1:00 p.m.-3:00 p.m.
Models of Neurons, Local Circuits and Systems

- *Which Features Trigger Action Potentials in Cortical Neurons in Vivo* *250*
- *Attention as Sigma-Pi Controlled Ach-Based Feedback* *256*
- *Motivational Modulation of Endogenous Inputs to the Superior Colliculus* *262*

NVIDIA Ex1023
Page F-4

- *Facilitatory Neural Activity Compensating for Neural Delays as a Potential Cause of the Flash-Lag Effect* 268
- *Fisher Information Quantifies Task-Specific Performance in the Blowfly Photoreceptor* 274
- *Evolutionary Training of a Biologically Realistic Spino-neuromuscular System* 280

1:00 p.m.-3:00 p.m.
Bioinformatics

- *Gene Regulatory Networks Inference with Recurrent Neural Network Models* 286
- *A Self-Organizing Neural Network Approach for the Identification of Motifs with Insertions and Deletions in Protein Sequences* 292
- *Functional Grouping of Genes Using Spectral Clustering and Gene Ontology* 298
- *A Simpler Bayesian Network Model for Genetic Regulatory Network Inference* 304
- *Effect of Non-Target Examples on E.coli Promoters Recognition Using Neural Networks* 310
- *Predicting Protein-Protein Interactions Based on Protein-Domain Relationships* 316

1:00 p.m.-3:00 p.m.
Special Session: Evolvable and Emergent Neural Systems

- *Automated Heuristic Growing of Neural Networks for Nonlinear Time Series Models* 320
- *ECG Signal Classification using Block-based Neural Networks* 326
- *An Evolved Seeega Player Capable of Strong Novice-Level Play* 332
- *Evolvable Neural Networks based on Developmental Models for Mobile Robot Navigation* 337
- *A Recurrent RBF Network Model for Nearest Neighbor Classification* 343
- *Face Recognition Using Modular Neural Networks and Fuzzy Sugeno Integral for Response Integration* 349

3:20 p.m.-5:20 p.m.
Control and System Identification

- *Application of a CMAC Neural Network to the Control of a Parallel Hybrid-Electric Propulsion System for a Small Unmanned Aerial Vehicle* 355
- *Inverse Optimal Nonlinear Recurrent High Order Neural Observer* 361
- *System and Method for Determining Harmonic Contributions from Non-Linear Loads Using Recurrent Neural Networks* 366
- *On Output Regulation for SISO Nonlinear Systems with Dynamic Neural Networks* 372
- *Indirect Field-oriented Linear Induction Motor Drive with Petri Fuzzy-neural-network Control* 378
- *Wavelet-Neural-Network-Based Backstepping Control for Chaotic Systems* 384

3:20 p.m.-5:20 p.m.
Spiking Neurons

- *Spatially and Temporally Local Spike-Timing-Dependent Plasticity Rule* 390
- *Differences in the subthreshold dynamics of leaky integrate-and-fire and Hodgkin-Huxley neuron models* 396
- *Rich phenomena of pulse-coupled spiking neurons with triangular waveform input* 400
- *Synchronized Theta Rhythm Selection in a Dentate Gyrus Network Model* 405
- *Noise Benefits in Spiking Retinal and Sensory Neuron Models* 410
- *A Spiking Neuron Representation of Auditory Signals* 416

NVIDIA Ex1023
Page F-5

3:20 p.m.-5:20 p.m.
Special Session: Computational Neurogenetic Modeling

- *Comparative genomic study of Parkinson's disease candidate genes* 422
- *NeuroGene: Integrated Simulation of Gene Regulation, Neural Activity and Neurodevelopment* 428
- *A Hierarchical Coevolutionary Method to Support brain-Lesion Modelling* 434
- *Genome Space and Structure Genome Invariants* 440
- *A Computational Neurogenetic Model of a Spiking Neuron*.....446

3:20 p.m.-5:20 p.m.
Special Session: Recent Advancements in Adaptive Resonance Theory

- *Self-Organizing Hierarchical Knowledge Discovery by an ARTMAP Information Fusion System*.....452
- *Modification of the ART-1 Architecture Based on Category heoretic Design Principles* 457
- *A vigilance-free ART network with general geometry internal classes* 463
- *On the design of an Ellipsoid ARTMAP classifier within the Fuzzy Adaptive System ART Framework* 469
- *Parallelizing the Fuzzy ARTMAP Algorithm on a Beowulf Cluster*.....475

7:00 p.m.-11:00 p.m.
Plenary Interactive Session I

Plenary Poster Session: Applications to genomics

- *Using real-valued meta classifiers to integrate binding site predictions*.....481
- *Cluster Ensemble for Gene Expression Microarray Data*.....487
- *Characterizing Human Gene Splice Sites Using Evolved Regular Expressions* 493
- *Prediction of Contact Map Integrated PNN with Conformational Energy*.....499
- *Neural Network-Based Analysis of DNA Microarray Data* 503
- *Neural Networks for Gene Expression Analysis and Gene Selection from DNA Microarray* 509

Plenary Poster Session: Proteomics and neuroinformatics

- *The Molecules Module of the Brain Architecture Management System* 515
- *Protein Flexibility Modeling Using Kernel Based Methods*.....521
- *An empirical comparison of individual machine learning techniques and ensemble approaches in protein structural class prediction* 527
- *Prediction of Protein Secondary Structure Using Improved Two-level Neural Network Architecture (no file)*
- *Protein secondary structure prediction using machine learning* 532
- *Protein Secondary Structure Prediction with a Hybrid RNN/HMM System* 538
- *Entropy Based Disease Classification of Proteomic Mass Sepectrometry Data of the Human Serum by a Support Vector Machine* 542

NVIDIA Ex1023
Page F-6

Plenary Poster Session: Data mining, text and pattern recognition

- *A Comparative Study on Term Weighting Schemes for Text Categorization* 546
- *One Class Support Vector Machine based Non-Relevance Feedback Document Retrieval* 552
- *Text Clustering with NTSO* 558
- *A Neuro-SVM Model for Text Classification using Latent Semantic Indexing* 564
- *Fast Text Categorization with Min-Max Modular Support Vector Machines* 570
- *Weight Sharing on Naive Bayes Document Model* 576
- *Optimization of Context Matching Parameters for Web Information Retrieval Using an Evolutionary Algorithm* 582
- *Snap-drift Learning for Phrase Recognition* 588
- *An Adaptive Function Neural Network (ADFUNN) for Phrase Recognition* 593
- *Treatment of Missing Data Using Neural* 598
- *An Iterative Relevance Feedback Learning Algorithm For Image Retrieval Systems* 604

Plenary Poster Session: Hardware

- *Design of an Optical Fixed-Weight Learning Neural Network* 610
- *Forgetful Logic Circuits for Pulse-Mode Neural Networks* 616
- *Artificial Neural Network Computation on Graphic Process Unit* 622
- *Circuit Implementation of Multi-Thresholded Neuron (MTN) Using BiCMOS Technology* 627
- *FPGA Implementation of Pulse Coupled Oscillator* 633
- *Analog Current-Mode Implementation of Central Pattern Generator for Robot Locomotion* 639
- *Toward an Analog VLSI Implementation of a Decision Making Model* 645
- *On Kolmogorov's Superpositions: Novel Gates and Circuits for Nanoelectronics* 651
- *Hardware Implementation of CMAC and B-Spline Neural Networks for Embedded Applications* 657
- *DSP-Based Neural Systems for the Perceptual Assessment of Visual Quality* 663
- *Analog Current-Mode Design for Soft-Max Computation* 669
- *Low-Voltage Pseudo Floating-Gate Reconfigurable Linear* 675
- *The Effects of Quantization on Support Vector Machines with Gaussian Kernel* 681
- *Building a Cheaper Artificial Brain* 685
- *A Digital LSI Architecture of Elastic Graph Matching and Its FPGA Implementation* 689
- *Simulated Control of a Tracking Mobile Robot by Four aVLSI Integrate-and-Fire Neurons Paired into Maps* 695

Plenary Poster Session: Neurodynamics

- *Recurrent Neural Networks Training with Stable Risk-Sensitive Kalman Filter Algorithm* 700
- *A Method of Oscillatory Trajectory Generation Using Recurrent Hybrid Neural Networks* 706
- *A New K-Winners-Take-All Neural Network* 712
- *A Constrained Optimization Algorithm for Training Locally Recurrent Globally Feedforward Neural Networks* 717
- *SCRAM: Statistically Converging Recurrent Associative Memory* 723
- *A New Model for Learning in Graph Domains* 729

NVIDIA Ex1023
Page F-7

- *Empirical Approximation for Lyapunov Functions with Artificial Neural Nets* *735*
- *Ability to Skip Steps Emerging from Chaotic Dynamics* *741*
- *Nonlinear dynamics on VSF-Network* *747*
- *Dynamical Behavior of a Chaotic Neural Network and its Application to Optimization Problems* *753*
- *An Analysis of Associative Chaotic Neurodynamics by Using Surrogate Neurons* *758*

Plenary Poster Session: Hopfield networks

- *A New Kind of Hopfield Networks for Finding Global Optimum* *764*
- *Learning cycles brings chaos in continuous Hopfield networks* *770*
- *Phase Diagrams for Locally Hopfield Neural Networks in Presence of Correlated Patterns* *776*
- *2 Types of Complex-Valued Hopfield Networks and the Application to a Traffic Signal Control* *782*
- *A New N-Parallel Updating Method of the Hopfield-Type Neural Network for N-Queens Problem* *788*

Plenary Poster Session: Reinforcement learning

- *On-Line System Identification Using Context Discernment* *792*
- *The k-Server Problem: A Reinforcement Learning Approach* *798*
- *Task Similarity Measures for Transfer in Reinforcement Learning Task Libraries* *803*
- *Neural Network Model for Time Series Prediction by Reinforcement Learning* *809*

Plenary Poster Session: Decision making, abductive and transductive reasoning

- *Learning Nonlinear State-Space Models for Control* *815*
- *A Neural Network Model for the Decision-Making Process Based on AHP* *821*
- *Negative Reinforcement and Backtrack-Points for Recurrent Neural Networks for Cost-Based Abduction* *827*
- *Distributed Computation for Neural-Based Abductive Reasoning* *833*
- *Transductive Modeling with GA Parameter Optimization* *839*
- *Symbolic Rule Extraction with a Scaled Conjugate Gradient Version of CLARION* *845*

Plenary Poster Session: Support Vector Machines

- *One-Against-All Multi-Class SVM Classification Using Reliability Measures* *849*
- *K-Fold Generalization Capability Assessment for Support Vector Classifiers* *855*
- *Weighted Support Vector Machine for Data Classification* *859*
- *Algorithms of Fast SVM Evaluation based on Subspace Projection* *865*
- *Yet Faster Method to Optimize SVR Hyperparameters based on Minimizing Cross-Validation Error* *871*
- *Support Vector Regression Performance Analysis and Systematic Parameter Selection* *877*
- *Pre-selection of Working Set for SVM Decomposition Algorithm* *883*
- *The Intermediate Matching Kernel for Image Local Features* *889*
- *MIMO SVMs for classification and regression using the geometric algebra framework* *895*
- *Fuzzy integral for a rapid mixture of support vector machines* *901*
- *Feature Subset Selection for Support Vector Machines using Confident Margin* *907*
- *Assignment Kernels For Chemical Compounds* *913*

NVIDIA Ex1023
Page F-8

- *Characterization of Data Complexity for SVM Methods* *919*
- *Optimizing Resources in Model Selection for Support Vectors Machines* *925*
- *A convergence rate estimate for the SVM Decomposition Method* *931*
- *Deriving Kernels from MLP Probability Estimators for Large Categorization Problems* *937*
- *SVM Ensembles for Selecting the Relevant Feature Subsets* *943*
- *Pattern De-Noising Based on Support Vector Data Description* *949*
- *Kernel Relevant Component Analysis for Distance Metric Learning* *954*

Plenary Poster Session: Learning and optimization

- *A new saliency measure for inputs selection and node pruning in neural network* *960*
- *One-Step Neural Network Inversion with PDF Learning and Emulation* *966*
- *Effective neural network pruning using cross-validation* *972*
- *An Efficient Learning Algorithm for Finding Multiple Solutions Based on Fixed-Point Homotopy Method* *978*
- *An Analysis of Overfitting in MLP Networks* *984*
- *Experimental Evaluation of a Hybrid Method for Configuring Ensemble Encoding Receptors* *989*
- *Dynamic construction of fault tolerant multi-layer neural networks* *995*
- *The p-Center Machine* *1000*
- *A Better Scaled Local Tangent Space Alignment Algorithm* *1006*
- *Gaussian Processes of Nonlinear Diffusion Filtering* *1012*
- *Fast Training of Multilayer Perceptrons with a Mixed Norm Algorithm* *1018*
- *Decorrelating Parametrical Neural Network* *1023*
- *Second-order backpropagation algorithms for a stagewise-partitioned separable Hessian matrix* *1027*
- *Prediction of time to event for censored data: ridge regression with linear constraints in kernel space* *1033*
- *A Threshold Varying Bisection Method for Cost Sensitive Learning in Neural Networks* *1039*
- *A Single-Layer Radial Basis Function Network Classifier and Its Applications* *1045*
- *OLS versus SVM approach to learning of RBF network* *1051*
- *Designing RBF classifiers for weighted Boosting* *1057*
- *Iterative Feature Weighting for Identification of Relevant Features with Radial Basis Function Networks* *1063*
- *Sequential Neuron Pruning Algorithm for RBF Network with Guaranteed Stability* *1069*
- *Factors of Overtraining with Fuzzy ARTMAP Neural Networks* *1075*
- *ART2 Based Classification of Sparse High Dimensional Parameter Sets For A Simulation Parameter Selection Assistant* *1081*
- *Fuzzy PSO: A Generalization of Particle Swarm Optimization* *1086*
- *Simulating A-Life Using Boltzmann Machines* *1092*
- *Artificial Cognitive BP-CT Ant Routing Algorithm* *1098*
- *Basic Property of a Quantum Neural Network Composed of Kane's Qubits* *1104*
- *Quantum Gauged Neural Networks: Learning and Recalling* *1108*

NVIDIA Ex1023
Page F-9

Plenary Poster Session: Mixture models and ensemble learning

- *Stochastic Complexity of Variational Bayesian Hidden Markov Models* 1114
- *New Experiments on Ensembles of Multilayer Feedforward for Classification Problems* 1120
- *A Research on Combination Methods for Ensembles of Multilayer Feedforward* 1125
- *Dynamically Weighted Majority Voting for Incremental Learning and Comparison of Three Boosting Based Approaches* 1131
- *A Comparison of Combination Methods for Ensembles of RBF Networks* 1137
- *New Boosting Methods of Gaussian Processes for Regression* 1142
- *Learning Probability Density Functions from Marginal Distributions with Applications to Gaussian Mixtures* 1148
- *Evaluation of Cluster Combination Functions for Mixture of Experts* 1154
- *Mixture of Heterogeneous Experts Applied to Time Series: A Comparative Study* 1160
- *Classification and Verification through the Combination of the Multi-Layer Perceptron and Auto-Association Neural Networks* 1166
- *A Classifier Ensemble Model and Its Applications* 1172

Plenary Poster Session: Self-organizing maps and associative memory

- *Systematic Rewiring in Associative Neural Networks with Small-World Architecture* 1178
- *Motion Perception with Recurrent Self-Organizing Maps Based Models* 1182
- *Retrieval Property of Associative Memory with Negative Resistance* 1187
- *A Weighted Voting Model of Associative Memory: Theoretical Analysis* 1193
- *New Results on Binary Auto- and Heteroassociative Morphological Memories* 1199
- *Self-Organizing Neural Grove and Its Applications* 1205
- *Incremental growing neural gas learns topologies* 1211
- *Robust Continuous Learning in a WTA Neural Network for Clustering Symbol Strings* 1217
- *An Associative Memory for the Online Recognition and Prediction of Temporal Sequences* 1223

Plenary Poster Session: Principal and independent component analysis

- *Similar-Image Retrieval Systems Using ICA and PCA Bases* 1229
- *Blind Inversion of Wiener System for Single Source* 1235
- *Blind Separation of Convolved Sources using the Information Maximization Approach* 1239
- *BSS Toolbox for Delayed and Convolved Mixtures* 1245
- *Diagonally Weighted and Shifted Criteria for Minor and Principal Component Extraction* 1251
- *Analysis of Signal Separation and Signal Distortion in Feedforward and Feedback Blind Source Separation Based on Source Spectra* 1257
- *Weighted Rayleigh Quotients for Minor and Principal Component Extraction* 1263

Plenary Poster Session: Probablistic and information theoretic methods

- *Feature Ranking using Supervised Neural Gas and Informational Energy* 1269
- *An Improved Kernel Fisher Discriminant Classifier and Its Applications* 1274
- *An Optimal Entropy Estimator for Discrete Random Variables* 1280
- *Comparing ``Pattern Discovery" and Back-Propagation Classifiers* 1286

NVIDIA Ex1023
Page F-10

- *Multivariate Regression Model Selection with KIC for Extrapolation Cases* *1292*
- *Robust Information Clustering* *1296*
- *Learning Nonlinear Constraints with Contrastive Backpropagation* *1302*
- *Sequential Relevance Vector Machine Learning from Time Series* *1308*

Plenary Poster Session: Neural networks and evolutionary computation

- *Performance Optimization of Function Localization Neural Network by Using Reinforcement Learning* *1314*
- *Sparse Bayesian learning and the relevance multi-layer perceptron* *1320*
- *Real-Coded Genetic Algorithm with Average-Bound Crossover and Wavelet Mutation for Network Parameters Learning* *1325*
- *A Directional Multi-resolution Ridgelet Network* *1331*
- *The Emergence of Verbs in an Artificial Life Simulation* *1337*
- *A variable-parameter neural network trained by improved genetic algorithm and its application* *1343*
- *Feature Subset Selection via Multi-Objective Genetic Algorithm* *1349*
- *A Model of Baldwin Effect in Populations of Self-Learning Agents* *1355*
- *Synthesis of Binary Cellular Automata based on Binary Neural Networks* *1361*
- *Genetic Algorithm-Based Variable Translation Wavelet Neural Network and its Application* *1365*
- *Evolution Strategies on Connection Weights into Modified Gradient Function for Multi-layer Neural Network* *1371*

Plenary Poster Session: Neural network architectures and structures

- *Neural Network Initialization with Prototypes - A Case Study in Function Approximation* *1377*
- *A Novel Radial Basis Function Network Classifier with Centers Set by Hierarchical Clustering* *1383*
- *A Simple Hierarchical Approximation RBF Neural Network* *1389*
- *The Evolving Tree, a Hierarchical Tool for Unsupervised Data Analysis* *1395*
- *Optimization of Modular Neural Networks Using Hierarchical Genetic Algorithms Applied to Speech Recognition* *1400*
- *Protein Sequence Classification Using Extreme Learning Machine* *1406*
- *Analyzing the State Space Property of Echo State Networks for Chaotic System Prediction* *1412*
- *Finding a Succinct Multi-layer Perceptron Having Shared Weights* *1418*

Tuesday, August 2, 2005

8:00 a.m.-9:00 a.m.
General Session
Functional Organization of the Primate Prefrontal Cortex for Memory 1424

9:30 a.m.-11:30 a.m.
SVM I

- *A Simple Trick for Constructing Bayesian Formulations of Sparse Kernel Learning Methods* *1425*
- *Efficient Parameter Selection for Support Vector Machines in Classification and Regression via Model-Based Global Optimization* *1431*
- *Sequential Bootstrapped Support Vector Machines - A SVM Accelerator* *1437*
- *Fbeta Support Vector Machines* *1443*
- *Time Series Filtering, Smoothing and Learning using the Kernel Kalman Filter* *1449*
- *A Combined SVM and LDA Approach for Classification* *1455*

9:30 a.m.-11:30 a.m.
Special Session: Computational Dynamical Modeling with Echo State Networks

- *Reservoir Riddles: Suggestions for Echo State Network Research* *1460*
- *Echo State Networks: Appeal and Challenges* *1463*
- *Computing with Transiently Stable States* *1467*
- *The Role of the RKH Space F in the Analysis and Design of Recurrent Neural Networks* *1473*
- *Feed-forward Echo State Networks* *1479*
- *Direct Adaptive Control: An Echo State Network and Genetic Algorithm Approach* *1483*

9:30 a.m.-11:30 a.m.
Evolutionary Algorithms and PSO

- *Non-Linear Mappings Based on Particle Swarm Optimization* *1487*
- *Evolutionary supervision of a dynamical neural network allows learning with on-going weights* *1493*
- *Evolutionary Neural Classification for Evaluation of Retail Stores and Decision Support* *1499*
- *BioAnt - Biologically Plausible Computer Simulation of an Environment with Ants* *1505*
- *Optimizing Class-Related Thresholds with Particle Swarm Optimization* *1511*
- *Lotto-Type Competitive Learning with Particle Swarm Features* *1517*

9:30 a.m.-11:30 a.m.
Special Session: Functional Neuroimaging of Cortical and Subcortical Functions

- *Reorganization and Plasticity in the Adult Brain During Learning of Motor Skills* *1523*
- *Cortico-Basal Ganglia Functional Connectivity Investigated with Transcranial Magnetic Stimulation* *1525*
- *Heading For Data-Driven Measures of Effective Connectivity in Functional MRI* *1528*
- *Connectivity of anatomical and functional MRI data* *1534*

1:00 p.m.-3:00 p.m.
Recurrent Neural Networks

- *Dynamical Consistent Recurrent Neural Networks* *1537*
- *Effectively Using Recurrently-Connected Spiking Neural Networks* *1542*
- *Adaptive Flight Control with Living Neuronal Networks on Microelectrode Arrays* *1548*
- *Introduction of an Hebbian unsupervised learning algorithm* *1552*

NVIDIA Ex1023
Page F-12

1:00 p.m.-3:00 p.m.
Self-Organizing Maps

- *A visualization technique for Self-Organizing Maps with vector fields to obtain the cluster structure at desired levels of detail* *1558*
- *Multi-Topographic Neural Network Communication and Generalization for Multi-Viewpoint Analysis* *1564*
- *A Self-Organizing Map for Concept Classification in Information Retrieval* *1570*
- *Training the SOFM Efficiently: An Example from Intrusion Detection* *1575*
- *Modular Network SOM (mnSOM): From Vector Space to Function Space* *1581*
- *Ranked Centroid Projection: A Data Visualization Approach for Self-organizing Maps* *1587*

1:00 p.m.-3:00 p.m.
Special Session: Transition: Imaging and Cortical Models

- *fMRI experiments and computational models of the function of the prefrontal cortex and the basal ganglia: a review* *1593*
- *Computational Models of Emotion* *1598*
- *Neural Networks of the Brain: Their Analysis and Relation to Brain Images* *1603*
- *Brain Categorization: Learning, Attention, and Consciousness* *1609*

1:00 p.m.-3:00 p.m.
Special Session: Applications of Learning and Data-Driven Methods to Earth Sciences and Climate Modeling

- *Complex Hybrid Models Combining Deterministic and Machine Learning Components as a New Synergetic Paradigm in Numerical Climate Modeling and Weather Prediction* *1615*
- *Atmospheric correction and oceanic constituents retrieval with a neuro-variational method* *1621*
- *Nonlinear Complex Principal Component Analysis and Its Applications* *1626*
- *Nonlinear Principal Predictor Analysis Using Neural Networks* *1630*
- *Temporal Neural Networks for Downscaling Climate Variability and Extremes* *1636*
- *A Spatiotemporal Approach to Tornado Prediction* *1642*

3:20 p.m.-5:20 p.m
Special Session: Models of Cortical and Subcortical Circuits

- *A computational model of reach decisions in the primate cerebral cortex* *1648*
- *Modeling Cortico-Subcortical Interactions During Planning, Learning, and Voluntary Control of Actions* *1653*
- *Modeling Emotional Influences on Human Decision Making Under Risk* *1657*
- *Modelling the Interaction of Attention and Emotion* *1663*

NVIDIA Ex1023
Page F-13

3:20 p.m.-5:20 p.m.
Visual and Image Processing

- *Neural Network Model for Analyzing Optic Flow* *1669*
- *Face Detection and Identification Using a Hierarchical Feed-forward Recognition Architecture* *1675*
- *Lossless High Dynamic Range Image Coding based on Lifting Scheme using Nonlinear Interpolative Effect of Discrete-Time Cellular Neural Networks* *1681*
- *Could Early Visual Processes be Sufficient to Label Motions* *1687*
- *A Phase-equation Model for a Large Phase Lead in Manual Tracking Caused by Intermittent Visual Information* *1693*
- *Investigations into the Analysis of Remote Sensing Images with a Growing Neural Gas* *1698*

3:20 p.m.-5:20 p.m.
Special Session: Applications of Learning and Data-Driven Methods to Earth Sciences and Climate Modeling (plus Panel Discussion)

- *Applications of Neural Network Methods to the Processing of Earth Observation Satellite Data* *1704*
- *Time Dependent Neural Network Models For Detecting Changes Of State In Earth and Planetary Processes* *1710*
- *Local and Hybrid Learning Models in Forecasting Natural Phenomena* *1716*
- *Semiblind source separation of climate data detects El Nino as the component with the highest interannual variability* *1722*
- *Panel discussion: Applications of Learning and Data-Driven Methods to Earth Sciences and Climate Modeling* *1728*

3:20 p.m.-5:20 p.m.
Diagnostics and Control, Power Systems

- *Continuous On-line Identification of Nonlinear Plants in Power Systems with Missing Sensor Measurements* *1729*
- *Collective Behavior Implementation in Powerline Surveillance Sensor Network* *1735*
- *Neural Networks based Scheme for Fault Diagnosis in Fossil Electric Power Plants* *1740*
- *Detection of Actuator Faults Using a Dynamic Neural Network for the Attitude Control Subsystem of a Satellite* *1746*
- *Dynamic Bayesian Networks for Machine Diagnostics: Hierarchical Hidden Markov Models vs. Competitive Learning* *1752*
- *Fuzzy Multi-Hidden Markov Predictor in Electric Load Forecasting* *1758*

7:00 p.m.-11:00 p.m.
Plenary Interactive Session II

Plenary Poster Session: Fuzzy neural systems

- *Flexible Takagi-Sugeno Fuzzy Systems* *1764*
- *A New Scene Analysis Using Genetic Algorithm Based Fuzzy ID3 Method* *1770*
- *A Comparative Study of the IDS method and Feedforward Neural Networks* *1776*

NVIDIA Ex1023
Page F-14

Plenary Poster Session: Vision and image processing

- *Edge Inference for Image Interpolation* 1782
- *Image Recognition Systems with Permutative Coding* 1788
- *A Modified Frequency Domain Cross Correlation Implemented in MATLAB for Fast Sub-Image Detection Using Neural Networks* 1794
- *A Motion Trajectory Based Video Retrieval System Using Parallel Adaptive Self Organizing Maps* 1800
- *Learning Informative Features for Spatial Histogram-Based Object Detection* 1806
- *A Hierarchical Bayesian Model of Invariant Pattern Recognition in the Visual Cortex* 1812
- *Text Extraction from Name Cards Using Neural Network* 1818
- *Visual Deficiency: Cognitive Performance and Adaptive Image Processing* 1824
- *On Using an Associative Memory for Improving Digital Color Images: Color Characterization, Enhancement, and Color Balancing* 1830
- *General Design Approach to Unit-linking PCNN for Image Processing* 1836
- *Model of Neuron-Like Systems.Examples of Dynamic Processes* 1842
- *Temporal Hand Gesture Recognition By Fuzzified TSK-Type Recurrent Fuzzy Network* 1848
- *Using Knowledge of the Region of Interest (ROI) in Automatic Image Retrieval Learning* 1854
- *Using Independent Subspace Analysis to Build Independent Spectral Representations of Images* 1860
- *Natural Image Compression Using An Extended Non-Negative Sparse Coding Neural Network Technique* 1866
- *Object Recognition Using Neurocomputing and Conformal Computing Geometry* 1872

Plenary Poster Session: Biometric, handwriting and other pattern recognition

- *A Robust Deterministic Annealing Algorithm for Data Clustering* 1878
- *Pruning Support Vectors for Imbalanced Data Classification* 1883
- *Normalized Neural Networks for Fast Pattern Detection* 1889
- *Optimal Kernels for Unsupervised Learning* 1895
- *Fast Pattern Detection Using Neural Networks and Cross Correlation in the Frequency Domain* 1900
- *Estimating Accurate Multi-class Probabilities with Support Vector Machines* 1906
- *Modular General Hyperline Segment Neural Network* 1912
- *Fourier Fuzzy Neural Network for Clustering of Objects Based on the Gross Shape and its Application to Handwritten Character Recognition* 1918
- *Third-order generalization and a new approach to systematically categorizing higher-order generalization* 1924
- *Low Complexity Iris Recognition Based on Wavelet Probabilistic Neural Networks* 1930
- *Efficient Video Object Classifier using Locality-Enhanced Support Vector Machines* 1936
- *A Comparative Analysis of the Performance of Hybrid and Non-Hybrid Multi-classifier Systems* 1941
- *Pattern Classification by Assembling Small Neural Networks* 1947
- *A Symbolic Approach to the Solution of F-Classification Problems* 1953
- *Optical Flame Detection Using Large-Scale Artificial Neural Networks* 1959

NVIDIA Ex1023
Page F-15

- *An Experimental Study of Several Decision Issues for Feature Selection with Multi-Layer Perceptrons* 1965
- *Spectral Feature Analysis* 1971
- *Self-Adaptive Kernel Machine: Online Clustering in RKHS* 1977
- *An Algorithm for Pruning Redundant Modules in Min-Max Modular Network* 1983
- *A Novel Fuzzy Clustering Neural Network* 1989

Plenary Poster Session: Face and texture recognition

- *Performance of Neural Classifiers for Fabric Faults Classification* 1995
- *Orientated Texture Segmentation for Detecting Defects* 2001
- *Modification of Correlation Kernels in SVM, KPCA and KCCA in Texture Classification* 2006
- *Neural Network-based Shape Recognition using Generalized Differential Evolution Training Algorithm* 2012
- *The Effect of Normal Adult Aging on Standard PCA Face Recognition Accuracy Rates* 2018
- *Face Recognition Method Independent of Rotation and Size Variations* 2024
- *Combined Subspace Method Using Global and Local Features for Face Recognition* 2030

Plenary Poster Session: Audition, speech recognition and production

- *Low Order Modeling for Multiple Moving Sound Synthesis using Head-related Transfer Functions' Principal Basis Vectors* 2036
- *Individualized HRTFs From Few Measurements: a Statistical Learning Approach* 2041
- *Framewise Phoneme Classification with Bidirectional LSTM Networks* 2047
- *Discriminative Training of Hidden Markov Models by Multiobjective Optimization for Visual Speech Recognition* 2053
- *Harmonic Envelope Prediction for Realistic Speech Synthesis Using Kernel Interpolation* 2059
- *A Soft Bayes Perceptron* 2064
- *Chaos and Speech Rhythm* 2070
- *Exploration of Rank Order Coding with Spiking Neural Networks for Speech Recognition* 2076

Plenary Poster Session: Nose, tactile, taste and other senses

- *A comparative study of neural network to artificial noses* 2081
- *Optimization of a learning algorithm for tactile pattern generation* 2087
- *A Combinative Function Approximation Model and Its Applications to Electronic Noses* 2093
- *A Novel Hybrid Learning Scheme For Pattern Recognition* 2099

Plenary Poster Session: Cognitive function

- *Convergence of Coherent Components of Neural Networks by Positive Correlation Learning* 2105
- *Learning of an XOR Problem in the Presence of Noise and Redundancy* 2111
- *A Multiple BAM for Hetero-association and Multisensory Integration Modelling* 2117
- *Effect of Curriculum on the Consolidation of Neural Network Task Knowledge* 2123

NVIDIA Ex1023
Page F-16

- *Creation of Long-term Mental Representations using the Dimension of Fractal dendrites* *2129*
- *Neurobiological Data Sustaining Opponent Processing Operations on Self Organizing Networks as Tools for the Modeling of Hippocampal Dynamics* *2135*
- *Neural Network Approaches to Personal Change in Psychotherapy* *2139*

Plenary Poster Session: Computational neuroscience

- *Role of Presynaptic Reuptake on Dopamine Modulation of Cortico-striatal Activity in TD Learning* *2145*
- *Two state transitions mediated by spontaneous and/or intensive stimulus on an integrate-and-fire neural network model* *2150*
- *Learning with Single Integrate-and-Fire Neuron* *2156*
- *Optimizing Neural Model Templates using Covariance Matrix Adaptation and Fourier Analysis* *2162*
- *Fast Constructive-Covering Approach for Neural Networks* *2167*
- *Ensembles of membrane proteins as statistical mixed-signal computers* *2173*
- *Towards the Modeling of Dissociated Cortical Tissue in the Liquid State Machine Framework* *2179*
- *The Bifurcating Neuron Network* *2184*
- *Spiking Neural Network Training Using Evolutionary Algorithms* *2190*
- *Increased Swimming Control with Evolved Lamprey CPG Controllers* *2195*

Plenary Poster Session: Signal processing and time series analysis

- *Modified Time-Based Multilayer Perceptron for Sensor Networks and Image Processing Applications* *2201*
- *On-line Bayesian Change Detection Scheme for Unknown Nonlinear Systems via Sequential Monte Carlo* *2207*
- *Independent Component Analysis Applications in Physics* *2213*
- *Embedded FastICA Algorithm Applied to the Sensor Noise Extraction Problem of Foundation Fieldbus Network* *2217*
- *Design of Neural Predictors using Tools of Chaos Theory and Bayesian Learning* *2222*
- *Harmonic Detection using Wavelet Transforms* *2228*
- *A Recurrent Neural Network for Sound-Source Motion Tracking and Prediction* *2232*

Plenary Poster Session: Control and robotics

- *Artificial neural networks associated to calorimetry to preview polymer composition of high solid content emulsion copolymerizations* *2237*
- *Supply Air Temperature Control of AHU with a Cascade Control Strategy and a SPSA Based Neural Controller* *2243*
- *Design and stabilization of sampled-data neural-network-based control systems* *2249*
- *Real-time Control of Variable Air Volume System Using a SPSA Based Neural Controller* *2255*
- *Discrete-Time Systems Neuro-Riccati Equation Solution* *2261*
- *Predictive Control Based on Feedforward Neural Network for Strong Nonlinear System* *2266*
- *Design and Experimental Evaluation of a 3-Mass Speed Control System with a Hybrid Structure of Sliding Mode Controller and CMAC* *2272*

NVIDIA Ex1023
Page F-17

- *Total Sliding-mode-based Genetic Algorithm Control for Linear Piezoelectric Ceramic Motor Drive* *2278*
- *Adaptive Tanaka-Sugeno Fuzzy Cerebellar Model Articulation Controller for Output Tracking Control* *2284*
- *Robust Output Tracking CMAC Control: The T-S Fuzzy Model-Based Approach* *2290*
- *Prototype Robotic Arm Assisted by Smart CMOS Vision Chips* *2296*

Plenary Poster Session: Diagnostics and quality control

- *A framework for the verification of air quality forecasting models using self-organizing feature maps* *2302*
- *Bayesian Neural Networks for Nonlinear Multivariate Manufacturing Process Monitoring* *2308*
- *Feature Selection and Condition Monitoring of Gearbox Using SOM* *2313*
- *Engine Data Classification with Simultaneous Recurrent Network Using a Hybrid PSO-EA Algorithm* *2319*
- *An Automatic Method to Detect Missing Components in Manufactured Products* *2324*
- *Cosmetic Defect Classification Found in Ophthalmic Lenses Using Artificial Neural Networks* *2330*
- *A Neuro-Fuzzy Based Sensor and Actuator Fault Estimation Scheme for Unknown Nonlinear Systems* *2335*
- *Aircraft Cabin Noise Minimization via Neural Network Inverse Model* *2341*
- *Dynamic Neural Network-based Estimator for Fault Diagnosis in Reaction Wheel Actuator of Satellite Attitude Control System* *2347*

Plenary Poster Session: Telecommunication applications

- *Modeling of Spiral Inductors Using Artificial Neural Network* *2353*
- *Sparse Channel Estimation With Regularization Method Using Convolution Inequality For Entropy* *2359*
- *A Neural Phase Rotator for PAPR Reduction of PCC-OFDM Signal* *2363*
- *Further Results on the EKF-CRTRL Equalizer for Fast Fading and Frequency Selective Channels* *2367*
- *Hybrid Neural-Phenomenological Sub-Models and its Application to Earth-space Path Signal Attenuation Prediction* *2372*
- *Application of Neural Networks to 3G Power Amplifier Modeling* *2378*

Plenary Poster Session: Heart applications

- *Bayesian ANN Classifier for ECG Arrhythmia Diagnostic System: A Comparison Study* *2383*
- *Input and Data Selection Applied to Heart Disease Diagnosis* *2389*
- *Reliable Determination of Sleep Versus Wake from Heart Rate Variability Using Neural Networks* *2394*
- *Neural Network Based Detection of Fetal Heart Rate Patterns* *2400*

Plenary Poster Session: Pharmacology and disease diagnosis

- *Exploratory Data Analysis of Dynamic Cerebral Contrast—Enhanced Perfusion MRI Time—Series* *2406*

NVIDIA Ex1023
Page F-18

- *A Neural Network-based Prediction Model of AR Inhibitory Activity from a Sparse Set of Compounds* *2411*
- *Multi-class Support Vector Machines for Modeling HIV/AIDS Treatment Adherence using Patient Data* *2417*
- *Neural network-based approach for the classification of wireless-capsule endoscopic images* *2423*
- *The Use of Clustering to Analyze Symptom-Based Case Definitions for Acute Gastrointestinal Illness* *2429*
- *A SVM Approach for Detection of Hemorrhages in Background Diabetic Retinopathy* *2435*
- *QRS Complex Detection Using Backpropagation Neural Network (no files)*
- *Dynamic Feature Fusion in the Self Organising Tree Map - applied to the segmentation of Biofilm Images* *2441*

Plenary Poster Session: Brain imaging and brain interfaces

- *Independent Component Analysis and High-Order Statistics for Automatic Artifact Rejection* *2447*
- *A Neural Supergraph Matching Architecture* *2453*
- *Comparison of TDNN Training Algorithms in Brain Machine Interfaces* *2459*
- *An Algorithm for Automatic Assignment of Artifact-Related Independent Components in Biomedical SIgnal Analysis* *2463*

Plenary Poster Session: Application to cancer diagnostics

- *Computational Intelligence Techniques for Acute Leukemia Gene Expression Data Classification* *2469*
- *Virtual Reality Visual Data Mining with Nonlinear Discriminant Neural Networks: Application to Leukemia and Alzheimer Gene Expression Data* *2475*
- *SVMs Applied to Objective Aesthetic Evaluation of Conservative Breast Cancer Treatment* *2481*
- *Classification of Breast Abnormalities in Digital Mammograms using Image and BI-RADS Features in Conjunction with Neural Network* *2487*
- *Feasibility of multi-layered perceptron network in discriminating breast magnetic resonance imaging lesions* *2493*
- *Automatic recognition of the blood cells of myelogenous leukemia using SVM* *2496*

Plenary Poster Session: Electroencephalograms and electromyograms

- *Neural Network Classification of EEG Signals using Time-Frequency Representation* *2502*
- *ICA And a Gauge of Filter for the Automatic Filtering of an EEG Signal* *2508*
- *On the Use of Clustering and Local Singular Spectrum Analysis to Remove Ocular Artifacts from Electroencephalograms* *2514*
- *A Fuzzy Approach for Key Variables Identification of EMG Evaluation System* *2520*
- *Extraction of event related potentials from EEG signals using ICA with reference* *2526*
- *PCA and ICA for the Extraction of EEG Dominant Components in Cerebral Death Assessment* *2532*

NVIDIA Ex1023
Page F-19

Plenary Poster Session: Financial and marketing applications

- *Vector Quantized Radial Basis Function Neural Network with Embedded Multiple Local Linear Models for Financial Prediction* 2538
- *Application of Multi-Branch Neural Networks to Stock Market Prediction* 2544
- *Data Preprocessing for Stock Market Forecasting using Random Subspace Classifier Network* 2549
- *Predicting Customer Behavior via Calling Links*....2555

Plenary Poster Session: Computer security, military and biomimetic applications

- *A Novel Image Retrieval System Based on BP Neural Network* 2561
- *Speaker Identification Using Speech and Lip Features* 2565
- *Comparison of a SOM Based Sequence Analysis System and Naive Bayesian Classifier for Spam Filtering* 2571
- *Neural Networks Following a Binary Approach Applied to the Integer Prime-Factorization Problem* 2577
- *Artificial Intelligence for Conflict Management* 2583
- *Fingerprint Recognition Using Modular Neural Networks and Fuzzy Integrals for Response Integration*....2589
- *Chaotic Associative Memory and Private V-mails* 2595
- *Using Domain Knowledge to Constrain Structure Learning in a Bayesian Bioagent Detector* 2601

Plenary Poster Session: Power systems applications

- *Power System Reduced Model by Artificial Neural Networks* 2607
- *FPGA Realization of Power Quality Disturbance Detection: An Approach with Wavelet, ANN and Fuzzy Logic*....2613
- *Short Term Load Forecasting for Iran National Power System and Its Regions Using Multi Layer Perceptron and Fuzzy Inference systems* 2619
- *Artificial Neural Networks for Temporal Processing Applied to Prediction of Electric Energy Generation in Small Hydroelectric Power Stations* 2625
- *Streamflow forecasting using neural networks and fuzzy clustering techniques* 2631

Plenary Poster Session: Special Session - Earth Sciences & Climate modeling

- *Neural Network Modelling of the 20-Year Flood Event for 850 Catchments Across the UK*....2637
- *Modelling Harbour Sedimentation Using ANN and M5 Model Trees* 2643
- *Classification of Infrasound Events Using Using Radial Basis Function Neural Networks* 2649
- *Neural Network River Discharge Forecasters: An Empirical Investigation of Hidden Unit Processing Functions Based on Two Different Catchments*....2655
- *Robustness of the NN Approach to Emulating Atmospheric Long Wave Radiation in Complex Climate Models* 2661
- *A Comparative Study of Artificial Neural Network Techniques for River Stage Forecasting*....2666
- *Issues in Designing Automated Minimal Resource Allocation Neural Networks* 2671
- *Wavelet Networks: An Alternative to Classical Neural Networks*....2674
- *Streamflow Forecasting with Uncertainty Estimate Using Bayesian Learning for ANN*....2680

NVIDIA Ex1023
Page F-20

- *Intelligent systems for meteorological events forecas 2686*
- *A multilayer perceptron approach for the retrieval of vertical temperature profiles from satellite radiation data 2689*
- *Machine Learning in Soil Classification 2694*
- *Estimation of Prediction Intervals for the Model Outputs using Machine Learning 2700*

Plenary Poster Session: Earth Sciences and Climate modelling

- *Determination of optimal batch size in incremental approach for tornado detection 2706*
- *Locally Recurrent Neural Networks Optimal Filtering Algorithms: Application to Wind Speed Prediction Using Spatial Correlation 2711*

Wednesday, August 3, 2005

8:00 a.m.-9;00 a.m.
General Session
Neural Networks for Feedback Control of Robots and Dynamical Systems 2717

9:30 a.m.-11:30 a.m.
Data Mining

- *Automatic Source Attribution: A Neural Networks Approach 2718*
- *Cross-Entropy Approach to Data Visualization Based on the Neural Gas Network 2724*
- *A Model of Document Clustering Using Ant Colony Algorithm and Validity Index.....2730*
- *A method to extract Non-Linear Principal Components of Large Datasets - An application in Skill Transfer 2736*
- *Identifying knowledge domain and incremental new class learning in SVM 2742*
- *Real-World Text Clustering with Adaptive Resonance Theory Neural Networks 2748*

9:30 a.m.-11:30 a.m.
Special Session: Hebb's Legacy

- *Hebbian Synaptic Modification in Cortical Circuits and Memory-Guided Behavior in Spatial Alternation and Delayed Non-Match to Position 1754*
- *Faithful Retinotopic Maps with Local Optimum Rules, Axonal Competition, and Hebbian Learning.....2760*
- *Modeling Structural Plasticity in the Barn Owl Auditory Localization System with a Spike-Time Dependent Hebbian Learning Rule.....2766*
- *Hebbian motor control in a robot-embedded model of habituation 2772*

9:30 a.m.-11;30 a.m.
Robotics

- *Constraints on Body Movement during Visual Development Affect Behavior of Evolutionary Robots 2778*

- *Dual Kalman Filters for Autonomous Terrain Aided Navigation in Unknown Environments* *2784*
- *New Memory Model for Humanoid Robots -Introduction of Co-Associative Memory Using Mutually Coupled Chaotic Neural Networks* *2790*
- *Bi-Criteria Torque Optimization of Redundant Manipulators Based on a Simplified Dual Neural Network* *2796*
- *Spatio-Temporal Neural Data Mining Architecture in Learning Robots* *2802*
- *Fault-tolerance by Regeneration: Using Development to Achieve Robust Self-Healing Neural Networks* *2808*

9:30 a.m.-11:30 a.m.
SVM II

- *Maximal Variation and Missing Values for Componentwise Support Vector Machines* *2814*
- *Fast Bayesian Support Vector Machine Parameter Tuning with the Nystrom Method* *2820*
- *Maximum Margin Classifiers with Noisy Data: A Robust Optimization* *2826*
- *Data Classification with a Relaxed Model of Variable Kernel Density Estimation* *2831*
- *A Ridgelet Kernel Approach for Regression using Particle Swarm Optimization Algorithm* *2837*

1:00 p.m.-3:00 p.m.
Hardware

- *Ultra Low Power Fault Tolerant Neural Inspired CMOS logic* *2843*
- *Noise Performance of Single-Electron Depressing Synapses for Neuronal Synchrony Detection* *2849*
- *A New Pulse Mode Self Organizing Map Hardware with Digital Phase Locked Loops* *2855*
- *A novel Approach to reduce Interconnect Complexity in ANN Hardware Implementation* *2861*
- *A Reconfigurable Parallel Architecture for SVM Classification* *2867*

1:00 p.m.-3:00 p.m.
Cognitive Function

- *An Integrate and Fire Model of Prefrontal Cortex provides a Biological Implementation of Action Selection in Reinforcement Learning Theory that Reuses Known Representations* *2873*
- *Knowing your place: Subfield specific involvement in hippocampal spatial processing* *2879*
- *Coherent learning in cortical maps: A generic approach* *2885*
- *Using Neural Dynamics to Switch Attention* *2891*
- *A Continuous Attractor Neural Network Model of Divided Visual Attention* *2897*
- *Bridging the Gap Between Vision and Language: A Morphodynamical Model of Spatial Categories* *2903*

NVIDIA Ex1023
Page F-22

1:00 p.m.-3:00 p.m.
Special Session: Performance of Neuro-Adaptive and Learning Systems: Assessment, Monitoring and Validation

- *Challenges in verification and validation of autonomous systems for space exploration* 2909
- *Rule Extraction as a Formal Method for the Verification and Validation of Neural Networks* 2915
- *An Approach To Predicting Non-Deterministic Neural Network Behavior* 2921
- *Testing Decision Systems with Classification Components* 2927
- *Validity Index in Dynamic Cell Structures* 2931
- *A Disturbance Rejection based Neural Network Algorithm for Control of Air Pollution Emissions* 2937

1:00 p.m.-3:00 p.m.
Special Session: Constructive/Hierarchical Self-Organizing Maps

- *A Gradient Descending Solution to the LASSO Criteria* 2942
- *A Constructive and Hierarchical Self-Organising Model in A Non-Stationary Environment* 2948
- *Investigation of Alternative Strategies and Quality Measures for Controlling the Growth Process of the Growing Hierarchical Self-Organizing Map* 2954
- *A hierarchical hybrid neural model with time integrators in long-term peak-load forecasting* 2960
- *Introduction to the SAM-SOM* and MAM-SOM* Families* 2966

3:20 p.m.-5:20 p.m.
Special Session: Approximate Dynamic Programming

- *Reinforcement Learning and the Frame Problem* 2971
- *An Analysis of Gradient-Based Policy Iteration* 2977
- *An Integrated Fault Tolerant Control Framework Using Adaptive Critic Design* 2983
- *Approximate Dynamic Programming for High Dimensional Resource Allocation Problems* 2989
- *Modeling Reward Functions for Incomplete State Representations via Echo State Networks* 2995
- *Continuous Adaptive Critic Desigs* 3001

3:20 p.m.-5:20 p.m.
Biomedical Applications

- *Classification of Plantar Pressure and Heel Acceleration Patterns Using Neural Networks* 3007
- *Feature Selection by Independent Component Analysis and Mutual Information Maximization in EEG Signal Classification* 3011
- *Neural Network and Principal Component Analyses of Highly Variable Myocardial Mechanical Waveforms Derived from Echocardiographic Ultrasound Images* 3017
- *A Recurrent Fuzzy-Neural Filter for Real-Time Separation of Lung Sounds* 3023
- *Stochastic Feature Selection for the Discrimination of Biomedical Spectra* 3029
- *Ovarian Cancer Diagnosis Using Complementary Learning Fuzzy Neural Network* 3034

NVIDIA Ex1023
Page F-23

3:20 p.m.-5:20 p.m.
Fuzzy-Neural Systems

- *Fuzzy ROC Curves for Unsupervised Nonparametric Ensemble Techniques 3040*
- *Neural Network with Fuzzy Dynamic Logic 3046*
- *Color Image Segmentation Using Fuzzy Min-Max Neural Networks 3052*
- *Skin Color Segmentation by Histogram-Based Neural Fuzzy Network 3058*
- *Learning a Hierarchical Fuzzy System with Autonomous Navigation as an Example.....3063*

3:20 p.m.-5:20 p.m.
Special Session: Biologically Inspired Computational Vision

- *A Biologically Inspired Computational Vision Front-end based on a Self-Organised Pseudo-Randomly Tessellated Artificial Retina.....3069*
- *A Chaos Synchronization-Based Dynamic Vision Model for Image Segmentation 3075*
- *Processing Landsat TM Data Using Complex-Valued NRBF Neural Network.....3081*
- *Comparison of Linear and Non-linear Data Projection Techniques in Recognizing Universal Facial Expressions.....3087*
- *Associative neural networks as means for low-resolution video-based recognition 3093*

Thursday, August 4, 2005

8:00 a.m.-9:00 a.m.
General Session
Beyond Correlation – Closing the Loop Between Brain and Theory by Extracting Representations and Altered Feedbacks 3099

9:30 a.m.-11:30 a.m.
Special Session: Neural Prostheses and the Neuron-Silicon Interface

- *Artificial Vision by Electrical Stimulation of the Retina 3100*
- *Physiological Activation of the Hind Limb Muscles of the Anesthetized Cat Using the Utah Slanted Electrode Array 3103*
- *Replacing Damaged Brain Regions with Biomimetic Microelectronic Neural Prostheses to Restore Cognitive Function 3109*
- *Getting Better Signals Out of the Brain: Decoding Algorithms and Autonomous Electrodes 3115*
- *Application of Polymer Microstructures with Controlled Surface Chemistries as a Platform for Creating and Interfacing with Synthetic Neural Networks 3116*

9:30 a.m.-11:30 a.m.
Applications I

- *An Experimental Comparison of Semi-Supervised ARTMAP Architectures, GCS and GNG Classifiers.....3121*
- *Time Series Study of GGAP-RBF Network: Predictions of Nasdaq Stock and Nitrate Contamination of Drinking Water 3127*
- *Environmental Informatics - Long-Lead Flood Forecasting Using Bayesian Neural Networks.....3133*
- *Detection of Disease Outbreaks in Pharmaceutical Sales: Neural Networks and Threshold Algorithms.....3138*

- *A Fast Neural Network-Based Detection and Tracking of Dim Moving Targets in FLIR Imagery* *3144*
- *GETnet: A General Framework for Evolutionary Temporal Neural Networks* *3150*

9:30 a.m.-11:30 a.m.
Neurodynamics

- *Temporal Discontinuities in Neocortical Dynamics* *3156*
- *Stability Conditions of the full KII Model of Excitatory and Inhibitory Neural Populations* *3162*
- *Associative Memory in the Strong Coupling Regime of a K-set Oscillator Network (no file)*
- *Evolving High Capacity Associative Memories with Efficient Wiring* *3168*

9:30 a.m.-11:30 a.m.
Learning I

- *Incremental Learning for Online Face Recognition* *3174*
- *Efficient Online Spherical K-means Clustering* *3180*
- *On Efficient Selection of Binary Classifiers for Min-Max Modular Classifier* *3186*
- *Are ARIMA Neural Network Hybrids Better than Single Models* *3192*
- *Embedding via clustering: Using spectral information to guide dimensionality reduction* *3198*
- *On the Evaluation of Relevance Learning by a Multi-layer Perceptron* *3204*

1:00 p.m.-3:00 p.m.
Telecommunications

- *A Hybrid Neural Network for Optimal TDMA Transmission Scheduling in Packet Radio Networks* *3210*
- *Nonlinear Channel Equalization with QAM Signal Using Chebyshev Artificial Neural Network* *3214*
- *Implementation of an MLP-based DOA System Using a Reduced Number of MM-wave Antenna Elements* *3220*
- *Least-Squares Support Vector Machines for DOA Estimation: A Step-by-Step Description and Sensitivity Analysis* *3226*
- *Recurrent Neural Equalization for Communication Channels in Impulsive Noise Environments* *3232*

1:00 p.m.-3:00 p.m.
Neuromorphic Hardware

- *A New Hybrid Neural System Interfacing Neurons and Silicon Hardware for Fast Signal Recognition* *3238*
- *Biologically plausible VLSI neural network implementation with asynchronous neuron and spike-based synapse* *3244*
- *Novel Digital Spiking Neuron and its Pulse-Coupled Network: Spike Position Coding and Multiplex Communication* *3249*
- *Digital Implementation of a Bio-inspired Neural Model for Motion Estimation* *3255*
- *Emulation Engine for Spiking Neurons and Adaptive Synaptic Weights* *3261*
- *An Orientation-Selective Multi-chip aVLSI Applicable to Texture Analysis* *3267*

NVIDIA Ex1023
Page F-25

1:00 p.m.-3:00 p.m.
Applications II
- *Information-Theoretic Feature Selection Algorithms for Text Classification* 3272
- *SOM Based Image Data Structuring in an Augmented Reality Scenario* 3278
- *Forecasting Energy Product Prices* 3284
- *Reinforcement Learning Approach to Individualization of Chronic Pharmacotherapy*.....3290
- *"Cognitive" Memory* 3296

1:00 p.m.-3:00 p.m.
Learning II
- *Hierarchical Fast Learning Artificial Neural Network* 3300
- *Non-Homogenous Structures in Neural Networks with Chaotic Recursive Nodes: Dealing with Diverse Multi-assemblies Architectures, Connectivity and Arbitrary Bifurcating Nodes* 3306
- *Improved Chaotic Neuro-Computer with Output-Coding for Quadratic Assignment Problems*.....3312
- *Chaotic Associative Memory using Internal Patterns for Image Retrieval by Color and Shape Information* 3318
- *Transient Dynamics of Non-Linear Models Describing Multi-Stable Stochastic Systems* 3324
- *How to Find Different Neural Networks by Negative Correlation Learning* 3330

3:20 p.m.-4:20 p.m.
General Session
Neuromophic Engineering: Overview and Potential 3334

NVIDIA Ex1023
Page F-26

ATTACHMENT G

NVIDIA Ex1023
Page G-2

IJCNN 2005: SESSION GRID

Sunday, July 31, 2005				
Time	**Lachine**	**Hampstead**	**Mont Royal**	**Verdun**
9:00 a.m.	Evolutionary Robotics		Bioinfomatics and Machine Learning: The Prediction of Protein Structures on a Genomic Scale	
11:00 a.m.	Break			
11:15 a.m.	Neural Networks that Actually Work in Prediction and Decision/Control: Common Misconceptions Versus Real-World Success	Evolving Connectionist Systems for Adaptive Learning and Knowledge Discovery: Principles, Models and Applications	Integrating Language and Cognition: New Results in Computational Intelligence	Feature Extraction in Computational Intelligence
1:15 p.m.	Break			
2:30 p.m.	Neural Networks for Dynamic Systems Feedback	Data Visualization of High Dimensional Scientific Data	New Formulations for Predictive Learning	Unsupervised Learning
4:30 p.m.	Break			
4:45 p.m.	Cognitive Memory	Biologically Plausible Artificial Neural Networks	Support Vector Machines and Kernel Based Learning	Nonlinear Manifolds in Pattern Recognition and Image Analysis
6:45 p.m.	End of the Day			

Monday, August 1, 2005				
Time	**Westmount**	**Outremont**	**Mont Royal/Hampstead**	**Verdun/Lachine**
8:00 a.m.	Plenary: Exploring Chemical Space with Computers: Challenges and Opportunities Professor Pierre Baldi, Director, Institute for Genomics and Bioinformatics, University of California, Irvine			
9:00 a.m.	Break			
9:30 a.m.	Special Session: Neural Networks Applications to Bioinformatics	Information-Theoretic and Bayesian Learning	Special Session: Neurodynamics and Intentional Dynamic Systems	Pattern Recognition
11:30 a.m.	Break			
1:00 p.m.	Bioinformatics	Special Session: Evolvable and Emergent Neural Systems	Models of Neurons, Local Circuits and Systems	Independent Component Analysis and Principal Component Analysis
3:00 p.m.	Break			
3:20 p.m.	Special Session: Computational Neurogenetic Modeling	Control and System Identification	Spiking Neurons	Special Session: Recent Advancements in Adaptive Resonance Theory
5:20 p.m.	Break			
7:00 p.m.	Plenary Poster Session - Fontaine Ballroom			
11:00 p.m.	End of the Day			

Tuesday, August 2, 2005				
Time	**Westmount**	**Outremont**	**Mont Royal/Hampstead**	**Verdun/Lachine**
8:00 a.m.	Functional Organization of the Primate Prefrontal Cortex for Memory Michael Petrides, Professor, Psychology Department/Neurology and Neurosurgery, McGill University and Director, Neuropsychology/Cognitive Neuroscience Unit Montreal, Neurological Institute and Hospital			
9:00 a.m.	Break			
9:30 a.m.	Evolutionary Algorithms and PSO	Special Session: Computational Dynamical Modeling with Echo State Networks	Special Session: Functional Neuroimaging of Cortical and Subcortical Functions	Support Vector Machines I
11:30 a.m.	Break			
1:00 p.m.	Special Session: Applications of Learning and Data-Driven Methods to Earth Sciences and Climate Modeling	Recurrent Neural Networks	Special Session: Transition: Imaging and Cortical Models	Self-Organizing Maps
3:00 p.m.	Break			
3:20 p.m.	Special Session: Applications of Learning and Data-Driven Methods to Earth Sciences and Climate Modeling, Plus Panel Discussion	Diagnostics and Control, Power Systems	Special Session: Models of Cortical and Subcortical Circuits	Visual and Image Processing
5:20 p.m.	Break			
7:00 p.m.	Plenary Poster Session - Fontaine Ballroom			
11:00 p.m.	End of the Day			

NVIDIA Ex1023
Page G-5

Wednesday, August 3, 2005				
Time	**Westmount**	**Outremont**	**Mont Royal/Hampstead**	**Verdun/Lachine**
8:00 a.m.	Plenary: Neural Networks for Feedback Control of Robots and Dynamical Systems Professor Frank L. Lewis, Head, Advanced Controls, Sensors and MEMS Group, Automation and Robotics Research Institute, The University of Texas at Arlington			
9:00 a.m.	Break			
9:30 a.m.	Robotics	Support Vector Machine II	Special Session: Hebb's Legacy	Data Mining
11:30 a.m.	Break			
1:00 p.m.	Hardware	Special Session: Performance of Neuro-Adaptive and Learning Systems: Assessment, Monitoring and Validation	Cognitive Function	Special Session: Constructive/Hierarchical Self-Organizing Maps
3:00 p.m.	Break			
3:20 p.m.	Special Session: Approximate Dynamic Programming	Biomedical Applications	Fuzzy-Neural Systems	Special Session: Biologically Inspired Computational Vision
5:20 p.m.	Break			
7:00 p.m.	Banquet			
9:00 p.m.	End of the Day			

Thursday, August 4, 2005				
Time	**Westmount**	**Outremont**	**Mont Royal/Hampstead**	**Verdun/Lachine**
8:00 a.m.	Plenary: Beyond Correlation – Closing the Loop Between Brain and Theory by Extracting Representations and Altered Feedbacks Professor Mitsuo Kawato, Director, Nara Institute of Science and Technology, ATR Computational Neuroscience Laboratories, Computational Neuroscience Laboratory, Japan			
9:00 a.m.	Break			
9:30 a.m.	Special Session: Neural Prostheses and the Neuron-Silicon Interface	Learning I	Neurodynamics	Applications I
11:30 a.m.	Break			
1:00 p.m.	Neuromorphic Hardware	Learning II	Telecommunications	Applications II
3:00 p.m.	Break			
3:20 p.m.	Plenary: Neuromorphic Engineering: Overview and Potential Professor Carver Mead, Gordon and Betty Moore Professor of Engineering and Applied Science, Emeritus Computation and Neural Systems, Division of Engineering and Applied Science, California Institute of Technology			
4:20 p.m.	End of the Day			

ATTACHMENT H

ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/4202315

Artificial neural network computation on graphic process unit

Conference Paper · January 2005

DOI: 10.1109/IJCNN.2005.1555903 · Source: IEEE Xplore

CITATIONS
68

READS
2,119

3 authors, including:



Zhongwen Luo
China University of Geosciences
22 PUBLICATIONS **401** CITATIONS
SEE PROFILE



Hongzhi Liu
Peking University
76 PUBLICATIONS **926** CITATIONS
SEE PROFILE

All content following this page was uploaded by Hongzhi Liu on 21 August 2014.

The user has requested enhancement of the downloaded file.

NVIDIA Ex1023
Page H-2

Artificial Neural Network Computation on Graphic Process Unit

Zhongwen Luo, Hongzhi Liu and Xincai Wu

Faculty of Information, China University of Geoscience(Wuhan) , Wuhan 430074,China
E-mail : luozw@cug.edu.cn,liuhz87.student@sina.com, xcwu@cug.edu.cn

Abstract

Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given.

Keywords: Graphic Process Unit; ANN; SOM; MLP

I. INTRODUCTION

Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation.

MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors.

For relative works, Kyoung-Su Oh et al.[2] implement an GPU based MLP for classify the text area in a image, and give an almost 20 time speed up over CPU. Thomas Rolfes[3] gives an artificial neural network implementation using a GPU-based BLAS level 3 style single-precisions general matrix-matrix product. Bohn[4] describes an SOM calculation method based on OpenGL hardware speed-up on SGI workstation, which inspired our work to further deploy the possibility to implement ANN calculation based on PC commodity graphic hardware.

In recent years, the graphic hardware performance is doubled every 12 months which is much faster than CPU's performance increase speed which is doubled every 18 months. And GPU vendors had make programmability on GPU, which make it possible for implement general-purpose computation.

In this paper, two kinds of ANN computation on GPU are given. In section 2, SOM computation model and implementation on graphic hardware is discussed. In section 3, MLP computation model and implementation on GPU is discussed. In section 4, the computation result and comparison are given for both CPU and GPU. In section 5, some of the design details and lessons we learned during implementation are given. In section 6, conclusion and some future works are given.

II. COMPUTATIONAL MODEL AND IMPLEMENT METHOD FOR SOM

A. The SOM Computational Model

The SOM takes a two-step computation: search for the best matching unit (BMU) and modify the map according to a distance function of the lateral connections. Usually, the Euclid distance is chosen as similarity measure method. The calculation formula takes as follow:

$$\| W_b - \xi \| < \| W_i - \xi \| \quad \text{for any i;} \qquad [1]$$

Modify the unit value regarding a distance function of the lateral connections as follow.

$$W_i^{new} = W_i^{old} - \varepsilon\Omega(r_b, r_i) * (W_i^{old} - \xi) \qquad [2]$$

As Bohn described, three OpenGL[5] extension functions are needed, they are blending, glColorMatrix and glminmax. These functions are fully support by SGI workstation, but only partially supported by PC graphic hardware. So it is difficult to fully implement SOM on OpenGL. Fortunately, current GPU provide more programmability, which makes it possible for implement the SOM calculation on GPU.

B. SOM Implementation on Programmable GPU

As discussed in the previous part, not all OpenGL functions supported by SGI workstation are supported by PC commodity graphic hardware. But recently, graphic hardware vendors provide programmability and some high-level program language[6,7]. This kind of programmability is at a lower level than OpenGL, and is more powerful than fixed OpenGL function. In this implementation, Cg [8] (C for graphic) is chosen as developing environment.

As described in previous part, the calculation contains two steps. The first is finding the best matching unit, and the second is adjusting the value according to the distance from BMU.

For finding the BMU, two steps are needed. In the first step, similarity measurement is calculated; in the second step, the minimum values which represent the BMU are found and located. The similarity computation code for GPU takes as follow:

```
Half 4 temp= tex2D(texture,coords) -intrain;
c.x = dot(temp,temp);
c.yz= coords.xy;
```

The first two sentences calculate the square of Euclid distance of two vectors, the third sentence save the unit coordinate, which will be used late for locating the best match unit.

The main difficulty in Cg computation comes from finding the minimum value and determination its location, for there is no global variable in Cg environment. We use a multipass method to calculate it. Our scheme shows as Fig 1. In each pass we find the minimum value of four units show as colored and save the result in a small size texture. After some step, the size decrease to 1, and we can get the minimum value and its location.

After get BMU, we can adjust the self-organize map according to equation [2].

III. MLP FOR REAL TIME BALL RECOGNIZING IN FIRA SOCCER ROBOT

A. Background

Currently, there are two main world soccer-robot competition, one is RoboCup[9], the other is FIRA[10]. In both case, Object are identified by their unique color. But in the near future, these color cues may be removed, so new vision algorithms based on model are needed to cope with the situation. The model based algorithm for finding ball

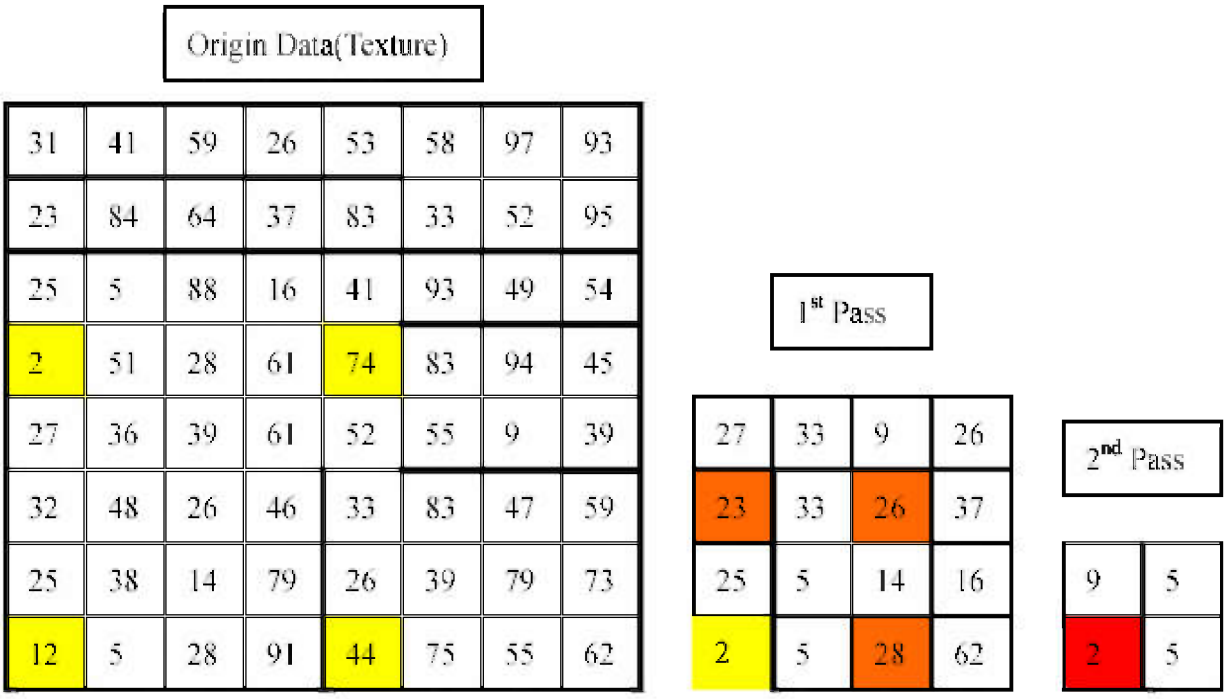


Fig. 1: Scheme for find the minimum value

and non-ball location will discussed below.

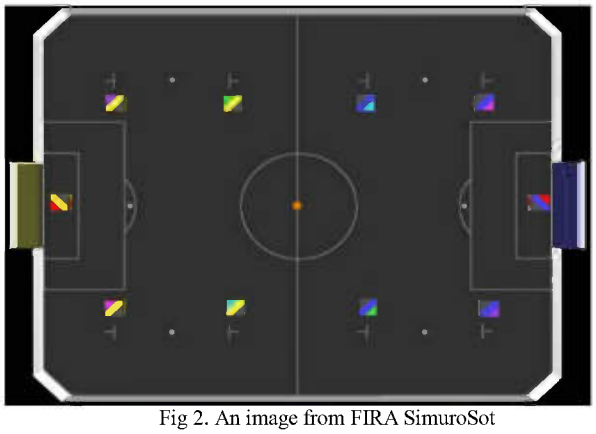


Fig 2. An image from FIRA SimuroSot

Fig2 shows an image taken from FIRA robot soccer 5vs5 simulator. Our task is to develop a model to recognize the ball, and then using this model to recognize the ball in real time.

For each location, the characters are calculated considering the color value at a small area around the location. The area radius takes as 7, which is shown in Fig 3,

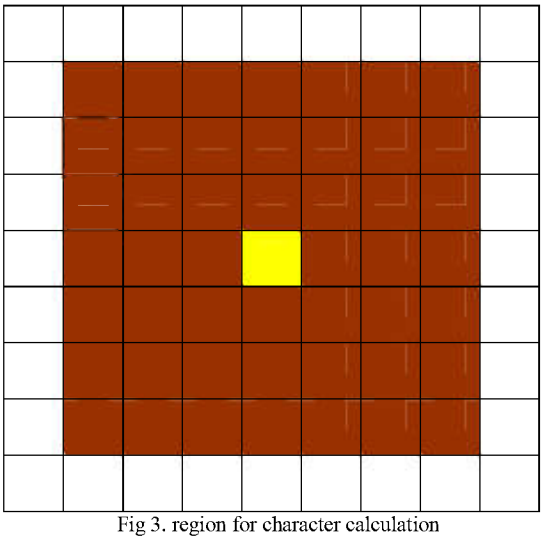


Fig 3. region for character calculation

Seven parameters are selected as characters of the ball. There are 3 average value of the colors red, green and blue around the concerning position:

$$mean_{x,y} = \sum_{i=-3}^{3}\sum_{j=-3}^{3} c_{x+i,y+j} \Big/ 49$$

3 standard deviation of the colors red green and blue:

$$delta_{x,y} = \sqrt{\sum_{i=-3}^{3}\sum_{j=-3}^{3}(c_{x+i,y+j} - mean_{x,y})^2} \Big/ 48$$

And the luminance:

$$luminance = r + g + b$$

B. Computational Model of MLP

Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network.

The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set.

The trained MLP is used to recognize and trace the ball in real time.

There are two main computation steps for MLP used as the classification machine. The first one is matrix multiplication:

$$net = w \bullet x + b \qquad [3]$$

The second one is sigmoid function calculation:

$$\sigma(net) = \frac{1}{1 + e^{-net}} \qquad [4]$$

As for the determination of ball on robot soccer, the MLP calculation was applied on each point in the play ground. And MLP was used to distinguish between ball and non-ball position.

c. MLP Implementation on Graphic Hardware

Current GPU has a limited instruction length and a limited number of temporary variables for calculation at each location. So multipass is needed for complex problem. For MLP discussed in this section, three passes are performed. In the first pass, average values of three color and luminance are calculated. In the second pass, the standard deviations of the three colors are calculated. In the third pass, classification result is got from MLP calculation on the characteristic.

In this calculation, we use a new Nvidia's GF6000 serious graphic hardware. The reason is that the ATI's GPU supports very few numbers of operations in each pass; so more passes are need for MLP calculation. Old Nvidia's GPU does not support fully float texture, which is crucial for the precision of result.

IV. RESULTS AND DISCUSSION

The environment for CPU computing is INTEL P4 2.4G. Based on this PC, ATI 9550 and Nvidia GF5700 GPU are used for SOM computing, Nvidia GF6200 GPU is used for MLP computing.

A. CPU and GPU Train Time for SOM and Discussion

Usually the train process for self organize map is time consuming when the map is large enough. So there is a need to speed up the training procedure. For our test problem, 80 data are chosen for training the SOM and average time for the computation on CPU and GPU can be reached. The result is shown in Fig 4.

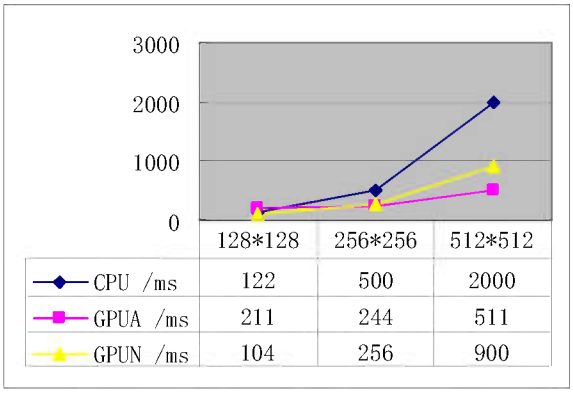


Fig. 4: SOM train computing time on GPU and CPU

The result shows that GPU based implementation is faster than CPU, especially for large self organize map. As map size increase, the computation time on GPU increase slowly than that on CPU. And different GPU had different result, for Nvidia's GPU, it takes the least time for small size SOM like 128*128, but for ATI's GPU it takes the least time for larger size SOM like 256*256. The difference may come from vendor's hardware implementation. The result shows that ATI's graphic card takes more time for the compile of program and the code is better optimized so computation time decrease with more data, but Nvidia's graphic card takes less time for compile and takes more time to computing.

B. MLP Computation Time for CPU and GPU

The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1.

The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time.

TABLE I: MLP COMPUTATION ON CPU AND GPU.

CPU /ms	11328
GPUN /ms	46

V. SOME IMPLEMENTATION DETAILS AND LESSONS

To increase the efficiency, some basic rule should obey. The main rules are given below.

First, create the GPU hardware program only once and enable it when it is needed. The reason is that when a new program is created, it will be compiled by Cg, which is time consuming.

Second, if possible, do one's best to decrease the calculation passes. For computation scheme described in section 2.2, the main bottleneck is at finding minimum procedure. The test shows that the bottleneck comes from multi-pass used in finding minimum procedure. Table 2 shows result for different scheme of finding the minimum. To decrease the pass, we make two changes, the first is to combine the value calculation with one pass of find minimum, and the second is to combine two pass of find minimum computation into one pass. Instead of calculate 4 units, 16 units are calculated. GPUA represent calculation on ATI graphic card and GPUN represent the calculation on NVIDIA graphic card. The performance increase is clear, especially for Nvidia's graphic card.

TABLE II: COMPARISON BETWEEN MORE AND FEW PASS

KFM size	128*128	256*256	512*512
GPUA more pass /ms	366	400	533
GPUA few pass /ms	211	244	511
GPUN more pass /ms	190	640	2889
GPUN few pass /ms	104	256	900

Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL's PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called "Render to Texture" to easy the use of PBuffer. We had used this class in our program.

Fourth, hardware from different venders usually has different property. So if one implementation is not work at one kind of hardware, try another implementation. For

example, in the calculation of minimum value, firstly we use the following code:

```
c=tex2D(texture,coords).x<tex2D(texture,coords+half
2(0,offset)).x? tex2D(texture,coords) :
tex2D(texture,coords+half2(0,offset));
```

Which can give the correct result in ATI card, but can not get correct result in an Nvidia's card, we think that the inner parallel schemes makes the difference. To work around it, the above sentence is changed to the following logical equivalent one:

```
half4 c=tex2D(tex,coords);
if (c.x>tex2D(tex,coords+half2(0,half_side)).x)
    c=tex2D(tex,coords+half2(0,half_side));
```

Then the result is correct for both graphic cards.

VI. CONCLUSION

In this paper, implementation for two ANN models on graphic hardware is given. Inherent parallelism of commodity graphic hardware is used to accelerate the computation of ANN. The result shows that GPU is capable for some of ANN calculation, the graphic hardware make it possible for an increasing performance/cost ratio on the area of large size ANN computation.

Compared to Bohn's initial computation on SGI workstation, our implementation has two benefits. One is our calculation is more precise, for we had use the float point computing. The other is that we only use a commodity available graphic card, which is much more easily available than SGI workstation so can be widely used.

The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have "video in" function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don't need to transfer data between CPU and GPU, which will speed the process.

We can also do other general ANN computation on GPU, because GPU provide almost all arithmetic operation, logic operation and some important mathematic function. And for the application of neural network on images, it is more naive to make such computing on a graphic hardware.

For future works, one is to make other kinds of ANN computation on GPU. The other is to further deploy the MLP on more real situation of robot soccer, which include select better parameter and use faster algorithm on GPU.

REFERENCES

[1] Teuvo Kohonen. Self-organizing maps. Springer Verlag, New York, 1997.

[2] Kyoung-Su Oh, Keechul Jung, GPU implementation of neural networks, Pattern Recognition 37, 6, 1311-1314, 2004

[3] Thomas Rolfes. Artificial Neural Networks on Programmable Graphics Hardware, in Game Programming Gems 4,pp373-378, 2004

[4] Bohn, C.A. Kohonen Feature Mapping Through Graphics Hardware. In Proceedings of 3rd Int. Conference on Computational Intelligence and Neurosciences 1998. 1998.

[5] Woo, M., Neider, J., Davis, T., Shreiner, D. OPENGL Programming Guide,Addison-Wisley,1999

[6] Femando R. and Killgard, M.J. The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics Addison-Wisley, 2003

[7] Harris, M. http://www.gpgpu.org/developer/, 2003

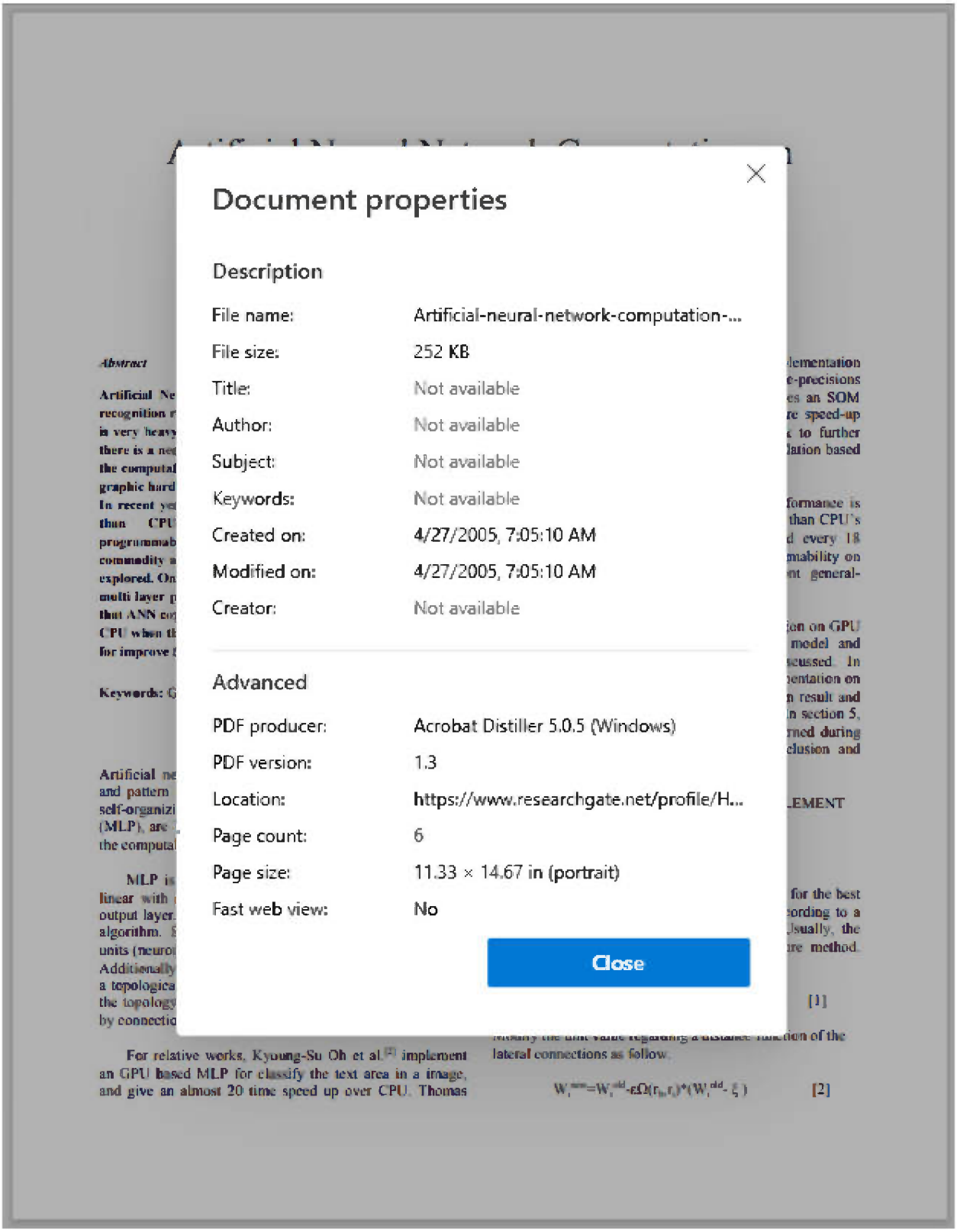
[8] Mark, W.R., Glanville, R. S., Akeley, K. and Killgard, M.J. 2003. Cg : A system for programming graphics hardware in a C-like language. ACM Trans. Graph. 22, 3, 896-907

[9] http://www.robocup.com

[10] http://www.fira.net

View publication stats

ATTACHMENT I



Document properties

Description

File name:	Artificial-neural-network-computation-...
File size:	252 KB
Title:	Not available
Author:	Not available
Subject:	Not available
Keywords:	Not available
Created on:	4/27/2005, 7:05:10 AM
Modified on:	4/27/2005, 7:05:10 AM
Creator:	Not available

Advanced

PDF producer:	Acrobat Distiller 5.0.5 (Windows)
PDF version:	1.3
Location:	https://www.researchgate.net/profile/H...
Page count:	6
Page size:	11.33 × 14.67 in (portrait)
Fast web view:	No

Close

ATTACHMENT J

Artificial Neural Network Computation on Graphic Process Unit

Zhongwen Luo, Hongzhi Liu and Xincai Wu

Faculty of Information, China University of Geoscience(Wuhan) , Wuhan 430074,China
E-mail : luozw@cug.edu.cn,liuhz87.student@sina.com, xcwu@cug.edu.cn

Abstract

Artificial Neural Network
recognition related area. I
is very heavy, in other cas
there is a need to apply a
the computation for ANN
graphic hardware is used
In recent years, graphic
than CPU. Graphic
programmability on GP
commodity available GPU
explored. One is the self-o
multi layer perceptron (M
that ANN computing on G
CPU when the neural netw
for improve the efficiency

Keywords: Graphic Proces

I. INT

Artificial neural network
and pattern recognition.
self-organizing maps (SO
(MLP), are implemented
the computation.

MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors.

For relative works, Kyoung-Su Oh et al.[2] implement an GPU based MLP for classify the text area in a image, and give an almost 20 time speed up over CPU. Thomas Rolfes[3] gives an artificial neural network implementation
3 style single-precisions
hn[4] describes an SOM
nGL hardware speed-up
ed our work to further
t ANN calculation based
e.

ardware performance is
much faster than CPU's
h is doubled every 18
ake programmability on
for implement general-

N computation on GPU
computation model and
lware is discussed. In
l and implementation on
computation result and
J and GPU. In section 5,
ssons we learned during
tion 6, conclusion and

L AND IMPLEMENT
SOM

The SOM takes a two-step computation: search for the best matching unit (BMU) and modify the map according to a distance function of the lateral connections. Usually, the Euclid distance is chosen as similarity measure method. The calculation formula takes as follow:

$$\| W_b - \xi \| < \| W_i - \xi \| \quad \text{for any i;} \qquad [1]$$

Modify the unit value regarding a distance function of the lateral connections as follow.

$$W_i^{new} = W_i^{old} - \varepsilon\Omega(r_b, r_i)*(W_i^{old} - \xi) \qquad [2]$$

Document properties

File name:	artificial-neural-network-computation-on-graphic-...
File size:	2.2 MB
Title:	Title
Author:	Author
Subject:	Subject
Keywords:	Keywords
Created:	8/29/00, 12:57:45 PM
Modified:	12/21/05, 9:09:52 AM
Application:	Creator
PDF producer:	Acrobat Distiller 3.01 for Windows
PDF version:	1.6
Page count:	5
Page size:	8.28 × 10.86 in (portrait)
Fast web view:	Yes

Close

0-7803-9048-2/05/$20.00 ©2005 IEEE

NVIDIA Ex1023
Page J-2

ATTACHMENT K

Bibliographic Record - OCLC Number: 62778290

LDR	Record Status	'c' - Corrected or revised
	Type of Record	'a' - Language material
	Bibliographic Level	'm' - Monograph/Item
	Type of Control	' ' - No specified type of control
	Encoding Level	' ' - Full level
	Descriptive Cataloging Form	'a' - AACR2
001	62778290	
005	20241110234252.9	
008	Date Entered on File	'060104'
	Type of Date/Publication Status	's' - Single date
	Date1/Beginning Date of Publication	'2005'
	Date2/Ending Date of Publication	' '
	Place of Publication, Production, or Execution Code	'nju' - New Jersey
	Illustrations	'a '
	Target Audience	' ' - Unknown or unspecified
	Form of Item	'o' - Online
	Nature of Contents Code	'b '
	Government Publication	' ' - Not a government publication
	Conference Publication	'1' - Conference publication
	Festschrift	'0' - Not a festschrift
	Index	'0' - No index
	Literary Form	'0' - Not fiction
	Biography	' ' - No biographical material
	Language Code	'eng' - English
	Modified Record	' ' - Record is not modified
	Cataloging Source Code	'd' - Other

006	Form of Material	'm' - Computer file
	Target Audience	' ' - Unknown or unspecified
	Form of Item	'o' - Online
	Type of Computer File	'd' - Document
	Government Publication	' ' - Not a government publication

007	Category of Resource	'c' - Electronic Resource
	Specific Material Designation (c)	'r' - Remote
	Color (c)	'm' - Mixed
	Dimensions (c)	'n' - Not applicable
	Sound (c)	'u' - Unknown
	Image Bit Depth (c)	'|||'
	File Formats (c)	'|' - No attempt to code
	Quality Assurance Target(s) (c)	'|' - No attempt to code
	Antecedent/Source (c)	'|' - No attempt to code
	Level of Compression (c)	'|' - No attempt to code
	Reformatting Quality (c)	'|' - No attempt to code

040 AZU **$b** eng **$e** pn **$c** AZU **$d** WAU **$d** YDXCP **$d** UWW **$d** IEEXO **$d** COO **$d** OCLCO **$d** OCLCQ **$d** OCLCF **$d** OCLCO **$d** OCL **$d** OCLCO **$d** OCLCQ **$d** IAS **$d** LIP **$d** REB **$d** O CLCQ **$d** CEF **$d** VT2 **$d** OCLCQ **$d** AU@ **$d** WYU **$d** LVT **$d** UWO **$d** OCLCQ **$d** HS0 **$d** UWK **$d** ERF **$d** UKBTH **$d** UHL **$d** SNU **$d** UKSSU **$d** EQF **$d** SXB **$d** OCLCQ **$d** O CLCO **$d** OCLCQ **$d** OCLCL **$d** OCLCO **$d** OCLCL **$d** OCLCO

019 990739018 **$a** 1058510808 **$a** 1069531342 **$a** 1096899116 **$a** 1108954677 **$a** 1110300706 **$a** 11 12524226 **$a** 1112925323 **$a** 1113038211 **$a** 1120905041 **$a** 1154916855 **$a** 1154974843 **$a** 1159 650001 **$a** 1164907703 **$a** 1165816224 **$a** 1165964972 **$a** 1166177762 **$a** 1167103058 **$a** 11739 90077 **$a** 1180929504

020 150909704X

020 9781509097043

020 **$z** 0780390482 **$q** (bd. ed.)

020 **$z** 9780780390485 **$q** (bd. ed.)

020 **$z** 0780390490 **$q** (CD ed.)

020 **$z** 9780780390492 **$q** (CD ed.)

050 4 QA76.87 **$b** .I57 2004

082 0 4 006.3/2 **$2** 22

088 05CH37662

090 **$b**

049 LDLL

111 2 International Joint Conference on Neural Networks **$d** (2005 : **$c** Montréal, Québec)

245 1 0 Proceedings of the International Joint Conference on Neural Networks (IJCNN) 2005 : **$b** July 31-Au gust 4, 2005, Hilton Montréal Bonaventure Hotel, Montréal, Québec, Canada / **$c** co-organized by the International Neural Network Society (INNS), IEEE Computational Intelligence Society.

246 3 IJCNN '05

246 1 4 2005 IEEE International Joint Conference on Neural Networks (IJCNN)

246 3 0 Joint Conference on Neural Networks

246 3 IJCNN 2005 Conference Proceedings

246 3 Neural Networks, 2005, Proceedings, 2005 IEEE International Joint Conference on

260 Piscataway, N.J. : **$b** IEEE Operations Center, **$c** ©2005.

300 1 online resource (5 volumes) : **$b** illustrations

336 text **$b** txt **$2** rdacontent

337 computer **$b** c **$2** rdamedia

338 online resource **$b** cr **$2** rdacarrier

500 "Bound edition IEEE Catalog No. 05CH37662; CD edition IEEE Catalog No. 05CH37662C"--Title pa ge verso

504 Includes bibliographical reference.

588 0 Print version record.

546 English.

650 0 Neural networks (Computer science) **$v** Congresses.

650 6 Réseaux neuronaux (Informatique) **$v** Congrès.

650 7 Neural networks (Computer science) **$2** fast

655 7 Conference papers and proceedings **$2** fast

710 2 International Neural Network Society. **$1** https://id.oclc.org/worldcat/entity/E39QH7JmqMHgkbmK3 byJM8wPGR

710 2 IEEE Computational Intelligence Society. **$1** https://id.oclc.org/worldcat/entity/E39QQPVp7VD6xmqmC9xK6GmJvR

776 0 8 **$i** Print version: **$a** International Joint Conference on Neural Networks (2005 : Montréal, Québec, Canada). **$t** Proceedings of the International Joint Conference on Neural Networks (IJCNN) 2005. **$d** Piscataway, N.J. : IEEE Operations Center, ©2005 **$z** 0780390482 **$w** (DLC) 2006296939 **$w** (OCoLC)62246786

856 4 0 **$3** IEEE Xplore **$u** http://ieeexplore.ieee.org/servlet/opac?punumber=10421

856 4 0 **$3** IEEE Xplore **$u** https://ieeexplore.ieee.org/servlet/opac?punumber=10421

856 4 0 **$u** http://VH7QX3XE2P.search.serialssolutions.com/?V=1.0&L=VH7QX3XE2P&S=JCs&C=TC0000395749&T=marc&tab=BOOKS

856 4 0 **$u** https://catalogue.solent.ac.uk/openurl/44SSU_INST/44SSU_INST:VU1?u.ignore_date_coverage=true&rft.mms_id=9997003024304796 **$z** IEEE Xplore **$a** Available **$p** 5324385730004796 **$x** UKSoSU

856 4 **$u** http://pmt-eu.hosted.exlibrisgroup.com/openurl/44OPN_INST/44OPN_services_page?u.ignore_date_coverage=true&rft.mms_id=9952420239602316 **$z** IEEE/IET Electronic Library (IEL) Conference Proceedings **$p** 5374863910002316

856 4 **$u** http://whel-primo.hosted.exlibrisgroup.com/openurl/44WHELF_ABW/44WHELF_ABW_VU1?u.ignore_date_coverage=true&rft.mms_id=9912040567702418 **$c** IEEE Xplore **$c** IEEE/IET Electronic Library (IEL) **$p** 5396604910002418

856 4 **$u** https://discover.gcu.ac.uk/discovery/openurl?institution=44GLCU_INST&vid=44GLCU_INST:44GLCU_VU2&?u.ignore_date_coverage=true&rft.mms_id=991002467350003836 **$p** 5368623820003836 **$x** StGlGCU

856 4 **$u** https://openurl.ac.uk/ukfed:bat?u.ignore_date_coverage=true&rft.mms_id=991003537999202761 **$p** 5341701020002761

856 4 **$u** https://shu.primo.exlibrisgroup.com/discovery/openurl?institution=44SHU_INST&vid=44SHU_INST:44SHU_VU1&u.ignore_date_coverage=true&rft.mms_id=9965995502501 **$z** IEEE/IET Electronic Library (IEL) Conference Proceedings **$p** 5337569100002501

856 4 **$u** http://imp-primo.hosted.exlibrisgroup.com/openurl/44IMP/44IMP_services_page?u.ignore_date_coverage=true&rft.mms_id=999753930001591 **$p** 5352154400001591

856 4 **$u** http://eu.alma.exlibrisgroup.com/view/uresolver/44WHELF_SWA/openurl?u.ignore_date_coverage=true&rft.mms_id=998183550502417 **$c** IEEE Xplore **$c** IEEE/IET Electronic Library (IEL) **$p** 5397468550002417 **$x** WlSwUW

856 4 **$u** https://whel-primo.hosted.exlibrisgroup.com/openurl/44WHELF_BANG/44WHELF_BANG_services_page?u.ignore_date_coverage=true&rft.mms_id=991004839939702422 **$p** 53124191490002422 **$x** WlBaU **$x** Available

856 4 **$u** http://whel-primo.hosted.exlibrisgroup.com/openurl/44WHELF_ABW/44WHELF_ABW_services?u.ignore_date_coverage=true&rft.mms_id=9912040567702418 **$c** IEEE Xplore **$c** IEEE/IET Electronic Library (IEL) **$p** 5396604910002418 **$x** WlAbUW

856 4 **$u** https://birmingham-primo.hosted.exlibrisgroup.com/openurl/44BIR/44BIR_VU1?u.ignore_date_coverage=true&rft.mms_id=9933491570304871 **$z** IEEE/IET Electronic Library (IEL) Conference Proceedings **$p** 53130365060004871 **$x** UkBU

938 YBP Library Services **$b** YANK **$n** 2267960

029 1 AU@ **$b** 000051517464

029 1 AU@ **$b** 000061116948

029 1 AU@ **$b** 000061420177

029 1 GBVCP **$b** 894476661

ATTACHMENT L

```
Leader 02203nam a2200433Ia 4500
001    991028507579703843
005    20221212193239.0
007    cr mnu--------
008    060104s2005    nyua    sb    101 0 eng d
020    ##$a0780390482 (bd. ed.)
020    ##$a0780390490 (CD ed.)
035    ##$a(AZU)b49877380-01ua_inst
035    ##$a(OCoLC)62778290$9ExL
040    ##$aAZU$cAZU
049    ##$aAZUG
111    2#$aInternational Joint Conference on Neural Networks$d(2005 :$cMontréal, Québec)
245    10$aProceedings of the International Joint Conference on Neural Networks (IJCNN) 2005$h[electronic resource] :$bJuly 31 - August
246    1#$aIJCNN'05
246    14$a2005 IEEE International Joint Conference on Neural Networks (IJCNN)
246    30$aJoint Conference on Neural Networks
246    3#$aIJCNN 2005 Conference Proceedings
260    ##$aPiscataway, N.J.$bIEEE Operations Center,$cc2005.
300    ##$a5 v. :$bill. ;$c28 cm.
500    ##$a"Bound edition IEEE Catalog No. 05CH37662 ; CD edition IEEE Catalog No. 05CH37662C"--T.p. verso.
504    ##$aIncludes bibliographical reference and author index.
533    ##$aElectronic reproduction.$bNew York, N.Y. :$cIEEE,$dc2005.$nSystem requirements: Adobe Acrobat Reader.$nMode of access: World
599    ##$adls
650    #0$aNeural networks (Computer science)$vCongresses.$0http://id.loc.gov/authorities/subjects/sh2008108385$2LCSH$041-LIBRARY_OF_CON
710    2#$aInternational Neural Network Society.$0http://id.loc.gov/authorities/names/n88134509$0http://viaf.org/viaf/126113409$2LCNAMES
710    2#$aIEEE Computational Intelligence Society.$0http://id.loc.gov/authorities/names/no2004115777$0http://viaf.org/viaf/311240141$2L
710    2#$aIEEE Xplore (Online service)$0http://id.loc.gov/authorities/names/no00037063$0http://viaf.org/viaf/139825222$2LCNAMES$041-LIB
710    2#$aIEEE Explore (Online service)$2LCNAMES$041-LIBRARY_OF_CONGRESS-no 00037063$9N
710    2#$aIEEE/IEE Electronic Library Online$2LCNAMES$041-LIBRARY_OF_CONGRESS-no 00037063$9N
710    2#$aIEL$2LCNAMES$041-LIBRARY_OF_CONGRESS-no 00037063$9N
710    2#$aI.E.E.E. Computational Intelligence Society$2LCNAMES$041-LIBRARY_OF_CONGRESS-no2004115777$9N
710    2#$aIEEE CIS$2LCNAMES$041-LIBRARY_OF_CONGRESS-no2004115777$9N
710    2#$aI.E.E.E. C.I.S.$2LCNAMES$041-LIBRARY_OF_CONGRESS-no2004115777$9N
710    2#$aCIS$2LCNAMES$041-LIBRARY_OF_CONGRESS-no2004115777$9N
710    2#$aC.I.S.$2LCNAMES$041-LIBRARY_OF_CONGRESS-no2004115777$9N
710    2#$aComputational Intelligence Society$2LCNAMES$041-LIBRARY_OF_CONGRESS-no2004115777$9N
710    2#$aINNS$2LCNAMES$041-LIBRARY_OF_CONGRESS-n  88134509$9N
710    2#$aInternational Neural Networks Society$2LCNAMES$041-LIBRARY_OF_CONGRESS-n  88134509$9N
740    0#$aNeural Networks, 2005, Proceedings, 2005 IEEE International Joint Conference on.
856    40$uhttp://ezproxy.library.arizona.edu/login?url=http://ieeexplore.ieee.org/servlet/opac?punumber=10421$zFIND on the Web
902    ##$a151124
948    ##$aLTI 10/15/2015
994    ##$awww
994    ##$aC0$bAZU
998    ##$a.b49877380$bwww  $c-
999    ##$b0$c060104$dm$ew$f-$g0
INST   ##$a01UA_INST
INT    ##$aE
PLK    ##$aAdditional Form.$b991048696696603843$mProceedings of the International Joint Conference on Neural Networks (IJCNN) 2005 : Ju
```

ATTACHMENT M

Artificial neural network computation on graphic process unit

Publisher: IEEE Cite This PDF

Zhongwen Luo ; Hongzhi Liu ; Xincai Wu All Authors

13 Cites in Papers
1 Cites in Patent
933 Full Text Views

Abstract

Document Sections

I. INTRODUCTION

II. COMPUTATIONAL MODEL AND IMPLEMENT METHOD FOR SOM

III. MLP FOR REAL TIME BALL RECOGNIZING IN FIRA SOCCER ROBOT

IV. RESULTS AND DISCUSSION

V. SOME IMPLEMENTATION DETAILS AND LESSONS

Show Full Outline

Authors

Figures

References

Citations

Keywords

Metrics

More Like This

Abstract:
Artificial neural network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given.

Published in: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.

Date of Conference: 31 July 2005 - 04 August 2005

DOI: 10.1109/IJCNN.2005.1555903

Date Added to IEEE *Xplore*: 27 December 2005

Publisher: IEEE

Print ISBN:0-7803-9048-2

Conference Location: Montreal, QC, Canada

ISSN Information:

Sign in to Continue Reading

Authors

Figures

References

Citations

Keywords

Keywords assist with retrieval of results and provide a means to discovering other relevant content. Learn more.

IEEE Keywords

Artificial neural networks, Computer networks, Graphics, Hardware, Central Processing Unit, Pattern recognition, Parallel algorithms, Concurrent computing, Self organizing feature maps, Neural networks

Index Terms

Neural Network, Artificial Neural Network, Graphics Processing Unit, Artificial Neural Network Computation, Self-organization, Multilayer Perceptron, Artificial Neural Network Model, Self-organizing Map,

Computational Model, Graphics Card, Kind Of Calculation, Matching Unit

Metrics

IEEE Personal Account

CHANGE USERNAME/PASSWORD

Purchase Details

PAYMENT OPTIONS

VIEW PURCHASED DOCUMENTS

Profile Information

COMMUNICATIONS PREFERENCES

PROFESSION AND EDUCATION

TECHNICAL INTERESTS

Need Help?

US & CANADA: +1 800 678 4333

WORLDWIDE: +1 732 981 0060

CONTACT & SUPPORT

Follow

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | IEEE Ethics Reporting | Sitemap | IEEE Privacy Policy

A public charity, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2025 IEEE - All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies.

ATTACHMENT N

Search within results

Export Set Search Alerts **Search History**

Showing **1-2** of **2** results for **"artificial neural networks" AND "graphics processing unit"** ×

Filters Applied: 2001 - 2006 ×

Conferences (2)

Need access to IEEE *Xplore* for your organization?

CONTACT IEEE TO SUBSCRIBE →

Show

All Results

Open Access Only

Year

Range Single Year

2001 2005

Clear Apply

Author

Affiliation

Publication Title

Publisher

Conference Location

Publication Topics

Computer games with intelligence

D. Johnson; J. Wiles

10th IEEE International Conference on Fuzzy Systems. (Cat. No.01CH37297)

Year: 2001 | Conference Paper | Publisher: IEEE

Cited by: Papers (22) | Patents (1)

Abstract HTML

Artificial neural network computation on graphic process unit

Zhongwen Luo; Hongzhi Liu; Xincai Wu

Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.

Year: 2005 | Conference Paper | Publisher: IEEE

Cited by: Papers (13) | Patents (1)

Abstract HTML

IEEE Personal Account

CHANGE USERNAME/PASSWORD

Purchase Details

PAYMENT OPTIONS

VIEW PURCHASED DOCUMENTS

Profile Information

COMMUNICATIONS PREFERENCES

PROFESSION AND EDUCATION

TECHNICAL INTERESTS

Need Help?

US & CANADA: +1 800 678 4333

WORLDWIDE: +1 732 981 0060

CONTACT & SUPPORT

Follow

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | IEEE Ethics Reporting | Sitemap | IEEE Privacy Policy

A public charity, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2025 IEEE - All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies.

ATTACHMENT O

IEEE *Xplore* 2.0 User Guide

Copyright © 2005. The Institute of Electrical and Electronics Engineers, Inc. Company and/or product names used herein are trademarks of their respective owners.

Table of Contents

OVERVIEW 1

Welcome to IEEE Xplore 1
Collections in the IEEE Xplore Database 1
Record Structure of the IEEE Xplore Database 2
Subscription Packages 2
Setting Up an IEEE Member Digital Library Account (IEEE Members) 3
Browse and Search Functions 4
Content Updates 4
Alerts 5
Top 100 Documents 5

GETTING STARTED 6

IEEE Xplore Home Page 6
IEEE Xplore Home Page for IEEE Members 7
IEEE Xplore Home Page for Institutions 8
Do I Need To Log In? 9
Logging In 11
Logging Out 12
What Can I Access? 12
Abstracts and AbstractPlus 12
Accessing the IEEE Member Digital Library (IEEE Members) 13
Accessing the IEEE Enterprise File Cabinet 13
Printing IEEE Xplore Pages 14

BROWSING IEEE XPLORE 15

Browsing IEEE Xplore General Procedure: 15
Browsing Journals & Magazines 15
Journals & Magazines Publication Page 16
Journals & Magazines Table of Contents 17
Table of Contents of Forthcoming Articles 18
Browsing Conference Proceedings 19
Conference Proceedings Home Page 19
Conference Proceedings Publication Page 20
Conference Proceedings Table of Contents 21
Browsing Standards 21
Standards Publication Page 22
Locating a Document in the IEEE Member Digital Library 22
Locating a Document in the IEEE Enterprise File Cabinet 23
Browsing by Keyword 24
Browsing Alphabetically 24

SEARCHING IEEE XPLORE 25

Searching IEEE Xplore General Procedure: 25
Using Basic Search 26
Searching within a Publication 27
Using Advanced Search General Procedure 27
Structured Advanced Search (Option 1) 28
Limiting a Search to Specific Collections 29

Limiting a Search to Specific Years 30
Controlling the Search Results Format 30
Free-Form Advanced Search (Option 2) 31
Summary of Field Codes 32
Summary of Search Operators 33
Using Author Search 34
CrossRef Search Pilot 36
Reusing Previous Searches 36
Tips for Effective Use of Search 37
Working with Search Results 43
Viewing Selected Search Results 44
Adjusting the Search Results Format 45

SEARCH EXAMPLES 46

Structured Advanced Search Examples (Option 1) 46
Free-Form Advanced Search Examples (Option 2) 47
Complex Free-Form Advanced Search Examples (Option 2) ... 48
Author Search Examples 49

WORKING WITH ABSTRACT AND ABSTRACTPLUS RECORDS 51

Abstract and AbstractPlus Pages 51
Downloading Citations 53
Arranging for Rights and Permissions 53

WORKING WITH DOCUMENTS 54

Viewing a PDF Document 54
Viewing HTML Full-Text 54
Printing Documents 55
Filing Documents in the IEEE Member Digital Library (IEEE Members) 55
Filing Documents in the IEEE Enterprise File Cabinet 56
Purchasing a Document 56

IEEE XPLORE MESSAGES 57

Login Messages and Denials 57
IEEE Member Digital Library Limit Exceeded 60
IEEE Enterprise Limit Exceeded 61

GLOSSARY 62

INDEX 69

Overview

Welcome to IEEE *Xplore*

IEEE *Xplore* is a powerful resource for accessing publications from the Institute of Electrical and Electronics Engineers (IEEE) and the Institution of Electrical Engineers (IEE). The IEEE *Xplore* database contains more than a million articles in over 12,000 individual publications. IEEE adds about 25,000 new pages to the database per month, which builds upon a back file of content published since 1988 plus select content back to 1952.

IEEE *Xplore* provides convenient, real-time access to this continually updated database. You can search the database by specifying one or more authors, index terms, and other criteria. Once located, you can view and print individual articles and papers, search results lists, tables of contents, bibliographic records, and full-page images with no limit on the number of prints.

Through the CrossRef Search feature, IEEE *Xplore* also provides access to documents published by other leading publishers.

Collections in the IEEE *Xplore* Database

The IEEE *Xplore* database contains the following IEEE and IEE publication collections:

- **Journals, transactions, and magazines** published by the IEEE, dating from January 1988, with select content back to 1952.
- **Conference proceedings** published by the IEEE, dating from January 1988, with select content back to 1953.
- **IEEE Standards**, including superseded standards but not draft standards, dating from January 1988.
- **IEE** journals, letters, magazines, and conference proceeding from 1988.

All visitors can browse tables of contents and Abstracts in the IEEE *Xplore* database. Through subscription, IEEE Members and institutions have access to more detailed Abstracts and full-text content in the IEEE *Xplore* database. Purchase of articles is available to all visitors.

To see the publications to which you have access, click **Access Information** in the top toolbar.

In search results, the Subscriber icon  indicates articles where you have access to full text.

NVIDIA Ex1023
Page O-6

Record Structure of the IEEE *Xplore* Database

The IEEE *Xplore* database merges a subset of the INSPEC bibliographic database of Abstracts, compiled from technical journals published worldwide, with full-page images of all publications and HTML versions of some publications. To locate articles, you can use IEEE *Xplore* to search complete Abstract records using the following functions:

- **Basic and Advanced search options** to search through Abstracts and article text.
- **Author search** by last name and initials.
- **Index terms** that you can click to search the database for documents containing the term.
- **References** link to view a list of the article's referenced documents with links to view the Abstract, full-text PDF, Buy via Ask*IEEE, or CrossRef.
- "**Citing Documents**" link to view documents that cite the current document.
- **Author name links** to search for other documents written by that author.
- **CrossRef Search** to search through publications by other leading publishers participating in the CrossRef system.

The IEEE *Xplore* database contains complete original page images, including all charts, graphs, diagrams, photographs, and illustrative material, from an integrated-circuit schematic to a topographic map to a photograph of a new crystalline structure. All articles, papers, and standards are in PDF format so you see them as published.

In addition, the IEEE *Xplore* database contains HTML-formatted versions of *Proceedings of the IEEE* and *IEEE Spectrum* starting from 2002.

Subscription Packages

You must subscribe to a publication in order to view the full text using IEEE *Xplore*. Click **Access Information** in the top toolbar to see which publications are in your subscription.

IEEE members have an automatic subscription to full text of *IEEE Spectrum* online. Members belonging to one or more IEEE societies have access to their societies' subscriptions.

Institutions may purchase the most comprehensive full-text subscription available in the IEEE *Xplore* database (known as IEL) or to a variety of subset collections. Available subscription packages include access to the Abstracts and full-text published since 1988 and select content published since 1952.

- **IEEE/IEE Electronic Library (IEL)**: all IEEE and IEE journals, transactions, and magazines, all IEEE and IEE conference proceedings and colloquia, and all active IEEE standards
- **IEEE All-Society Periodicals Package (ASPP)**: IEEE society-sponsored journals, transactions, and magazines
- **IEEE Conference Proceedings Order Plan (POP)**: select IEEE conference proceedings

- IEEE **Conference Proceedings Order Plan All (POP ALL)**: all IEEE conference proceedings
- IEEE **Enterprise**: pre-paid number of downloads from all IEEE journals, magazines, and conference proceedings
- **The Proceedings of the IEEE**: all issues of *Proceedings of the IEEE*
- IEEE **Spectrum Magazine**: all issues of *IEEE Spectrum*

Setting Up an IEEE Member Digital Library Account (IEEE Members)

The IEEE Member Digital Library brings IEEE Members exclusive, instant online access to full text of all IEEE journals, magazines, transactions, letters, and conference papers. IEEE Member Digital Library subscribers can file up to 25 articles each month in a filing cabinet.

To set up your IEEE Member Digital Library account:

1. You need to be an IEEE member.
2. You need to have a Web Account. Your IEEE Web Account is the key that unlocks both your online IEEE personal subscriptions and your IEEE Member Digital Library File Cabinet.
    - To become a member, go to www.ieee.org and click Membership.
    - When you create or renew your membership, select the IEEE Member Digital Library service.
    - To establish your Web Account, go to www.ieee.org, click Web Accounts, and create a Web Account.
    - Remember to keep your IEEE membership current.

As an IEEE Member Digital Library subscriber, you can view the full text of articles from publications that are not in your personal subscriptions. While logged in as an IEEE member, simply file the article in your IEEE Member Digital Library File Cabinet for convenient full-text viewing.

Browse and Search Functions

The browse functions are available to all visitors and are located under BROWSE in the navigation menu bar. After selecting a collection — Journals & Magazines, Conference Proceedings, or Standards — you can browse to a title of interest by using either the Browse by keyword or the Browse alphabetically option.

The Home page Basic Search function (search brief Abstracts) is available to all users, including users who do not subscribe to IEEE *Xplore* (guests). All other search functions are available only to IEEE members and subscribing institutions. IEEE *Xplore* provides four ways to search:

- **Basic Search** (Home page) lets you search for a single word or phrase throughout the entire database.
- **Advanced Search** provides more controlled searching. You can narrow the search to specific collections and publication years, and you can specify the format of search results. You can also narrow the search to specific fields of the database, such as Document Title, Publication Title, or Abstract. Option 1 structures the process of creating a search query, while Option 2 provides a free-form box for entering the search. Lists of field codes and search operators can easily be displayed to provide assistance.
- **Author Search** enables you to enter a complete or partial name and searches the author field in each citation record for your entry.
- **CrossRef Search** enables you to search publications by other leading publishers, employing the Google search engine.

Content Updates

The Latest Content Update link displays a page of links to the added or updated journals and magazines, conference proceedings, and standards for the past four weeks. On that page, you can also view updates for the current and the past year.

To view recently added content:

1. Click the link for the week with additions and updates you want to view.
2. Click the link to a publication to view its table of contents.

To view content added this year or last year:

1. On the Content Updates page, click the Content Updates Archive link.
2. On the Content Updates Archive page, from the pull-down menu select the week for which you want to view posted updates.
3. Select a publication type.
4. Click **Submit**.

Alerts

You can sign up to receive regular email notification of recently posted journals and magazines, including a direct link to the latest issue's table of contents in IEEE *Xplore*.

To begin receiving email alerts:

1. Click the Visit Alerts link on the Home page or Alerts in the top toolbar.
2. On the sign-up page, enter your name and email address.
3. Click **Submit**.
4. On the selection page, click to select each publication for which you want to receive update alerts.
5. Click **Submit Selection**.

To change your selections:

6. Repeat steps 1 through 3.
7. Click to select the publications to add to your alerts.
8. Click to unselect the publications to remove from your alerts.
9. Click **Submit Selection**.

To unsubscribe from email alerts:

10. Repeat steps 1 through 3.
11. Click **Unsubscribe**.

Top 100 Documents

You can view the 100 most frequently accessed documents for the month. To view the top 100 documents, click the View Top 100 link on the IEEE *Xplore* Home page. A list of the documents is presented in Citation format, which you can change to Citation & Abstract. Links are available for viewing Abstracts and full text.

Getting Started

IEEE *Xplore* Home Page

The Home page is your starting place for accessing documents in IEEE *Xplore*. Everyone visiting IEEE *Xplore* can:

Task	Options on Home Page
Browse through publications, where you can view tables of contents and Abstracts	
Perform a quick search of brief Abstracts	
View the latest updates to IEEE *Xplore*	Latest Content Update
Sign up for email alerts to find out when new issues are available	Visit Alerts
View the 100 most frequently accessed documents for the month	View Top 100

From anywhere in IEEE *Xplore*, you can return to the Home page by clicking  in the top toolbar.

Click  to see which publications are in your subscription.

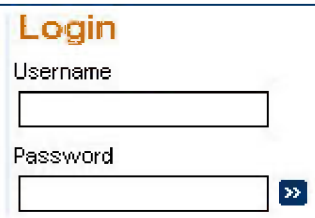
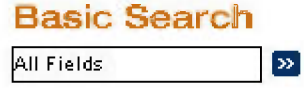
If you are an IEEE member, more options are available to you (see "IEEE *Xplore* Home Page for IEEE Members").

If you belong to an organization that subscribes to an IEEE online package, more options are available to you (see "IEEE *Xplore* Home Page for Institutions").

NVIDIA Ex1023
Page O-11

IEEE *Xplore* Home Page for IEEE Members

The Home page is your starting place for accessing documents in IEEE *Xplore*. As an IEEE member, you can:

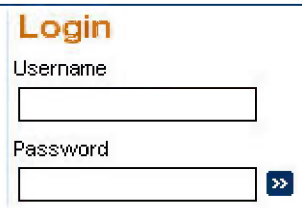
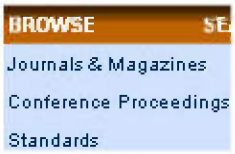
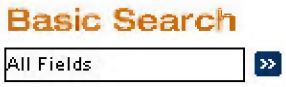
Task	Options on Home Page
Log in to access your personal subscriptions	
Perform advanced searching, author searching, and CrossRef searching Search session history	 or » Advanced Search » Author Search » CrossRef Search (Home page)
Access and file documents in your personal IEEE Member Digital Library	
Browse through publications	 or Browse » Journals & Magazines » Conference Proceedings » Standards
Perform a quick search in all fields	
View the latest updates to IEEE *Xplore*	Latest Content Update
Sign up for email alerts to find out when new issues are available	Visit Alerts
View the 100 most frequently accessed documents for the month	View Top 100

From anywhere in IEEE *Xplore*, you can return to the Home page by clicking  in the top toolbar.

After you log in, click  to see which publications are in your subscription.

IEEE *Xplore* Home Page for Institutions

The Home page is your starting place for accessing documents in IEEE *Xplore*. Through your institution's subscription, you can:

Task	Options on Home Page
Log in to access your institution's subscription benefits (if required)	
Perform advanced searching, author searching, and CrossRef searching Search session history	 or » Advanced Search » Author Search » CrossRef Search
Access and file documents in your organization's IEEE Enterprise File Cabinet	
Browse through publications	 or Browse » Journals & Magazines » Conference Proceedings » Standards
Perform a quick search in all fields	
View the latest updates to IEEE *Xplore*	Latest Content Update
Sign up for email alerts to find out when new issues are available	Visit Alerts
View the 100 most frequently accessed documents for the month	View Top 100

From anywhere in IEEE *Xplore*, you can return to the Home page by clicking  in the top toolbar.

After you log in, click  to see which publications are in your institution's subscription.

NVIDIA Ex1023
Page O-13

Do I Need To Log In?

Anyone can access IEEE *Xplore* without logging in to browse the tables of contents and Abstract pages for the journals, conference proceedings, and standards in the collection. In addition, some users may have full-text access without logging in, depending on how their institution is accessing IEEE *Xplore* and their institution's subscription status.

Review the table below to determine whether you need to log in. Note that subscribing institutions can set up IEEE *Xplore* to verify their subscription status by IP address or a login ID (username and password). Your librarian or system administrator can tell you if you need to log in. After you log in, a welcome message with your name or your institutions name appears in the top banner.

If You Are:	You Can View:	Need IEEE Web Account?	Log In Using Which User ID?
Non-member or Guest	Tables of contents	no	none
	All Abstract records	no	none
IEEE member	All AbstractPlus records	yes	IEEE Web Account username and password
	IEEE Spectrum	yes	IEEE Web Account username and password
	Personal subscriptions	yes	IEEE Web Account username and password
IEEE member with an IEEE Member Digital Library subscription	Up to 25 nonsubscription full-text documents per month in IEEE Member Digital Library	yes	IEEE Web Account username and password
IEEE society member only	AbstractPlus records for personal subscriptions	yes	IEEE Web Account username and password
	Personal subscriptions	yes	IEEE Web Account username and password
Non-member at a subscribing institution	Institution's subscription package(s)	no	Institution username and password if institution requires user login; otherwise none (authenticated by IP address)
IEEE member at a subscribing institution	All AbstractPlus records	yes	IEEE Web Account username and password
	IEEE Spectrum	yes	IEEE Web Account username and password

NVIDIA Ex1023
Page O-14

If You Are:	You Can View:	Need IEEE Web Account?	Log In Using Which User ID?
	Personal subscriptions	yes	IEEE Web Account username and password
	Institution's subscription package(s)*	no	Institution username and password if institution requires user login; otherwise none (authenticated by IP address)
IEEE member with an IEEE Member Digital Library subscription at a subscribing institution	Up to 25 nonsubscription full-text documents per month in IEEE Member Digital Library	yes	IEEE Web Account username and password
Society member at a subscribing Institution	AbstractPlus records for personal subscriptions	yes	IEEE Web Account username and password
	IEEE Spectrum	yes	IEEE Web Account username and password
	Personal subscriptions	yes	IEEE Web Account username and password
	Institution's subscription package(s)*	no	Institution username and password if institution requires user login; otherwise none (authenticated by IP address)
IEEE member at an institution, while traveling	All AbstractPlus records	yes	IEEE Web Account username and password
	IEEE Spectrum	yes	IEEE Web Account username and password
	Personal subscriptions	yes	IEEE Web Account username and password
	Institution's subscription package(s)*	no	Institution username and password
IEEE member with an IEEE Member Digital Library subscription and at an institution, while traveling	Up to 25 nonsubscription full-text documents per month in IEEE Member Digital Library	yes	IEEE Web Account username and password

If You Are:	You Can View:	Need IEEE Web Account?	Log In Using Which User ID?
Society member at a subscribing institution, while traveling	AbstractPlus records for personal subscriptions	yes	IEEE Web Account username and password
	Personal subscriptions	yes	IEEE Web Account username and password
	Institution's subscription package(s)*	no	Institution username and password
Non-member at a subscribing institution, while traveling	Institution's subscription package(s)	no	Institution username and password

***Note**: A member at an access-authenticated subscribing institution who logs in with an IEEE Web Account username and password has access to the content allowed by membership subscriptions only. To access the subscribing institutions content, DO NOT log in as a member.

Logging In

The Login area allows you to access IEEE member or institutional subscriber benefits, such as access to search functions and full text of documents.



To log in:

1. Enter your username for your IEEE Web Account, or your institution's username if your institution is not authenticated by IP address.
2. Enter your IEEE Web Account password, or your institution's password if your institution is not authenticated by IP address.

   Note: Your username and password are case-sensitive.
3. Click ».

The system verifies your username and password and returns you to the Home page or the page you were attempting to access.

A login area appears when you are not logged in and try to access the AbstractPlus or full text of a publication, use a search function (other than Basic Search), or click Access Information. For more information, see "Do I Need To Log In?"

Note: IEEE members need to create a username and password by establishing an IEEE Web Account before they can log in. The IEEE Web Account is also necessary for IEEE members who subscribe to the IEEE Member Digital Library.

NVIDIA Ex1023
Page O-16

Logging Out

If you are accessing IEEE *Xplore* through an institution, logging out of IEEE *Xplore* frees up one of your institution's seats for another user.

To log out:

Click Log Out to log out of IEEE *Xplore*.

What Can I Access?

The Access Information toolbar link enables you to see which publications you can access in a personal subscription or an institutional subscription.

Note: IEEE members must first log in with their IEEE Web Account username and password to view details of their personal IEEE online subscriptions.

If your institution's subscription requires logging in, to view details of those subscriptions you must sign in with the institution's username and password.

To discover what you can access:

1. Click Access Information.
2. Enter your username and password.
3. Click ».

The system displays a list of publications in your personal subscription or your institution's subscription. Also, a welcome message in the top banner displays your name or your institution's name.

Note: Watch for the Subscriber icon  in search results. It indicates that you have access to full text.

Abstracts and AbstractPlus

Abstract

Everyone can view an Abstract. The standard Abstract record provides bibliographic citation and Abstract information.

AbstractPlus

Logged-in users with subscription access can view the AbstractPlus page. It provides the same information as the Abstract page, plus the following information and functions:

- Keyword searching using **index term links**
- By-author searching using **author links**
- Accessing related documents using **linked references**
- Accessing **citing documents**
- **Downloading of citations**

- Arranging for **rights and permissions** by selecting 

, which takes you to the Copyright Clearance Center's Rights link electronic permissions service page
- **CODEN designation** and **INSPEC accession number**

Your institution's subscription determines whether you have access to the AbstractPlus for the article you are viewing.

Accessing the IEEE Member Digital Library (IEEE Members)

To access the IEEE Member Digital Library, follow these steps to open your IEEE Member Digital Library File Cabinet. These steps assume you have already signed up for an IEEE Member Digital Library account as part of your IEEE membership.

1. Log in to your IEEE Web Account.

   Note: If you are an IEEE member with access to IEEE *Xplore* from an institution subscribing to IEEE *Xplore*, you need to log in to your Web Account to access the IEEE Member Digital Library.

2. Click IEEE MEMBER DIGITAL LIBRARY in the navigation menu bar. Then click **File Cabinet**. IEEE *Xplore* displays a list of the documents in your File Cabinet for the current month.

Note: If you see "Welcome IEEE member Digital Library Subscriber" when you first view your IEEE Member Digital Library File Cabinet, your File Cabinet is empty and ready for you to start adding documents.

Accessing the IEEE Enterprise File Cabinet

To access the IEEE Enterprise File Cabinet, follow these steps.

1. If you see the Login area, log in to your institution's IEEE Enterprise account.

   Note: If you are logged in as an IEEE member but also have access through a subscribing institution, you need to log in to your institution's account to access the IEEE Enterprise File Cabinet.

2. Hover on or click IEEE ENTERPRISE in the navigation menu bar. Then click **File Cabinet**. IEEE *Xplore* displays a list of all documents in your IEEE Enterprise File Cabinet.

If you see "Welcome IEEE Enterprise Subscriber" when you first view your IEEE Enterprise File Cabinet, your file cabinet is empty and ready for you to start adding documents.

Printing IEEE *Xplore* Pages

You can print any page from IEEE *Xplore* by using your browser's **Print** function.

Some IEEE *Xplore* pages have a "printer friendly" alternative format. This format removes the navigation choices from the page so that the printout is easier to read.

Any page that has 

in the upper-right corner has a printer-friendly version. Click this button to display the printer-friendly version. Then use your browser's **Print** function.

When printing is finished, click 

in the upper-right corner to return to the interactive IEEE *Xplore* page.

The following IEEE *Xplore* pages have printer-friendly versions:

- Table of Contents
- Abstract
- Search Results
- AbstractPlus

NVIDIA Ex1023
Page O-19

Browsing IEEE *Xplore*

Browsing IEEE *Xplore* General Procedure

Use the following general procedure to browse the journals and magazines, conference proceedings, and standards in the IEEE *Xplore* collections.

Selecting a Collection to Browse

1. Hover on or click BROWSE in the navigation menu bar. Then select **Journals & Magazines**, **Conference Proceedings**, or **Standards**. IEEE *Xplore* displays a list of titles starting with the letter A or beginning with a number.

Locating a Publication

1. Choose a particular title to browse. You can browse by keyword to quickly locate a specific publication, or you can browse alphabetically to look through the list of titles.
2. In the list of titles, click the ⊞ next to a title to view its title history.
3. View multiple pages of results by using the result-range links, such as 26-50.
4. Click a title to view the list of issues for that publication. The publication page is displayed.

Locating a Document (Issue or Volume)

For more details on finding a specific journal or magazine, conference proceedings volume, or standard, see the appropriate topic below:

- Browsing Journals & Magazines
- Browsing Conference Proceedings
- Browsing Standards

Browsing Journals & Magazines

The **Browse Journals & Magazines** page enables you to browse through all of the periodicals in the database.

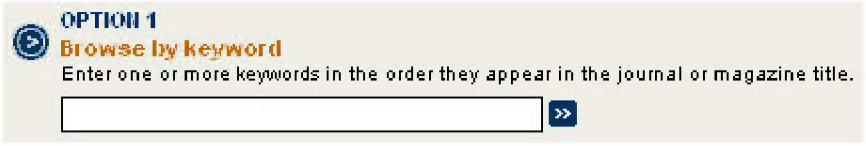
To locate a journal:

1. Hover on or click BROWSE in the navigation menu bar. Then select **Journals & Magazines**. IEEE *Xplore* displays a list of titles starting with the letter A.
2. If you are interested in only IEEE titles or IEE titles but not both, make the appropriate choice in **CHOOSE**:

   Click ».

3. Browse the list of journals by:
   - Using the Browse by keyword box.



   or

   - Using the Browse alphabetically area.



4. Use the result-range links (for example, 26-50) to view more pages of results.
5. Click the ⊞ next to a title to view its title history.
6. Click a journal or magazine title to see the list of issues. The publication page is displayed.

   Note: If the publication has forthcoming articles approved for publication, a **Forthcoming** label appears above **Year** on the publication page:



   Click the View Articles button to see a table of contents listing the forthcoming articles.

To locate an issue (on the publication page):

1. In the **Year** list, click the year of the issue you want to view.
2. In the **Issue** list, click the issue you want to view. The table of contents for the issue appears.

You can view the AbstractPlus or full text if you or your institution subscribes to the publication. You can view brief Abstracts if you do not subscribe to the publication.

Journals & Magazines Publication Page

The Journals & Magazines publication page appears when you click a specific journal or magazine title. From the publication page, you can:

All visitors:

- Go to a specific issue by selecting the year and the issue.



  From that issue's table of contents, you can browse article titles and view Abstracts.
- Look up the issue frequency, name of editor, ISDN number, and other publishing information.
- Review the title history, if the publication title has changed over time.

NVIDIA Ex1023
Page O-21

Institutional Subscribers and IEEE Members (login or access authentication required):

- Perform a simple keyword search of the publication to locate documents.



- From the results list, view the Abstract of any article. Subscribers can view the AbstractPlus.

Subscribers to the publication:

- From an issue's table of contents, view the AbstractPlus and full text of articles.
- Read more detailed publication information, such as the list of Editorial Board members, and the announcements from the society publishing the journal.



Authors:

- Submit a manuscript for publication to the journal.



Journals & Magazines Table of Contents

The table of contents page appears when you choose an issue on a journal publication page. From the table of contents page, you can:

All visitors:

- Print the table of contents.

  Use the 

button to display just the table of contents for printing.
- View the Abstract of a specific article by clicking the Abstract link.
- View multiple pages of results by using the result-range links, such as 26-50.
- For articles to which you do not have access:

  IEEE Members: You can access full text by filing the documents in your IEEE Member Digital Library File Cabinet (if you subscribe) or by purchasing the documents.

  If you belong to a **subscribing institution**: You can access full text by filing the documents in your institution's Enterprise File Cabinet (if your institution subscribes) or by purchasing the documents.

  Other guests: You can access full text by purchasing the documents.

Institutional Subscribers and IEEE Members (login or access authentication required):

- Perform a simple keyword search of the documents in the issue.



Subscribers to the publication:

- View a detailed Abstract by clicking AbstractPlus.
- View full text of articles by clicking the PDF link.
- Read more detailed publication information, such as the list of Editorial Board members, and the announcements from the society publishing the journal.



Authors:

- Submit a manuscript for publication to the journal.



Table of Contents of Forthcoming Articles

When articles have been accepted for future publication in a journal or magazine, the publication page displays this indicator:



When you click the View Articles button, a table of contents appears, listing the titles of articles accepted for future publication in the journal or magazine. From here, you can:

All visitors:

- Print the table of contents. Use the printer friendly button to display just the table of contents for printing.
- View the Abstract of a specific article by clicking Abstract.

Institutional Subscribers and IEEE Members (login or access authentication required):

- Perform a simple keyword search of the documents in the list.



Subscribers to the publication:

- View a detailed Abstract by clicking AbstractPlus.
- View full text of articles to which you have access by clicking PDF.

Authors:

- Submit a manuscript for publication to the journal.



Browsing Conference Proceedings

The **Browse Conference Proceedings** page enables you to browse through all of the conference proceedings in the database.

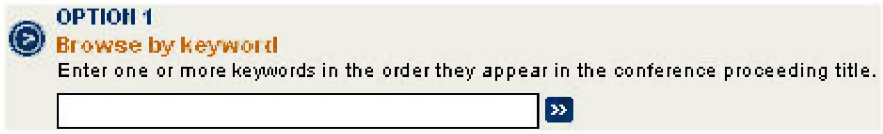
To locate a conference proceeding:

1. Hover on or click 

in the navigation menu bar. Then select **Conference Proceedings**. IEEE *Xplore* displays a list of titles starting with the letter A or beginning with a number.
2. If you are interested in only IEEE titles or IEE titles but not both, make the appropriate choice in **CHOOSE**:

   Click .
3. Browse the list of conference proceedings by:
   - Using the Browse by keyword box.

   or

   - Using the Browse alphabetically area.

4. Use the result-range links (for example, 26-50) to view more pages of results.
5. The ⊞ next to a title indicates a conference grouping. Click the ⊞ next to a title to view the titles of all conferences in the grouping.

   Note: If you click the title next to the ⊞, you can view the conference proceedings Home page.
6. Click a conference publication title to see the list of volumes. The publication page is displayed. You can view the AbstractPlus or full text if you or your institution subscribes to the publication. You can view brief Abstracts if you do not subscribe to the publication.

Conference Proceedings Home Page

When you click the ⊞ next to a conference title, you view the Home page for the grouping of conferences under that title. On the conference proceedings Home page, you can:

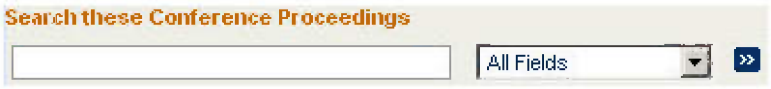
All visitors:

- Go to a specific conference proceeding by clicking its link.

NVIDIA Ex1023
Page O-24

Institutional Subscribers and IEEE Members (login or access authentication required):

- Perform a simple keyword search of the listed conference proceedings.



- From the results list, view the Abstract of any article. Subscribers can view the AbstractPlus.

Subscribers to the publication:

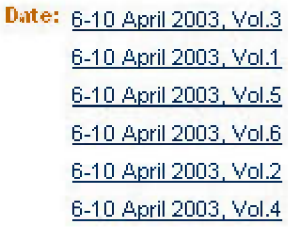
- From an issue's table of contents, view the AbstractPlus and full text of papers in the publication.

Conference Proceedings Publication Page

The **Conference Proceedings** publication page appears when you click a conference title. From the publication page, you can:

All visitors:

- Go to a specific volume by clicking its title.



From the volume's table of contents, you can browse paper titles and view Abstracts.

- Review the title history and related conference proceedings titles.

Institutional Subscribers and IEEE Members (login or access authentication required):

- Perform a simple keyword search of the conference proceeding volumes to locate papers.



Subscribers to the publication:

- From an issue's table of contents, view the AbstractPlus and full text of papers in the publication.

Conference Proceedings Table of Contents

The table of contents page appears when you choose a volume on a conference proceedings publication page. From the table of contents page, you can:

All visitors:

- Print the table of contents.

  Use the printer friendly button to display just the table of contents for printing.
- View the Abstract of a specific paper by clicking the Abstract link.
- View multiple pages of results by using the result-range links, such as 26-50.
- For articles to which you do not have access:

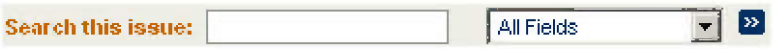
  IEEE Members: You can access full text by filing the documents in your IEEE Member Digital Library File Cabinet (if you subscribe) or by purchasing the documents.

  If you belong to a **subscribing institution**: You can access full text by filing the documents in your institution's Enterprise File Cabinet (if your institution subscribes) or by purchasing the documents.

  Other guests: You can access full text by purchasing the documents.

Institutional Subscribers and IEEE Members (login or access authentication required):

- Perform a simple keyword search of the documents in the publication.



Subscribers to the publication:

- View a detailed Abstract by clicking AbstractPlus.
- View full text of papers by clicking PDF.

Browsing Standards

The **Browse Standards** page enables you to browse through all of the standards in the database.

To locate a standard:

1. Hover on or click BROWSE in the navigation menu bar. Then select **Standards**. IEEE *Xplore* displays a list of titles organized by standard number.
2. Browse the list of standards using the Browse by keyword box.



3. Use the result-range links (for example, 26-50) to view more pages of results.

NVIDIA Ex1023
Page O-26

4. Click a standard title to view the publication page for the standard.

You can view the AbstractPlus or full text if your institution subscribes to the publication. You can view brief Abstracts if you do not subscribe to the publication.

Standards Publication Page

The **Standards** publication page appears when you click the title of a standard. From the publication page, you can:

All visitors:

- View the Abstract by clicking the Abstract link.
- For a standard to which you do not have access: You can access full text by purchasing the standard at http://shop.ieee.org/ieeestore/.

Institutional Subscribers and IEEE Members (login or access authentication required):

- View a detailed Abstract by clicking AbstractPlus.

Locating a Document in the IEEE Member Digital Library

1. Log in to your IEEE Web Account.
2. Hover on or click IEEE MEMBER DIGITAL LIBRARY in the navigation menu bar. Then select **File Cabinet**. IEEE *Xplore* displays your Member Digital Library File Cabinet, showing documents filed in the current month.
3. Select the year in which you filed the document you want to locate. Then select the month.



IEEE *Xplore* displays the documents filed during the selected month and year. The documents are listed in the order in which you filed them.

4. You can resort the documents by article title or author last name (first author) by using the following choices:



Locating a Document in the IEEE Enterprise File Cabinet

Use the following procedure to quickly locate a specific document in the IEEE Enterprise File Cabinet.

1. Hover on or click IEEE ENTERPRISE in the navigation menu bar. Then click **File Cabinet**. IEEE *Xplore* displays your IEEE Enterprise File Cabinet.
2. Enter one or more words in the **Quick Find** box.

   If you enter more than one word, Quick Find searches for the words as a phrase, in the order you entered them, not as individual words.

   Note: You cannot include Boolean operators (AND, OR, and NOT) or special characters (, ( ) @ [ ] { } < > “ ‘ / \). If you use one of these operators or characters, you receive the message “Request Cannot Be Processed.”
3. Select **Publication Title**, **Document Title**, or **Author** after **IN** to specify the type of search.
4. Click ». IEEE *Xplore* searches the File Cabinet for the terms you entered, and presents a list of matching documents.

   The Quick Find function returns plural and suffixed forms of the words you entered. For example, if you enter one word, “electron”, the results include words such as electrons, electronic, and electronically. If you enter more than one word (a phrase), only the plural and suffixed forms of the last word are included. This capability helps you locate articles when you are unsure of the exact title or name.

   The results match the words you entered regardless of most punctuation you include. For example, if you search for **solid-state**, your results may include “solid-state”, “solid state”, and “solid_state”.
5. To search within your results, repeat steps 2 through 4.
6. To view again all articles in the File Cabinet, click your browser **Back** button.

If you receive no results, click your browser **Back** button and modify your search query to include fewer words. If you are searching in the Document Title field, you can broaden your search to the Publication Title field.

You can also browse the File Cabinet if you are unsure of the exact words to use in a search query.

NVIDIA Ex1023
Page O-28

Browsing by Keyword

Use the following procedure to quickly locate a specific journal, conference proceeding volume, or standard. This procedure assumes you have clicked BROWSE and selected a publication type (**Journals & Magazines**, **Conference Proceedings**, or **Standards**).

1. In the **Browse by keyword** box:



   enter one or more keywords from the title of the publication.

   Note: If you are unsure of the spelling, you may enter a partial keyword.

2. Click ». IEEE *Xplore* filters through each publication's title to locate the words you entered, and presents a list of titles including those words.
3. If more titles containing the keyword exist than appear on the first page, click the result-range links, such as 26-50, to view more pages of titles.

The Browse by keyword function returns results containing plural and suffixed forms of the keyword(s). For example, if you enter the keyword "electron", the results include titles containing words such as electrons, electronic, and electronically. This capability helps you locate publications by topic when you are unsure of the exact title.

The results match your keyword(s) regardless of most punctuation you include. For example, if you search for **solid-state**, your results may include "solid-state", "solid state", and "solid_state".

Browsing Alphabetically

Use the following procedure to see a list of titles for journals, conference proceedings, or standards. This procedure assumes you have clicked BROWSE and selected a publication type (**Journals & Magazines**, **Conference Proceedings**, or **Standards**).

1. In the **Browse alphabetically** area:



   click a letter to see publication titles beginning with that letter.

   Note: You can click the letter matching the first word in the title or the first significant word in the title.

   The first page of titles beginning with that letter appears.

2. If more titles beginning with that letter exist than appear on the first page, click the result-range links, such as 26-50, to view more pages of titles.

Searching IEEE *Xplore*

Searching IEEE *Xplore* General Procedure

Use the following general procedure to search the IEEE *Xplore* collections.

Quick Searching

1. On the Home page, enter a keyword in the **Basic Search** box and click  .



   Basic Search is available to everyone visiting IEEE *Xplore*.

or

   You can perform a quick search on a publication page or table of contents page for a publication. Enter a keyword in the **Search This Publication**, **Search this issue**, or **Search this Conference Proceeding** box and click  . You need to be logged in to IEEE *Xplore* to perform searching within publications.

More Advanced Searching

You need to be logged in to your institution's subscription account to access advanced searching functions. IEEE members must be logged in to IEEE *Xplore* with their IEEE Web Account username and password.

1. Select **Advanced** or **Author** under 

.
2. Enter a keyword, phrase, author name, or search query in the text box(es).
3. Select additional search options (Advanced search).
4. Click 

or  . IEEE *Xplore* returns a list of results. The Subscriber icon  indicates the articles where you have access to full text.

Revising Search Results

5. You can refine your search by modifying or replacing your search query and clicking  .

Searching within Search Results

6. To search for a subset of results within your current results set, enter one or more keywords, click the checkbox next to **Check to search only within this results set**, and click  .

Selecting a Result

7. View multiple pages of results by clicking the result-range page links, such as 26-50.

NVIDIA Ex1023
Page O-30

8. Click Abstract or AbstractPlus to view an article's Abstract, or PDF to view the full text.

Note: You can view full text if you or your institution subscribes to the publication. You can view the AbstractPlus if you are an IEEE member or your institution subscribes to the publication. All visitors can view brief Abstracts.

Using Basic Search

Basic searching enables you to search the database for one or more keywords in any field. Use advanced searching if you want to narrow your search by field, collection, or date, or if you want to combine keywords in specific ways.

To perform a basic search:

1. On the IEEE *Xplore* Home page, enter one or more keywords in the **Basic Search** box:



2. Click ».

   If IEEE *Xplore* asks you to log in: Enter your username and password. See "Do I Need To Log In?" if you need more information.

   IEEE *Xplore* looks for matches in any of the following fields: document title, publication title, author, Abstract, index terms, and affiliation. It displays a list of results matching your keyword(s).

3. You can view a detailed Abstract by clicking AbstractPlus.
4. For articles in publications to which you subscribe, you can view full text by clicking PDF.

   The Subscriber icon indicates articles where you have access to full text.

5. You can view multiple pages of results by using the result-range links, such as 26-50.

For articles in publications to which you do not subscribe:

- **IEEE Members**: You can access full text by filing the documents in your IEEE Member Digital Library File Cabinet (if you subscribe) or by purchasing the documents.
- If you belong to a **subscribing institution**: You can access full text by filing the documents in your institution's IEEE Enterprise File Cabinet (if your institution subscribes) or by purchasing the documents.
- **Other guests**: You can access full text by purchasing the documents.

NVIDIA Ex1023
Page O-31

Searching within a Publication

You can use simple keyword searching to locate specific articles or papers in the publication you are browsing. You need to be logged in to IEEE *Xplore* to perform searching within a publication.

1. Hover on or click BROWSE in the navigation menu bar. Then select **Journals & Magazines**, **Conference Proceedings**, or **Standards**. IEEE *Xplore* displays a list of titles starting with the letter A.
2. Access the publication you want by browsing by keyword or browsing alphabetically.
3. Optional: Click the issue or volume containing the article(s) or paper(s) of interest.
4. To find the articles or papers of interest, enter one or more keywords in the **Search This Publication**, **Search this issue**, or **Search this Conference Proceeding** box.
5. Click ».

   If IEEE *Xplore* asks you to log in: Enter your username and password. See "Do I Need To Log In?" if you need more information.

   IEEE *Xplore* searches in the following fields for the keyword(s) you entered: document title, publication title, author, Abstract, index terms, and affiliation. It displays a list of results matching your keyword(s)

From the results list, you can view the AbstractPlus or full text of articles.

The Subscriber icon indicates the articles where you have access to full text.

Using Advanced Search General Procedure

Use advanced search if you want to narrow your search by field, collection, or date, or if you want to combine keywords in specific ways. Advanced Search is available to IEEE members and subscribers.

To perform an Advanced Search:

1. Hover on or click SEARCH in the navigation menu bar. Then select **Advanced**.

   If IEEE *Xplore* asks you to log in: Enter your username and password. See "Do I Need To Log In?" if you need more information.

   IEEE *Xplore* displays the **Advanced Search** page.
2. Enter your search query using either the structured search (Option 1) or the free-form search (Option 2). When you click Run Search, IEEE *Xplore* displays the results of your search.
3. You can view a detailed Abstract by clicking AbstractPlus.
4. For articles in publications to which you subscribe, you can view full text by clicking PDF.

   The Subscriber icon indicates articles where you have access to full text.

NVIDIA Ex1023
Page O-32

5. You can view multiple pages of results by using the result-range links, such as 26-50.

For articles in publications to which you do not subscribe:

- **IEEE Members**: You can access full text by filing the documents in your IEEE Member Digital Library File Cabinet (if you subscribe) or by purchasing the documents.
- If you belong to a **subscribing institution**: You can access full text by filing the documents in your institution's IEEE Enterprise File Cabinet (if your institution subscribes) or by purchasing the documents.
- **Other guests**: You can access full text by purchasing the documents.

Structured Advanced Search (Option 1)

The **Advanced Search** page provides a structured way for you to perform a more complex search. The following procedure assumes you have clicked SEARCH in the navigation menu bar and selected **Advanced**.

1. In the Option 1 area, click in the first text box and enter a keyword. The keyword can be a subject term, last name, date, or phrase. If you are unsure of the spelling, you may enter a partial word or name. You may also use wildcards.

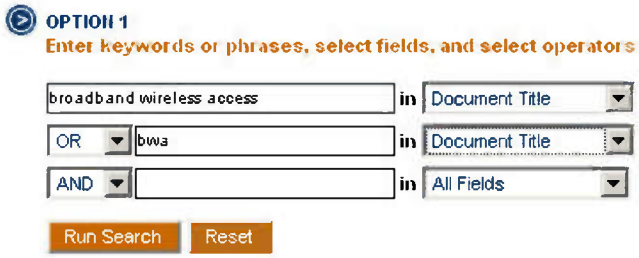
   IEEE *Xplore* searches for items regardless of capitalization. See "Entering Search Keywords" for additional rules governing entry of search keywords.

2. Limit your search by selecting a choice in the field list next to the text box.



   IEEE *Xplore* looks for the search string in all fields unless you select a choice from this list.

3. To include additional words or phrases in the search query, enter search text and select field codes for the other search query boxes as necessary. Define the relationship between the query boxes by selecting search operators from the pull-down menus between the text boxes. For example:



The search operators available in the pull-down menus are:

AND: Searches for records that meet the two conditions it joins.

OR: Searches for records that meet either or both of the two conditions it joins.

NOT: Searches for records that do *not* meet the specified condition.

4. Limit your search results by adjusting the search options. You can:
   - Select publication types
   - Select years to search
   - Organize the format of search results
5. Click Run Search. IEEE *Xplore* looks through the citation records and presents a list of results matching your query. The Subscriber icon indicates articles where you have access to full text.
6. If more titles containing the keyword exist than appear on the first page, click the result-range links, such as 26-50, to view more pages of titles.

Limiting a Search to Specific Collections

In advanced search, you can limit your search to a subset of the IEEE *Xplore* collections by selecting the appropriate choices under **Publications**.



IEEE *Xplore* searches in all collections until you specify a subset. It keeps your choice active until you change it.

Institutions that subscribe to certain subset packages (see “Subscription Packages”) can limit search results to only those documents to which they have full access by selecting **Search subscribed content only**.

NVIDIA Ex1023
Page O-34

Limiting a Search to Specific Years

In advanced search, you can limit your search to specific years of publication by selecting the appropriate choices under **Select date range**.

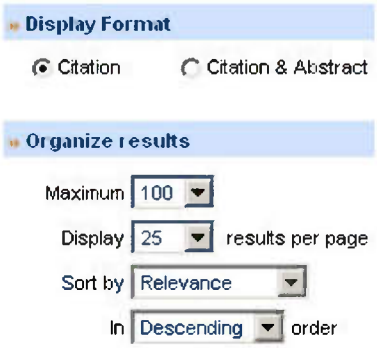


IEEE *Xplore* searches in all publication years until you specify a subset. It keeps your choice active until you change it.

You can limit search results to the most recently added content by selecting **Search latest content update**.

Controlling the Search Results Format

In advanced search, IEEE *Xplore* normally displays your results using the following format specifications:



You can change these specifications to view your results in a different format. IEEE *Xplore* keeps your specifications active until you change them.

Your options are:

- **Citation** format consists of document title, author, publication title, and bibliographic information. **Citation & Abstract** format also includes the first 30 to 40 words of the Abstract.
- **Maximum** sets the total number of results found. Choices are 100, 250, and 500.
- **Display...results per page** sets the number of results displayed per page. Choices are 25, 50, and 100.
- **Sort by** sets the criteria for ordering the results. Choices are **Relevance** (how well the result matches the search query as determined by IEEE *Xplore*), **Year** (numerical order by publication year), and **Document Title** (alphabetical, ignoring leading articles such as “a”, “the”, and “an”).
- **In...order** sets the sort order. Choices are **Descending** (high to low) and **Ascending** (low to high).

NVIDIA Ex1023
Page O-35

Free-Form Advanced Search (Option 2)

The **Advanced Search** page enables you to enter a free-form search query. In free-form advanced search, you can specify and perform more complex searches than you can using structured advanced search. You can join more than four search terms, use other operators in addition to **And** and **Or**, and exert more control over the order in which to evaluate expressions.

The following procedure assumes you have clicked SEARCH in the navigation menu bar and selected **Advanced**.

1. In the Option 2 area, click in the text box and enter a keyword. The keyword can be a subject term, last name, date, or phrase. If you are unsure of the spelling, you may enter a partial word or name. You may also use wildcards.

   IEEE *Xplore* searches for items regardless of capitalization. See "Entering Search Keywords" for additional rules governing entry of search keywords.

2. Limit your search by specifying which field to search in, using the **<in> operator** and a field code as shown in the following syntax:

   keyword <in> fieldcode

   where **keyword** is your search keyword or phrase and **fieldcode** is a field abbreviation listed in the summary of field codes.

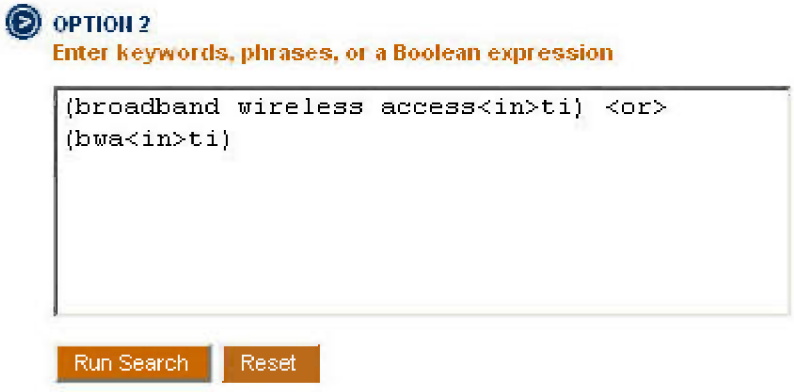
   IEEE *Xplore* looks for the keyword in all fields unless you limit the search to specific fields.

3. Combine multiple search expressions as needed. Use operators to define the relationship between search expressions, using the following syntax:

   search_expression <operator> search_expression

   where **search_expression** is a search string and an optional field specification, and **<operator>** is an operator listed in the summary of search operators.

   For example, the following search expression looks for either the phrase **broadband wireless access** or the abbreviation **bwa** in document titles:



   See "Search Examples" for more information.

4. Limit your search results by adjusting the search options. You can:
   - Select publication types
   - Select years to search
   - Organize the format of search results
5. Click Run Search. IEEE *Xplore* looks through the citation records and presents a list of results matching your query. The Subscriber icon indicates articles where you have access to full text.
6. If more titles containing the keyword exist than appear on the first page, click the result-range links, such as 26-50, to view more pages of titles.

Summary of Field Codes

Field codes are elements that identify specific parts of an Abstract record. By limiting a search to a specific field, you can reduce the time it takes to process the search and produce more targeted results. Here are the field codes you can use in search expressions.

Field Name	Field Code
Abstract	**ab**
Affiliation	**cs**
All fields	**metadata**
Author(s)	**au**
Catalog number	**ca**
CODEN	**cn**
Conference date	**cy**
Document title	**ti**
Editor(s)	**au**
Full text	**pdfdata**
Index term	**de**
ISBN	**in**
ISSN	**in**
Issue number	**is**
Issue part number	**pt**
Meeting date	**cy**
Part number	**pt**
Publication name	**jn**
Publication year	**pyr**
Subject term	**de**
Title	**ti**
Volume	**vo**

Summary of Search Operators

Search operators are elements that express relationships between search terms or search expressions, or that otherwise modify a query. In structured advanced searching, you can use the logical operators **And**, **Or**, and **Not**. In free-form advanced searching, you can use any of the following operators.

Operator	Syntax	Find Results that...
<and>	***x* <and> *y***	Match both expressions ***x*** and ***y***
<or>	***x* <or> *y***	Match either expression ***x*** or ***y*** or both
<not>	**<not> *x***	Do *not* match expression ***x***
	x* <not> *y	Match expression ***x*** but *not* ***y***
<in>	***x* <in> *y***	Match expression ***x*** in field ***y***.
<paragraph>	***x* <paragraph> *y***	Match expressions ***x*** and ***y*** in same paragraph
<sentence>	***x* <sentence> *y***	Match expressions ***x*** and ***y*** in same sentence.
<phrase>	***x* <phrase> *y***	Match expressions ***x*** and ***y*** in same phrase.
<near>	***x* <near/*y*> *z***	Match expression ***x*** within ***y*** words of expression ***z***.
<many>	**<many> *x***	Match expression ***x***, with results listed in score order based on density of ***x***.
<order>	**<order> *x***	Match order of terms in expression ***x***.
<thesaurus>	**<thesaurus> *x***	Match synonyms of ***x*** as well as ***x*** itself.
<wildcard>	**<wildcard> `*x*`**	Match variations or patterns. `**[*xy*]**` Match any letter ***x*** or ***y***. `**[^*xy*]**` Match any letter *but* ***x*** or ***y***. `**{*ab*,*xy*}**` Match string ***ab*** or ***xy***.
<stem>	**<stem> *x***	Match stem variations of ***x***; don't score results.
<word>	**<word> *x***	Match ***x*** exactly.

NVIDIA Ex1023
Page O-38

Using Author Search

Use author search if you want to find documents by specific authors. Author search is available to IEEE members and subscribers.

To perform an author search:

1. Hover on or click SEARCH in the navigation menu bar. Then select **Author**.

   If IEEE *Xplore* asks you to log in: Enter your username and password. See "Do I Need To Log In?" if you need more information.

   IEEE *Xplore* displays the **Author Search** page.

2. Enter a last name in the **Quick Find an Author** box and click ».

   Note: Use one of the following formats to enter the name. For example, if you want to search for Mary R. Jones:

   - Jones (just the last name)
   - Jones M (the last name plus one initial, with or without a following period)
   - Jones M. R. (the last name plus two initials, each followed by a period and separated with a space)

   Make sure that you do NOT use a comma. It is a special query language character.

   Note: If you enter a partial last name, IEEE *Xplore* returns a list of last names starting with the letters you entered.

3. Use the Next 50 and Previous 50 buttons to scroll through the results, 50 at a time.

4. Click a name to view a list of articles written by that author. The Subscriber icon indicates articles where you have access to full text.

You can also browse author names alphabetically.

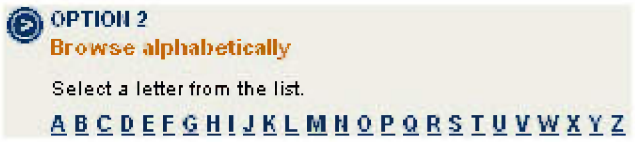
NVIDIA Ex1023
Page O-39

Browsing Authors Alphabetically

You can look through alphabetical lists of author names in IEEE *Xplore*.

This procedure assumes you have clicked SEARCH in the navigation menu bar and selected **Author**.

1. On the Author Search page, click a letter to view author names beginning with that letter.



   IEEE *Xplore* returns a list of the first 50 authors whose last names start with that letter.

2. Use the Next 50 and Previous 50 buttons to scroll through the results, 50 at a time.
3. Click a name to view a list of articles written by that author. The Subscriber icon indicates articles where you have access to full text.

Formatting Author Names

To find an article written by a particular author, you need to follow the IEEE *Xplore* rules for specifying author names:

- Enter the last name first.
- If you know the first name or initial, enter a space, followed by the *initial*. The period (**.**) after the initial is optional. Do not enter a comma (**,**) after the last name and do not enter more than the initial letter of the first name.
- If you know the middle name or initial, enter a space after the first initial, followed by the initial of the middle name, followed by a period (**.**).
- Asian names appear as provided in the article or paper. If an Asian name includes a non-Asian first name, that part of the name is abbreviated to a first initial.

Examples:

johnson j. w.

Matches ***Johnson, J.W.*** Does not match ***Johnson, J.*** (both initials must match) or ***Johnson, J.R.***

johnson s.

Matches ***Johnson, S.*** and ***Johnson, S.L.*** Does not match ***Johnson, J.*** or ***Johnson, J.S.***

NVIDIA Ex1023
Page O-40

CrossRef Search Pilot

CrossRef Search Pilot allows you to search full text of participating publishers' scholarly documents indexed by Google™. Use CrossRef Search Pilot, powered by the Google search engine, to locate content from participating publishers' sites, including IEEE.

Full-text access to articles is determined by each publisher and your institution's subscriptions. CrossRef Search is available to IEEE members and institutional subscribers.

To perform a CrossRef Search:

1. Hover on or click SEARCH in the navigation menu bar. Then select **CrossRef**.

   If IEEE *Xplore* asks you to log in: Enter your username and password. See "Do I Need To Log In?" if you need more information.

   IEEE *Xplore* displays the **CrossRef Search Pilot** page.
2. Enter your search query into the search box. When you click Search, the first page of Google search results appears in a new browser window, grouped by publisher.
3. You can view an article by clicking its title. You may be asked to log in to the publisher's website.
4. You can view multiple pages of results by using the page links.

   Result Page: 1 2 3 4 5 6 7 8 9 10 Next

For the most comprehensive search of IEEE content, use the IEEE *Xplore* Advanced Search.

Reusing Previous Searches

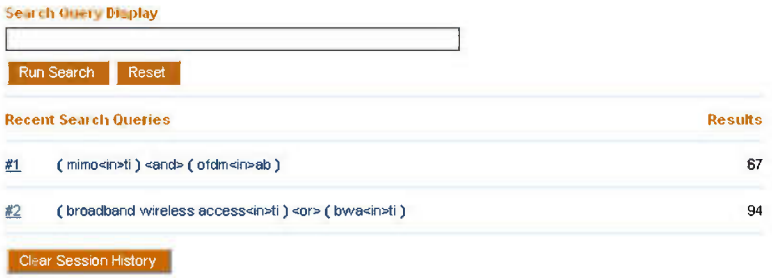
Your session history lists the search queries you have used since you last logged in to IEEE *Xplore*. You can reuse or modify these search queries. Session history is erased when you log off or your session times out.

To reuse a search query from earlier in your session:

1. Hover on or click SEARCH in the navigation menu bar. Then select **Session History**. (On the **Search Results** page, you can also click View Session History.)

   Note: If you are asked to log in at this point, your session history will be empty.

   IEEE *Xplore* displays the **Search Session History** page, listing search queries you entered earlier in the session. For example:



2. Click the number of the search query you want to reuse. The query appears in the **Search Query Display** box.
3. (Optional) Modify the query as needed in the **Search Query Display** box.
4. Click Run Search. IEEE *Xplore* runs the search and displays the results page.

   The Subscriber icon  indicates articles where you have access to full text.

Tips for Effective Use of Search

Ideally the first query you enter for your search yields the results you want. Typically, you need to reformulate your searches to get the best results. Here are guidelines and strategies for improving search results:

- Entering Search Keywords
- Revising a Search
- Searching for an Exact Word or Phrase
- Searching with Wildcard Characters
- Specifying Multiple Criteria
- Nesting Search Operations
- Searching with Incomplete Information
- Narrowing a Search
- Broadening a Search

Entering Search Keywords

The following conventions govern how you enter search keywords:

- **Capitalization**: IEEE *Xplore* searches for items regardless of capitalization.
- **Keyword phrases**: IEEE *Xplore* interprets multiple keywords as a phrase.
- **Stem variations**: IEEE *Xplore* looks for plural and suffixed forms of any keywords entered. For example, if you enter **cable**, IEEE *Xplore* also looks for words like **cables** and **cabling**. See “Searching for an Exact Word or Phrase” if you do not want IEEE *Xplore* to search for variations.
- **Punctuation**: IEEE *Xplore* ignores most punctuation. For example, if you search for **solid-state**, IEEE *Xplore* looks for “solid-state”, “solid state”, and “solid_state”, as well as any other occurrences of “solid state” with punctuation between the two words.
- **Special characters**: In IEEE *Xplore*, the following characters have special meaning in free-form search queries, so do not use them in your search keywords:

  , ( ) @ [ ] { } < >

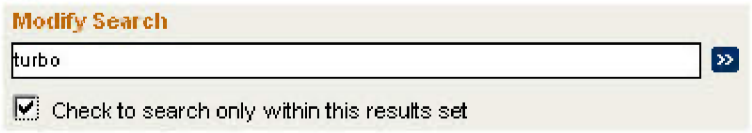
NVIDIA Ex1023
Page O-42

Revising a Search

If the search results you receive from using Advanced Search consist of too many or too few documents, you can revise the search query to improve the results. Below are ways to revise the search query. See Narrowing a Search and Broadening a Search for strategies on improving search queries.

To search within the current search results:

- Enter additional keywords into the **Modify Search** box, click the checkbox for **Check to search only within this results set**, and click ». For example:



To modify the existing search query:

- Return to your original search page using your browser's **Back** button and edit the search query.

To start a new search:

- Enter a new query in the text box on the search results page and click ».

  or

- Click New Search on the search results page to display a blank Advanced Search page, where you can enter a new query.

Searching for an Exact Word or Phrase

To search for an exact match, enter the search word or phrase in quotation marks. For example, if you enter the following search word:

"cable"

IEEE *Xplore* looks for matches in any record containing the exact word *cable* in any field. It does not look for *cables, cabled,* or *cabling*.

Notes:

- If the text contains punctuation, replace each punctuation mark with a space.
- You cannot use the wildcard characters **?** and ***** within the quotation marks.
- IEEE *Xplore* matches the text you enter only if the record contains that text as a whole word or series of whole words. It does not match text within a word. (To match text within a word, use wildcard characters outside the quotation marks.)

Searching with Wildcard Characters

To search for words that have spelling variations or contain a specified pattern of characters, use **?** and ***** (called wildcard characters) to represent the variations.

? Represents any single alphanumeric character in the position it occupies.

***** Represents zero, one, or multiple alphanumeric characters in the position it occupies.

NVIDIA Ex1023
Page O-43

Notes:

- The **?** wildcard can appear at the beginning of a word, in the middle of a word, or at the end of a word.
- The ***** wildcard can appear at the end of a word to find words with different endings, or in the middle of a word to find words containing a specified pattern of characters. The ***** wildcard can *not* appear at the beginning of a word.
- You cannot use **?** or ***** within quotation marks when you specifying an exact match.
- Wildcard matching must match all characters explicitly specified. If you enter the keyword **cable**, IEEE *Xplore* also looks for records containing the stem variation **cabling**. If you enter **cable***, IEEE *Xplore* does not match **cabling**, because there is no **e** to match.
- The ***** wildcard can match words that are not the stem variations that normal IEEE *Xplore* searching matches. For example, **cable*** matches *cableguns* and *cableway*, but **cable** does not; it matches words with common suffixes such as **-ed**, **-es**, and **-ing**.

Examples:

cable?

Matches any record containing six-character words such as *cables* or *cabled* in any field. Does not match *cable* (**?** must match one character)*, cabling, cableguns,* or *cableway*.

t?re

Matches any record containing four-character words such as *tire, tyre,* or *tore* in any field. Does not match *tires* or *three*.

???agon

Matches *hexagon* or *paragon*. Does not match *pentagon*.

t?re*

Matches *tire* (in which ***** matches no characters), *tyre, tires, tyres, three,* and *threshold*.

m*n*

Matches man, mean, median, moon, magnification, modification, management, mnemonic, and any other word in which **m** is followed by **n**.

electro*

Matches electron, electrons, electronic, electronics, electromagnetic, electromechanical, electrostatic, and electrolyte.

optic* <in> de

Matches records containing *optic, optics,* or *optical* in the Index Terms field.

C3?.2*

Matches records containing *C37.20.2-1993*, *C37.20.2b-1994*, or *C37.26-1972* (IEEE standards).

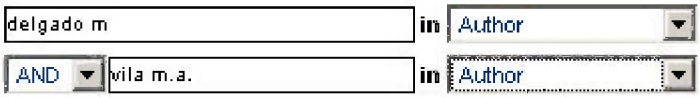
NVIDIA Ex1023
Page O-44

Specifying Multiple Criteria

If you are searching for documents meeting any of several criteria, you need to define the relationship between your search strings using search operators. The following examples assume you are using structured Advanced Search (Option 1).

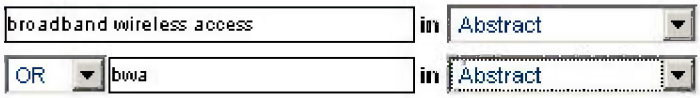
Specifying a combination of criteria:

To search for articles meeting a combination of criteria (for example, articles jointly written by M. Delgado and M.A. Vila), select the **And** operator to combine items. Your search results will list all articles meeting all criteria you specify. For example:



Specifying either of several criteria:

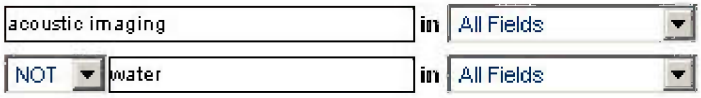
To search for documents meeting any one of the criteria, select the **Or** operator to combine items. Your search results will list all documents containing any of the items you specify. For example:



In this example, IEEE *Xplore* finds documents that contain either the phrase **broadband wireless access**, or the abbreviation **BWA**, in the Abstract field.

Specifying the exclusion of criteria:

If you want to exclude an item from your search, use the **Not** operator. Your search results will include those documents that meet the first criteria, but exclude the second. For example:



In this example, IEEE *Xplore* finds documents that contain the phrase **acoustic imaging** in any field, but not the word **water**.

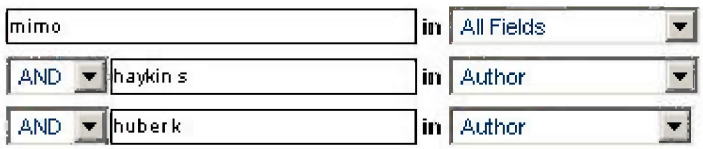
Nesting Search Operations

When a search query consists of multiple operations, you want to make sure IEEE *Xplore* performs the operations in the order you intend.

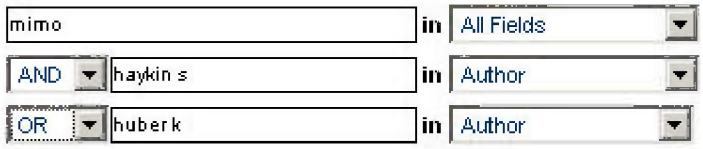
In Structured Advanced Search:

In structured Advanced Search (Option 1), the order of search operations is governed by the order of criteria in the search query boxes. If you use all three search query boxes, the expression formed by the first two boxes takes precedence over the entry in the third box (in other words, the first operator has precedence over the second).

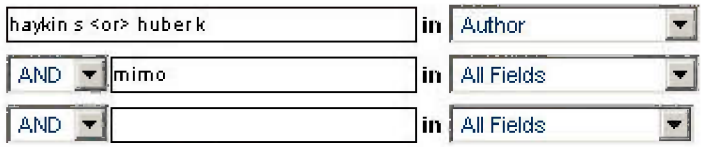
For example, if you want to find all articles on **MIMO** written by both S. Haykin and K. Huber, you would enter the following search selections:



However, suppose you want to find articles by either of these authors about **MIMO**. If you specify:



IEEE *Xplore* finds articles by Haykin about **MIMO**, and all articles by Huber regardless of subject. That is, IEEE *Xplore* processes **And** operations first. (If you specify the **Or** condition first, using separate boxes, IEEE *Xplore* still processes the **And** operation first.) To instruct IEEE *Xplore* to process an **Or** operation before an **And** operation, enter the entire **Or** expression into a single box, using the <or> operator as follows:



This search query finds documents written by either Haykin or Huber (or both) that contain **MIMO** in any field.

In Free-Form Advanced Search:

In the free-form Advanced Search (Option 2), use parentheses to group elements as operations and to specify which operations take place before others. For example, in the search query:

mimo <and> ((Haykin S <in> au) <or> (Huber K <in> au))

IEEE *Xplore* performs the **In** operations first, the **Or** operation second, and the **And** operation third. That is, it looks for records containing either **Haykin S** or **Huber K** in the author field, plus the word **MIMO** in any field. Without the parentheses, the search query would be:

mimo <and> Haykin S <in> au <or> Huber K <in> au

In this case, IEEE *Xplore* looks for documents about **MIMO** by **Haykin** and all documents by **Huber**, regardless of subject. It processes the search query from left to right, as if parentheses surrounded "mimo <and> Haykin S <in> au".

Searching with Incomplete Information

Here are some tips when you have limited information with which to begin a search, but want to minimize the time you spend searching.

If you do not know the exact spelling of an author's name:

- Using Author Search, type the beginning letters of the author's name in the **Quick Find** box. Click » and scan the resulting author list for a name you recognize.

  or

- Using Advanced Search, type as many letters as you know of the author's name in the first text box, and use wildcard characters to specify letters you do not know or are unsure of. Then click Run Search and scan the search results for a name you recognize.

NVIDIA Ex1023
Page O-46

If you do not know the full title of a publication:

- Select the collection (**Journals**, **Conference Proceedings**, or **Standards**) under BROWSE in the navigation menu bar. Type as many words as you know from the publication title in the **Browse by keyword** box. Click » and scan the results list for a title you recognize.

  or

- Using Advanced Search, type as many words as you know from the publication title in the first text box and select **Publication Title** from the pull-down menu. Then select the **Or** operator, and enter keywords appropriate to your search in the second text box. Click Run Search and scan the search results for a title you recognize.

Narrowing a Search

Your first search attempt might produce a long list of results. To reduce the number of results and improve their relevance, you can narrow your search criteria in the following ways:

- Read the citations of relevant titles, looking for additional keywords related to your search. Then modify your search query to include those additional keywords.
- Select a specific collection or date range.
- Analyze the results list for relevant titles, and note recurring authors or publications. Then add an author or publication keyword to your search query using the **And** operator. (See Specifying Multiple Criteria.)

Note: Observe the guidelines for nesting search operations when you combine **Or** and **And** operators.

Broadening a Search

If IEEE *Xplore* finds few or no documents matching your criteria, you can broaden your search criteria to find more matches. Try these techniques:

- Search in full text:



- Add an alternative search term to your original specification, using the **Or** operator. (See Specifying Multiple Criteria.)
- Use truncation and the * wildcard character to specify additional variations of a search term.

  Example: To find documents about **photosensitive** materials or **photosensitivity**, you can specify:

  photosensitiv*

  IEEE *Xplore* returns all matches containing words that begin with **photosensitiv**.

- Use the **?** wildcard character to specify a single unknown or varying character in your search term.

  Example: Suppose you know that someone named Gardner or Gartner wrote an article about the use of robots. You can specify **robots** for the subject term, and for the author you can specify:

  Gar?ner

  IEEE *Xplore* returns all documents about robots written by either Gardner or Gartner.

Working with Search Results

The Search Results page lists descriptions of documents that meet your search criteria. This page appears following a search using Basic Search, Advanced Search, Author Search, or Searching within a Publication.

The top of the Search Results page provides a search results summary. For example:



Order of search results: Search results are organized by relevance in descending order with 25 results per page. Users who log in can change these settings on the Advanced Search page under **Organize results**.

Format of search results: The Citation display format includes the document title, author(s)/editor(s), the publication title, and page numbers. Users who log in can select Citation & Abstract under **Display Format**, which displays 30 to 40 words of the document Abstract with each result.

Full-text access: The Subscriber icon  indicates documents where you have access to full text.

Collection descriptor: The last line of each result identifies the publisher of the document (IEEE or IEE), tells you whether a document appears in a journal (JNL), conference proceeding (CNF), or standard (STD), and provides choices for viewing the document content.

From the Search Results page, you can:

All visitors:

- Print the search results. Use the 

button to display just the search results for printing.
- Email the search results. Use the 

button to open a separate **Email an IEEE *Xplore* Page** window. Fill in the fields and click 

.
- View multiple pages of results by using the result-range links, such as 

.
- View the Abstract of a specific article by clicking the

link.
- For articles to which you do not have access:

  IEEE Members: You can access full text by filing the documents in your IEEE Member Digital Library File Cabinet (if you subscribe) or by purchasing the documents.

If you belong to a **subscribing institution**: You can access full text by filing the documents in your institution's IEEE Enterprise File Cabinet (if your institution subscribes) or by purchasing the documents.

Other guests: You can access full text by purchasing the documents.

Institutional Subscribers and IEEE Members (login or access authentication required):

- View more detail for all results by choosing Citation & Abstract under **Display Format**.
- Select specific results to view on a separate page by clicking the checkbox (☑) before items and then clicking View Selected Items. The View Selected Items page appears, listing your selected search results in the Citation & Abstract format.
- If you have full-text access to all results on the page, the **Download Citations** choice appears. You can download citations for all items on the page.
- View the extended Abstract record for a document by clicking AbstractPlus.
- Revise your search specification to generate broader or narrower results using the **Modify Search** query box.
- View a list of previous searches conducted during the current login session by clicking View Session History. You can use this list to repeat a search.
- Start a new search by clicking New Search.

Subscribers to the publication:

- View the list of references contained in a document by clicking References. The list of references opens in a separate page.
- View the full text of a document to which you or your institution subscribes by clicking PDF. If IEEE does not hold copyright to the document or the document is not available in IEEE *Explore*, instead of this link you will see [Buy Via Ask*IEEE], which you can click to purchase the document.

Viewing Selected Search Results

The View Selected Items page displays a subset of your search results. To return from this page to your full search results, click View Search Results.

From the View Selected Items page, you can:

Institutional Subscribers and IEEE Members (login or access authentication required):

- Print the page. Use the printer friendly button to display just the selected results for printing.
- Email the list of selected items. Use the e-mail button to open a separate **Email an IEEE *Xplore* Page** window. Fill in the fields and click Send.

NVIDIA Ex1023
Page O-49

- View the extended Abstract record for a document by clicking AbstractPlus.
- For articles to which you do not have access:

  IEEE Members: You can access full text by filing the documents in your IEEE Member Digital Library File Cabinet (if you subscribe) or by purchasing the documents.

  If you belong to a **subscribing institution**: You can access full text by filing the documents in your institution's IEEE Enterprise File Cabinet (if your institution subscribes) or by purchasing the documents.

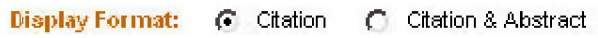
  Other guests: You can access full text by purchasing the documents.
- If you have full-text access to all results on the page, the **Download Citations** choice appears. You can download citations for all items on the page.

Subscribers to the publication:

- View the list of references contained in a document by clicking References. The list of references opens in a separate page.
- View the full text of a document to which you or your institution subscribes by clicking PDF. If IEEE does not hold copyright to the document or the document is not available in IEEE *Explore*, instead of this link you will see [Buy Via Ask*IEEE], which you can click to purchase the document.

Adjusting the Search Results Format

On the search results page, users who log in can adjust the search results format by choosing between the **Citation** format and the **Citation & Abstract** format.



- The **Citation** format displays the document title, author(s)/editor(s), the publication title, and page numbers.
- The **Citation & Abstract** format displays 30 to 40 words of the document Abstract with each result.

To change the sort order or number of results per page, click your browser **Back** button and make the appropriate selections under **Organize results**. Then click » to repeat the search with the format adjustments.

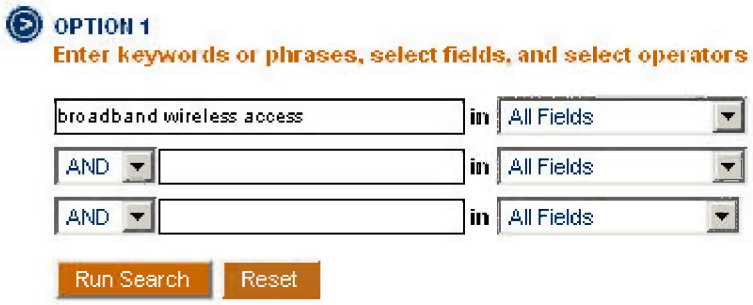
NVIDIA Ex1023
Page O-50

Search Examples

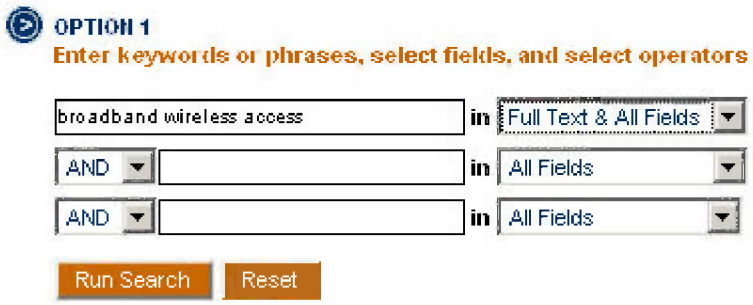
Structured Advanced Search Examples (Option 1)

Suppose you want to find articles about **broadband wireless access** technology.

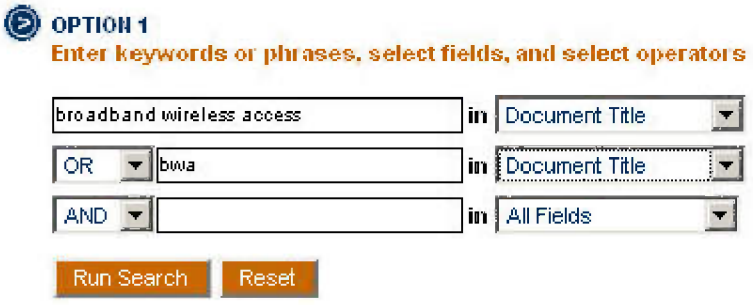
1. Hover on or click SEARCH and then click **Advanced**.
2. You can construct a search query that searches for the phrase **broadband wireless access** in all fields:



3. You can expand this search to a full-text search by selecting **Full Text & All Fields**:

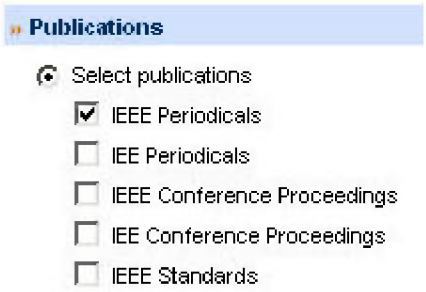


4. Suppose you want only articles with this topic in their title. In addition, you know that the term is sometimes abbreviated to **BWA**. You can look for either topic in the document title by using the **Or** operator as follows:



Limiting Records Searched:

To limit this search to the IEEE Periodicals collection (journals, magazines, and transactions), deselect the other choices under **Publications**.

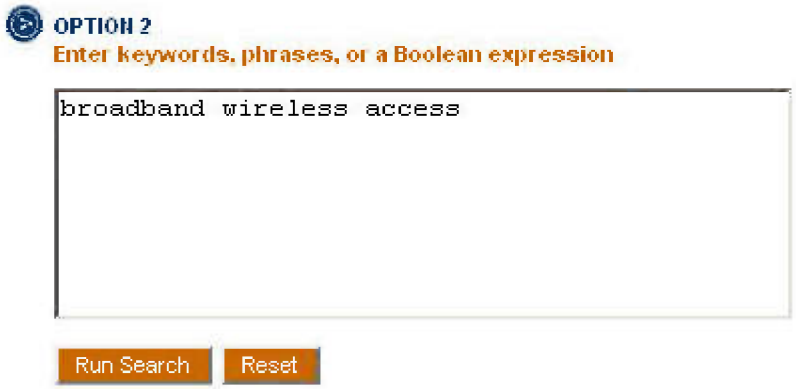


Also, you can narrow the search to a specific date range by using **Select date range**. See “Limiting a Search to Specific Years” for more information.

Free-Form Advanced Search Examples (Option 2)

Suppose you want to find articles about **broadband wireless access** technology.

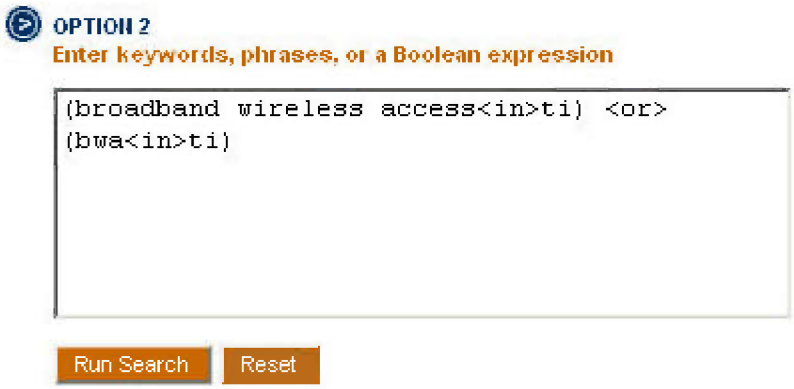
1. Hover on or click SEARCH and then click **Advanced**.
2. You can use the free-form area to enter a search query that searches for the phrase **broadband wireless access** in all fields:



3. You can expand this search to all fields and full text as follows:



4. Suppose you want only articles with **broadband wireless access** in their title. In addition, you know that the phrase is sometimes abbreviated to **BWA**. You can look for either topic in the document title by using the **Or** operator as follows:



Limiting Records Searched:

To limit this search to the IEEE Periodicals collection (journals, magazines, and transactions), deselect the other choices under **Publications**.



Also, you can narrow the search to a specific date range by using **Select date range**. See "Limiting a Search to Specific Years" for more information.

Complex Free-Form Advanced Search Examples (Option 2)

In free-form advanced search (Option 2), you can specify and perform more complex searches than you can using structured advanced search (Option 1). You can use many search terms and a variety of search operators, and you can control the order in which to evaluate expressions.

Using logical operators and parentheses:

You create more complex queries using free-form advanced search by joining search expressions with logical operators and using parentheses to group and prioritize parts of the query. Although parentheses are not always required, they help you to see the structure of your query more clearly, and may make it easier for you to edit the query.

Here is an example of a more complex query:

(<or>(rfid,radio frequency identification)) <and> (<or>(privacy,security))

NVIDIA Ex1023
Page O-53

With some operators, like the two **<or>** operators in the example, you can enter the operator only once and follow it with a list of the search terms it operates on. This form makes it a little easier to create and read search expressions.

Using other operators and modifiers:

In free-form advanced search, you have access to operators you cannot use in structured advanced search, such as **<near>**, **<order>**, and **<many>**.

Here is an example of a complex query using these operators:

<many>(<order>(<and> (moon, man, vehicle)))

First, a record must contain some variation of all three terms before it can match. Then, those three terms must appear within the record in the order shown. After all matching records are found, IEEE *Xplore* assigns a score to each one, so that you can sort them by score and view together those with dense occurrences of these terms.

Author Search Examples

Suppose you want to find a conference paper by J.R. Tuttle about RFID (radio frequency identification) technologies. Here are three ways you perform an author search in IEEE *Xplore* to find the paper:

- Using Author Search
- Using Structured Advanced Search (Option 1)
- Using Free-Form Advanced Search (Option 2)

Using Author Search

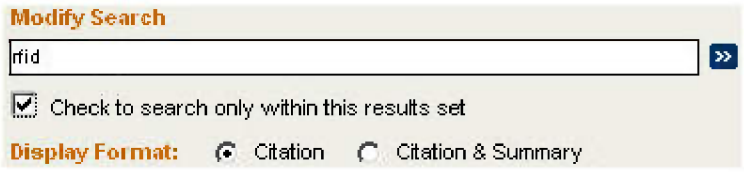
1. Hover on or click SEARCH and then click **Author**.
2. Enter the author's last name and first initial in the box.



3. Select the author from the list of results.



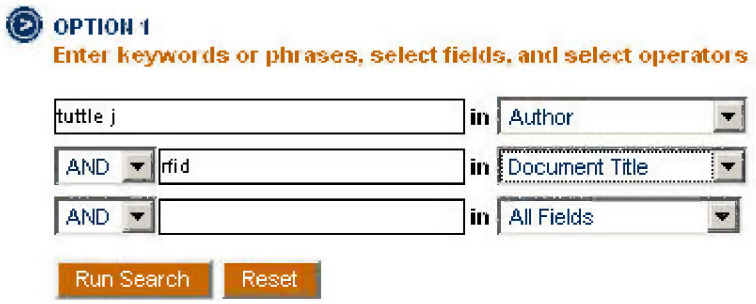
4. If the list of documents is long, perform a search within the results.



NVIDIA Ex1023
Page O-54

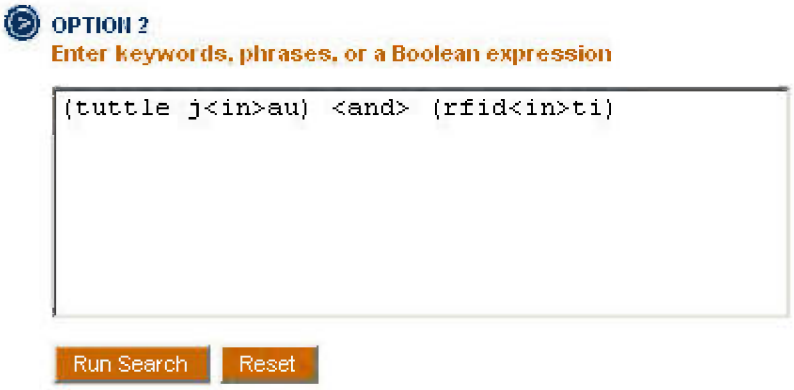
Using Structured Advanced Search (Option 1)

1. Hover on or click SEARCH and then click **Advanced**.
2. Construct a search query that searches for **Tuttle J** in the author field and **RFID** in the document title field.



Using Free-Form Advanced Search (Option 2)

1. Hover on or click SEARCH and then click **Advanced**.
2. Construct a free-form search query that searches for **Tuttle J** in the author field and **RFID** in the document title field.



Working with Abstract and AbstractPlus Records

Abstract and AbstractPlus Pages

The Abstract page provides full Abstract records for most documents, including the title, author(s), publication identity, and page numbers, which are also listed on the Search Results page. Everyone can view an Abstract.

Logged-in users can view the AbstractPlus page. It provides additional information; see “Abstracts and AbstractPlus” for more information.

From the Abstract page, you can:

- Read the full bibliographic citation for the document.
- Read the full Abstract.
- Print the page.
- Access full text:

  IEEE Members: You can access full text by filing the document in your IEEE Member Digital Library File Cabinet (if you subscribe) or by purchasing the document.

  If you belong to a **subscribing institution**: You can access full text by filing the document in your institution’s IEEE Enterprise File Cabinet (if your institution subscribes) or by purchasing the document.

  Other guests: You can access full text by purchasing the document.

If you see the **login area** at the side of the Abstract page, log in to access the AbstractPlus page.

From the AbstractPlus page, you can:

- Read the full bibliographic citation and Abstract for the document.
- See occurrences of the search terms you specified, indicated by **words shown in red**.
- Print or email the page.
- Access full text:

  Subscribers to the publication: Click PDF to view full text.

  IEEE Members who do not subscribe to the publication: You can access full text by filing the document in your IEEE Member Digital Library File Cabinet (if you subscribe) or by purchasing the document.

  If you belong to a **subscribing institution**: You can access full text by filing the document in your institution’s IEEE Enterprise File Cabinet (if your institution subscribes) or by purchasing the document.

  Other guests: You can access full text by purchasing the document.

NVIDIA Ex1023
Page O-56

- Search on author names conveniently by clicking the live author-name links.
- Search on subject terms conveniently by clicking the live index-term links.
- View the list of references in the document, if available, and view the individual Abstracts and full text for each reference (available to publication subscribers).
- View a referenced document if it is available via a CrossRef link to the publisher's web site.
- View the list of documents that cite the current document, if available, and view the individual Abstracts and full text for each document (available to publication subscribers).
- Download the document citation by choosing the citation format and file format in the **Download this citation** area.
- Arrange for rights and permissions by clicking 

, which takes you to the Copyright Clearance Center's Rightslink electronic permissions service page.
- View the table of contents for the publication containing the article by clicking View TOC.
- View the next or previous article or paper in the publication by using Previous Article | Next Article.

The Abstract record has "Not Available" in the Abstract section if it represents one of the following:

- An interim IEEE-created record that will be replaced when the final INSPEC bibliographic record is available.
- An IEEE-created record for a document (such as a letter to the editor, paper submission guidelines, or a new product review) for which INSPEC supplies no bibliographic record.
- Reprint articles for which IEEE does not hold copyright. However, you can purchase these articles by clicking the [Buy Via Ask*IEEE] link.

Downloading Citations

The **Download this citation** area on the AbstractPlus page enables you to download Abstract records into your bibliographic management program, such as EndNote. This feature aids you in collecting pre-formatted citation information for future reference needs.

To download Abstract information:

1. Select **Citation** or **Citation & Abstract** in the Download field.
2. Select **EndNote,ProCite,RefMan** or **ASCII Text** in the File Format field.
3. Click Download. The following occurs:
   - If you chose **EndNote,ProCite,RefMan** format, a File Download dialog box appears which allows you to either open the file or save it to disk. Choose a method and click **OK** to complete the download.
   - If you chose **ASCII Text** format, the text appears in a new browser window for you to read or copy and paste into another location.

Arranging for Rights and Permissions

The Copyright Clearance Center's Rightslink electronic permissions service offers users the opportunity to obtain near-instant rights to reuse IEEE-copyrighted material in books, journals, magazines, newsletters, CD- or DVD-ROMs, brochures/pamphlets, coursepacks and classrooms.

When you click 

on a document's AbstractPlus page, the CCC Rightslink page opens. It explains the terms for obtaining reuse rights. After you have read the information, you can continue the process by clicking Proceed to Rightslink. A separate Rightslink page opens to guide you through the process.

NVIDIA Ex1023
Page O-58

Working with Documents

Viewing a PDF Document

IEEE *Xplore* uses Adobe Acrobat Reader to present complete full-page images of documents in PDF format. These images show all text and illustrative material, as published. Click PDF on a Search Results, Abstract or AbstractPlus, or Table of Contents page to view the full text of the document.

Note: Viewing a full-text PDF is available to the publication's subscribers, for documents filed in either the IEEE Member Digital Library File Cabinet or the IEEE Enterprise File Cabinet, or for purchased documents.

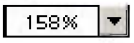
When viewing a PDF document, you can:

- View the displayed document page, using:

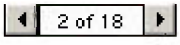
   to enlarge the image size

   to decrease the image size

   to view the image at 100%

  to select a specific image viewing size

- Move the document page on the screen by holding down the left mouse button as you move the cursor over the page image. The cursor appears as a hand ().
- View the next pages of the document by clicking  .
- View the last page of the document by clicking  .
- Return to previous pages of the document by clicking  .
- Return to the first page of the document by clicking  .
- Go to a specific page by clicking 

  in the lower-left corner and either entering a number or clicking an arrow to go forward or go back one page.
- Print the entire article or the current page by clicking the  button.

Viewing HTML Full-Text

If a document is available in HTML full-text format, you can click the HTML link in the search results listing or Abstract to view the document in HTML format. You need to log in as a subscriber to the publication to view the full text. Use your browser Print button to print the document.

Use your browser Back button to return to your search results or Abstract page.

NVIDIA Ex1023
Page O-59

Printing Documents

Most full-text documents available in IEEE *Xplore* are in PDF format. The *IEEE Spectrum* magazine and *Proceedings of the IEEE* also contain some articles in HTML full-text format.

To print the full text of a PDF document:

1. While viewing the PDF document, click  at the top of the page viewer window.
2. In the **Print** window, choose which pages to print by selecting **All pages** or specifying page numbers in the **Pages from** boxes.
3. Click **OK.** The selected pages are printed.

To print the full text of an HTML document:

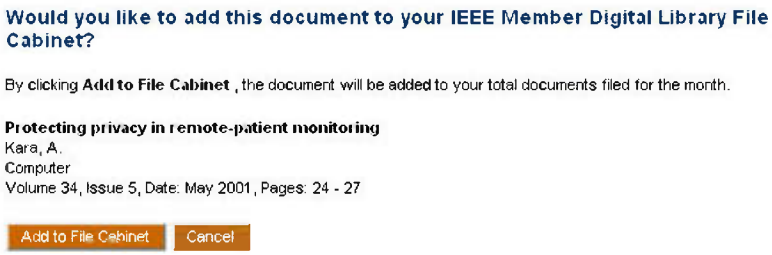
1. While viewing the HTML document, click the **Print** button in your browser toolbar.
2. In the **Print** window, choose which pages to print by selecting **All pages** or specifying page numbers in the **Pages from** boxes.
3. Click **OK.** The selected pages are printed.

Filing Documents in the IEEE Member Digital Library (IEEE Members)

If you subscribe to the IEEE Member Digital Library, you can file and access up to 25 documents each month that are not part of your online subscriptions. The documents you can file must be in IEEE journals, magazines, transactions, letters, or conference papers.

To file a document in your IEEE Member Digital Library File Cabinet:

1. In the search results list or the AbstractPlus record, click PDF to access full text for the document. If the publication is not part of your subscription, you receive a message asking if you want to file the document, similar to this example:

   In the **Account Information** section, you can see how many documents are already filed in your IEEE Member Digital Library File Cabinet and how many more you can file for the current month.

2. Click  to file the document in your IEEE Member Digital Library File Cabinet.

   Note: You cannot undo the action of storing an article in the File Cabinet.

   An updated view of your IEEE Member Digital Library File Cabinet appears.

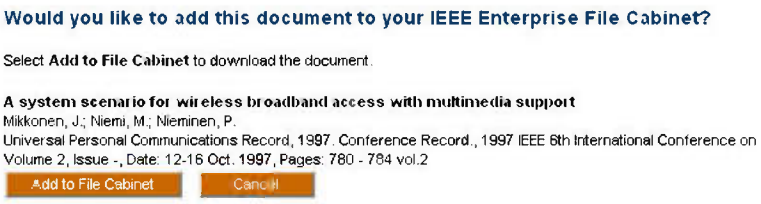
NVIDIA Ex1023
Page O-60

If you see the message, "The document you have selected was previously downloaded to your File Cabinet", you can view the document you just filed by opening your IEEE Member Digital Library File Cabinet and clicking PDF below the document citation.

Filing Documents in the IEEE Enterprise File Cabinet

To file a document in your institution's IEEE Enterprise File Cabinet:

1. In the search results list or the AbstractPlus record, click PDF to access full text for the document. You receive a message asking if you want to file the document, similar to this example:



   In the **Account Information** section, you can see how many documents are already filed in your IEEE Enterprise File Cabinet.

2. Click Add to File Cabinet to file the document in the Enterprise File Cabinet.

   Note: You cannot undo the action of storing an article in the File Cabinet.

   An updated view of the IEEE Enterprise File Cabinet appears.

If you see the message, "The document you have selected was previously downloaded to your File Cabinet", you can view the document you just filed by opening your IEEE Enterprise File Cabinet and clicking PDF below the document citation.

Purchasing a Document

When you click PDF to view an article, if you do not subscribe to the publication in which it appears, IEEE *Xplore* displays a login page. Below the login area is the IEEE Article Purchase area.

1. To begin the purchase process, click Buy this document now.

   IEEE *Xplore* displays the IEEE Article Purchase page. This page confirms that you do not subscribe to the article, presents the citation for the article you are purchasing, and provides the price of the article. It also indicates how you may access the article after you complete your purchase.

2. Click Buy this document now to continue the process.

   IEEE *Xplore* presents the IEEE Article Purchase form.

3. Complete the form. Be sure to click the checkbox indicating you accept the terms and conditions.

4. Click Submit to complete the transaction (click it only *once*).

   An email confirmation will be sent to you indicating the web address for downloading the full text of the document you purchased.

NVIDIA Ex1023
Page O-61

IEEE *Xplore* Messages

A denial message appears when you attempt to use a feature or view a document that you are not allowed to access given your current login status.

Login Messages and Denials

Login Denial

Most users need to log in to gain access to search functions and documents. (See "Do I Need To Log In?" for detailed information about whether you need to log in.) If you are presented the Login page, and you are:

A Non-Member/Guest

You do not need to log in. You can perform a Basic Search of all brief Abstracts and access all tables of contents and brief Abstracts of all content available in IEEE *Xplore*.

An IEEE Member

Have you set up a username and password? IEEE members need to establish an IEEE Web Account before they can log in.

Or, you may be attempting to access content for which you do not have a subscription. Click Access Information at the top to see which publications are in your subscription.

Employed by an IEEE Subscribing Institution

Are you required to log in to view your institution's subscriptions? Some institutions are identified by IP address. Check with your librarian or system administrator.

Are you using your institution's username and password? Check with your system administrator or librarian to verify that it is correct.

Finally, you may be attempting to access content for which your institution does not have a subscription. Click Access Information at the top to see which publications are in your institution's subscription.

NVIDIA Ex1023
Page O-62

Invalid Login

IEEE *Xplore* did not recognize the username and password combination you entered. Please try again, using these guidelines:

- Usernames and passwords are case sensitive. Be sure the CAPS LOCK key is not depressed, and enter the username and password using the correct case.
- If you are copying and pasting your username or password from another source, be sure you are not including any extraneous characters, such as a trailing space.

If you are an IEEE Member and your IEEE Web Account username and password is not recognized, please contact IEEE member services at member-services@ieee.org for assistance.

Already Logged In

You have entered a username and password while already logged in to IEEE *Xplore*. Your existing session will be replaced when you log in again, and recent searches in your session history will be deleted.

Enter your login information to continue beginning a new session, or use your browser Back button to return to your previous session.

Request Invalid

A denial message appears if you attempt to view full text without authorized access. If you attempted to view full text and were denied, and you are:

A Non-Member/Guest — You do not have access to full texts.

An IEEE Member — You have access to the full text of *IEEE Spectrum*, any personal IEEE online subscriptions, any articles eligible for filing in your IEEE Member Digital Library File Cabinet (IEEE Member Digital Library subscribers only), and any articles available for single article purchase. If you receive a denial message:

- Try logging out and in again.
- Check that the document you are trying to access is included in your personal IEEE online subscriptions.
- Consider filing the document in your IEEE Member Digital Library File Cabinet, if you are an IEEE Member Digital Library subscriber.
- If you are employed by a subscribing institution and the document is part of the institution's subscription, log out of your IEEE Web Account and log into your institution's IEEE *Xplore* account to access the document.

Otherwise, please contact IEEE member services at member-services@ieee.org for assistance.

Employed by an IEEE Subscribing Institution

You have access to the full text of all documents in your institution's subscription package. If you receive a denial message:

- Try logging out and in again.
- Check that the document you are trying to access is included in your institution's IEEE online subscription.
- Consider filing the document in your IEEE Enterprise File Cabinet, if your institution subscribes.
- Consider purchasing the document you are trying to access, if Buy this document now appears at the bottom of the login area.

Otherwise, please contact customer support at onlinesupport@ieee.org for assistance.

Search Invalid

A denial message appears if you attempt to use one of the search functions — Author Search, Advanced Search, CrossRef Search Pilot, or Searching within a Publication — without authorized access. If you attempted to perform a search and were denied, and you are:

A Non-Member/Guest

You do not have access to these search functions. You can use the browse and Basic Search functions.

An IEEE Member

You have access to all search and browse functions. If you are having trouble accessing one of these functions, your login may have timed out. Try logging in again, or try logging out and in again. If you still have difficulty, please contact IEEE member services at member-services@ieee.org for assistance.

Employed by an IEEE Subscribing Institution

You have access to all search and browse functions. If you are having trouble accessing one of these functions, your login may have timed out. Try logging in again, or try logging out and in again. If you still have difficulty, please contact your librarian or system administrator.

Otherwise, please contact customer support at onlinesupport@ieee.org for assistance.

View Abstract Denial

Anyone can view an Abstract without logging in to IEEE *Xplore*. If you view an Abstract and you are:

A Non-Member/Guest

You can view brief Abstracts but not the more detailed AbstractPlus.

NVIDIA Ex1023
Page O-64

An IEEE Member	You have access to detailed AbstractPlus records if you log in. If you think you are already logged in, try logging out and in again. If you are still having difficulty, please contact IEEE member services at member-services@ieee.org for assistance.
Employed by an IEEE Subscribing Institution	You have access to AbstractPlus records of documents included in your subscription if you log in. If you think you are already logged in, try logging out and in again. If you are still having difficulty, contact your system administrator or librarian. Otherwise, please contact customer support at onlinesupport@ieee.org for assistance.

Concurrency Denial

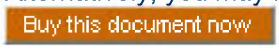
Your institution's subscription includes a limited number of concurrent connections. All connections are currently busy. Please try accessing IEEE *Xplore* again in a few minutes.

- You can browse tables of contents without logging in.
- If you are an IEEE Member, log in with your IEEE Web Account to access full text in your personal subscriptions.
- If you continue having difficulty logging in, contact your system administrator or librarian.

IEEE Member Digital Library Limit Exceeded

Your IEEE Member Digital Library File Cabinet already contains 25 documents for the current month. You cannot add any more documents to your File Cabinet until the first day of the next month.

If you are employed by an institution with an IEEE online subscription, you can log out and then log back in with your institution's login ID.

Alternatively, you may be able to purchase the document. Look for the 

button.

Note: You cannot purchase the document in IEEE *Xplore* if it is an IEEE standard.

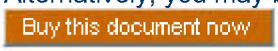
NVIDIA Ex1023
Page O-65

IEEE Enterprise Limit Exceeded

Your institution's File Cabinet already contains the maximum number of documents for its subscription. You can continue to access documents that are in the File Cabinet until the noted expiration date. To download additional documents, the institution's IEEE Enterprise account must be renewed.

Check with your institution's information professional or librarian to see if a copy of the document is available in your library.

If you are an IEEE Member and the document is in your personal subscriptions, you can log out and then log back with your IEEE Web Account to access the document.

Alternatively, you may be able to purchase the document. Look for the 

button.

Note: You cannot purchase the document in IEEE *Xplore* if it is an IEEE standard.

NVIDIA Ex1023
Page O-66

Glossary

A

Abstract: A summary of the contents of a document. The Abstract commonly explains the main points made in an article or paper, including the main theme, argument, process, and results.

Abstract record: A description of an article, paper, or standard that includes the following information: author name(s); editor name(s); title of the article, paper, or standard; title, volume/issue number, page number, and date of the publication in which the article or paper appeared and other identifying information including the ISSN or ISBN number.

AbstractPlus: The AbstractPlus page provides the same information as the Abstract page, plus: INSPEC accession numbers, live index terms for convenient keyword searching, live author links for convenient by-author searching, references with live links to documents, the Printer Friendly button, link to documents that cite the current document, download Citation capability, a Rightslink button that takes you to the Copyright Clearance Center's Rightslink electronic permissions service page, and where you can arrange for copy and use rights.

Access authentication: Verifying the identity of a user accessing IEEE *Xplore* based on the user's IP address.

Accession number, INSPEC: The record number created when the bibliographic record was added to the INSPEC database, a component of IEEE *Xplore*. All IEEE *Xplore* documents have a unique INSPEC accession number.

Adobe Acrobat Reader: The viewer software used by IEEE *Xplore* to display PDF (portable document format) documents.

Affiliation: The organization or company employing the author of an article or paper, or the corporate source if no personal authors are credited.

Alerts, email: Email messages that let you know when a publication you have specified is updated. Click the Alerts button to display instructions for setting up email alerts.

Alphanumeric character: Any letter A-Z or a-z, or any digit 0-9.

And operator: Indicates the relationship between your search criteria. And means results should match all criteria.

ANSI: The American National Standards Institute, an organization that coordinates U.S. voluntary standards for a wide range of business, public, and consumer areas.

Author: The author or presenter of the document as published. On the AbstractPlus page, you can click an underlined author name to search by that author.

NVIDIA Ex1023
Page O-67

Author list: A complete listing of all authors in the IEEE *Xplore* database, which can be accessed from the Author Search page. You can search by keyword or browse alphabetically through the list for a particular author. Clicking an author name returns a list of articles and papers by that author.

B

Back button: A button on the browser toolbar that enables you to go to the previously viewed page. For example, while viewing a list of search results, you can click the Back button to return to the search page that generated those results.

Browser: The viewing software through which you use IEEE *Xplore*. To read about browser requirements for using IEEE *Xplore*, see the IEEE *Xplore* Guide topic "What are the optimal system requirements for using IEEE *Xplore*?".

Buy Via Ask*IEEE: The Buy Via Ask*IEEE link enables you to purchase articles for which IEEE does not hold copyright or if the document is not available in IEEE *Xplore*. This link appears on Abstract and/or AbstractPlus pages.

C

Case sensitive: Making a distinction between characters entered in uppercase (capital letters) and lowercase.

Catalog number: The IEEE Catalog Number for the document.

CODEN: A unique alphanumeric code assigned to serial and monographic publications and used as a permanent identifier. The CODEN system was developed in 1963 by the American Society for Testing and Materials for scientific and technical publications, but has since expanded in scope. In 1975, the Chemical Abstracts Service assumed responsibility for the assignment and dissemination of CODEN designations.

Colloquium: A meeting or conference, usually conducted by the IEE.

Conference date: Date of the conference where the paper was presented.

Conference proceedings list: A complete listing of all conference proceedings in the IEEE *Xplore* database which can be accessed from the Browse Conference Proceedings page. You can search by keyword or browse alphabetically for a particular conference. Clicking a conference title returns a list of volumes for that conference.

Conference title: The name of a conference or meeting.

Country: The country in which the conference was held.

D

Document: An article, paper, or standard stored as a record in IEEE *Xplore*.

Document title: The full title of the document as published, also referred to as document title. Conference paper titles and standard titles are also found under the generic term Document Title in IEEE *Xplore*.

NVIDIA Ex1023
Page O-68

Document type: The publication category of the documents in IEEE *Xplore*. CNF indicates conference proceedings, JNL indicates periodicals (journals, magazine, and transactions), and STD indicates standards.

E

Email alert: Email messages that let you know when a publication you have specified is updated. Click the Alerts button to display instructions for setting up email alerts.

End page: The ending page of the article within the publication.

F

Field: A specific type of information within an IEEE *Xplore* record. For example, the Author field, abbreviated au, contains the name of the author of the document.

Field codes: Abbreviations for fields that you use in free-form search expressions. You can access a list of field codes by clicking the Advanced Search page.

Fields menu: A pull-down menu on the Advanced Search page that allows you to select a specific field in which to search for the associated search text.

Full text: The complete text of an article, paper, or standard in its published form, including both the bibliographic descriptions and the articles themselves. In IEEE *Xplore*, full text is available in portable document format (PDF) files. Select content is available as full text in HTML files.

H

Help button: A link appearing at the bottom of IEEE *Xplore* pages that opens a help topic related to your current context.

I

IEL: IEEE/IEE Electronic Library: A digital library of all articles, conference papers, and standards published by IEEE and IEE since 1988, with select content from 1952.

Index (subject) term: A significant word used as a descriptor of the subject of an article or paper. On the AbstractPlus page, you can click an underlined index term to search for documents containing that term.

INSPEC: An English-language information service providing a bibliographic database of Abstracts compiled from scientific and technical literature, of which IEEE *Xplore* uses a subset.

ISBN: The International Standard Book Number: a four-part, ten-character identifying code given to a book (a non-serial literary publication) before publication. The four parts are: group identifier (e.g., national, geographic, language); publisher identifier; title identifier; and check digit. Started by British publishers in 1967, the standard book number has been an international standard since 1969.The numbering system is administered among cooperating publishers in participating countries by a standard book numbering agency. Most IEEE *Xplore* conference publications have a unique ISBN.

NVIDIA Ex1023
Page O-69

ISSN: The International Standard Serial Number: the international numerical code that identifies a serial publication, based on American National Standard Identification Number for Serial Publications, Z39.9-1971, and approved by the International Organization for Standardization as ISO 3297, International Standard Serial Numbering. The ISSN program became operative in the United States in 1971. Each IEEE *Xplore* transaction, magazine, and journal publication has a unique ISSN.

Issue list: The issue list appears in the Issues pull-down menu on the publication page. The most recent issues appear at the top. If you click a year, the list of recent issues is replaced by a list of issues for that year. Click an issue to access its table of contents.

Issue number: The number that uniquely identifies an issue of a periodical or newspaper.

Issue part number: The number that uniquely identifies part of an issue of a periodical or newspaper.

J

Journal: As used in IEEE *Xplore*, any serial publication such as a journal, transaction, letter, or magazine.

Journals list: A complete listing of all journals in the IEEE *Xplore* database, which can be accessed from the Browse Journals & Magazines page. You can search by keyword or browse alphabetically for a particular journal. Clicking a journal title returns a list of issues from that journal.

L

Latest Content Update: The Latest Content Update link on the IEEE *Xplore* Home page displays a page listing links to the added or updated journals and magazines, conference proceedings, and standards for the past four weeks. On that page, you can also view updates for the current and the past year.

Login ID: The username and password you use to log into IEEE *Xplore*.

Login link: A link located at the top of all IEEE *Xplore* pages that starts the login process. If you are already logged in, a “Welcome” message appears in the blue banner.

Logout link: A link located at the top of all IEEE *Xplore* pages that allows you to log out of IEEE *Xplore*.

M

Matches: Records that meet search criteria and are listed on the search results list.

Modify Search query box: An area that appears on the Search Results page, displaying the query just executed and letting you modify the query. After modifying the query, click the Go button.

NVIDIA Ex1023
Page O-70

N

Navigation menu bar: The row of choices (Browse, Search, IEEE *Xplore* Guide, Support) below the banner at the top of each IEEE *Xplore* page that provides links to pages in IEEE *Xplore*.

Nesting operations: The use of parentheses to specify the order in which IEEE *Xplore* processes the parts of a search query. Without any parentheses, IEEE *Xplore* processes free-form queries from left to right. With the use of parentheses, IEEE *Xplore* in parentheses before items outside parentheses, and processes items in the innermost pair of parentheses first.

Not operator: Indicates the relationship between your search criteria. Not means the results should not match this particular criterion.

O

Operator: A syntax element in search expressions that describes a relationship between criteria. Commonly used operators are And, Or, and Not.

Operators menu: A pull-down menu on the Advanced Search page that allows you to specify the relationship between several search strings using the And, Or, and Not operators.

Or operator: Indicates the relationship between your search criteria. Or means results should match any one or more criteria.

P

Password: Part of the information you enter in the Login area, in addition to your username. Passwords are case sensitive.

PDF: PDF (Portable Document Format) is a format that maintains the page layout, color, graphics, and typography of the original document as published. It can be viewed onscreen or printed from any type of computer, using the Adobe Acrobat Reader software.

Personal security identifier: A reminder word to help you recover your password if you forget it. You define this identifier when you set up an IEEE Web Account.

Printer friendly icon: An icon on the Table of Contents, Abstract/AbstractPlus, and Search Results pages that allows you to display the page in a format that is better for printing.

Publication date: The date of publication of the document.

Publication information: Information about the editorial board or staff for the journal selected. This area appears on the left side of the Publication page.

Publication name: The name of the publication in which an article or paper appears.

Publisher: The publisher of the document in which an article or paper appears.

Publisher location: The city and state or province in which the publisher of an article is located.

NVIDIA Ex1023
Page O-71

Q

Query: A single keyword, expression (keywords and operator), or combination of expressions that specifies criteria for locating records. Also referred to as a search string. In IEEE *Xplore*, you use Advanced Search to build search queries, using either the structured or free-form areas.

R

Record: The bibliographic information associated with each article, paper, and standard in the IEEE *Xplore* database.

Reference list: In the Abstract record, the bibliographic references for the article. An AbstractPlus displays hyperlinks to the referenced documents' Abstracts and PDF full-text if the document is in the IEEE *Xplore* database or available via a CrossRef link to the publisher's web site.

Relevance ranking: The determination of relevance of a search result to the query that generated it, according to the number of occurrences of the search term(s) in the resulting document.

Reset button: Clears the search query box (or boxes) so you can reenter your search criteria.

Result-range page links: Numbers such as 26-50 that appear at the top and bottom of each Search Results page, allowing you to go to additional results pages generated by the search.

Rightslink button: The Rightslink button displays the Rightslink page, where you can arrange for copy and use rights. This button appears on the AbstractPlus page.

S

Score: The relevance ranking based on the number of occurrences in the matching document of the search criteria you specified.

Search expression: A combination of search text, operators, and field codes that forms part of a search query.

Search query box: A text box on a search page where you enter your search query.

Search result: On the Search Results page, a description of a document stored in IEEE *Xplore* that matches the search criteria supplied using the Basic, or Advanced, or Author Search.

Search text: Words, partial words, and phrases you enter in a search query box to find matching information in IEEE *Xplore* database records. You can search for exact matches or variations of words and phrases. You can use field codes and search operators with your search text to form search expressions.

Standard number: The identifying number of an IEEE standard.

Standards list: A complete listing of all standards in the IEEE *Xplore* database which can be accessed from the Browse Standards page. You can search by standard number or keyword to locate a particular standard.

Start page: The starting page of the article within the publication.

NVIDIA Ex1023
Page O-72

Stem variation: The stem is the base word for which you are searching. IEEE *Xplore* normally searches for the stem and its variants (the "stem variations", which include endings such as plurals, -ing, and -ed.

Subject term: A significant word used as a descriptor of the subject of an article or paper. On the AbstractPlus page, you can click an underlined index term to search for documents containing that term.

Subscriber icon: An icon that appears on lists of search results next to publications to which you or your institution subscribes.

T

Table of contents listing: A listing of an article, conference paper, or standard on the table of contents page of a publication volume. The listing contains the title, author (if applicable), and page number.

Truncation: Specification of part of a word, to which you add wildcard characters to represent the remainder of the word.

U

Username: Part of the information you enter in the Login area, in addition to your password. Usernames are case sensitive.

V

View Top 100: The View Top 100 link on the IEEE *Xplore* Home page displays a page that lists the most frequently requested documents during the current month.

Volume: The publication volume in which the article appears.

W

Wildcard character: A character used in search queries to represent one or more unspecified characters. The * wildcard character represents any or no characters and the ? wildcard character represents any single character.

NVIDIA Ex1023
Page O-73

Index

*****

* wildcard character....38, 42

?

? wildcard character....38, 42

A

Abstract record....51
AbstractPlus....12
AbstractPlus record
  viewing....51
AbstractPlus record....51
advanced search
  complex free-form examples....48
  examples....46, 47, 48, 49
  fields....32
  nesting operations....40
  operators....33
  structured....28
alerts for updates....5
and (operator)....28, 40
archived update information....4
article purchase....56
author search
  formatting....35
  search examples....49
author search....35

B

basic search....26
bibliographic citation....51
broadening a search....38, 42
browse and search functions....4
browsing
  alphabetically....24, 35
  issues or publications....16, 19, 20
  standards....22
  tables of contents....17, 18, 21
browsing....15

C

CCC Rightslink service....53
characters
  special....37
  uppercase and lowercase....37
  wildcard....38
characters....37
citation
  downloading....53
citation....51
codes
  field....32
codes....32
collections
  IEEE Xplore....1
  limiting search to specific....29
collections....1, 29
combining search expressions....40
complex search expressions....40, 48
conference proceedings
  browsing....19
  finding....24
  home page....19
  publication page....20
  table of contents....21
conference proceedings....1, 2
content updates....4
Copyright Clearance Center Rightslink service53
CrossRef Search Pilot....36

D

do you need to log in?....9
documents
  printing....55
  viewing top 100....5
documents, purchasing....56
documents, viewing HTML full-text....54
documents, viewing PDF full-text....54
downloading citations....53

E

email alerts....5
exact words....38
examples, search
  advanced search....46, 47
  author searches....49
  complex free-form search....48
examples, search....46, 47
excluding records....40
expressions
  complex....48

NVIDIA Ex1023
Page O-74

F

field searching 28
fields
  field code summary 32
fields 2, 32
file cabinet, IEEE Enterprise
  accessing 13
  filing documents 56
  limit exceeded 61
  locating a filed document 23
file cabinet, IEEE Enterprise 56
file cabinet, IEEE Member Digital Library
  filing documents 55
file cabinet, IEEE Member Digital Library 55
formatting author names 35
formatting search results 30, 45
forthcoming articles, table of contents 18
free-form advanced search
  examples 47, 48
  fields 32
full text, viewing 54

H

history, session 36
Home page, IEEE Xplore 6, 7, 8

I

IEEE Enterprise file cabinet
  accessing 13
  filing documents 56
  limit exceeded 61
  locating a filed document 23
IEEE Enterprise file cabinet 56
IEEE Member Digital Library
  accessing 13
  filing documents 55
  limit exceeded 60
  locating a filed document 22
  setting up account 3
IEEE Member Digital Library 3, 13, 55, 60
IEEE Xplore collections 1
IEEE Xplore Home page 6, 7, 8
IEEE Xplore overview 1
in (operator) 28

J

joining search expressions 40
journals
  finding 24
  forthcoming articles 18
  publication page 16
  table of contents 17
journals 1, 2

K

keyword search 26

L

letters
  uppercase and lowercase 37
letters 37
limit exceeded in
  IEEE Enterprise file cabinet 61
  Member Digital Library 60
limit exceeded in 60, 61
limiting a search to specific collections 29
locating a conference proceeding 19
locating a document in the IEEE Enterprise file cabinet 23
locating a document in the IEEE Member Digital Library 22
locating a journal 15
locating a standard 21
logging in
  denials 57, 58, 59, 60
  what can I access? 12
logging in 11, 12
logging out 12
lowercase letters 37

M

magazines
  finding 24
  forthcoming articles 18
  publication page 16
  table of contents 17
magazines 1, 2
Member Digital Library
  accessing 13
  filing documents 55
  limit exceeded 60
  locating a filed document 22
  setting up account 3
Member Digital Library 3, 13, 55, 60

N

names
  formatting author 35
names 35
narrowing a search 38, 42
nesting search operations 40
not (operator) 28

O

operators
  and 28
  in 28, 40

NVIDIA Ex1023
Page O-75

not 28
or 28, 40
operators 33
or (operator) 28, 40
order of search terms 40
organizing search results 30
overview of IEEE Xplore 1

P

page viewer 54
password 11
PDF page viewer 54
periodicals
finding 24
forthcoming articles 18
publication page 16
table of contents 17
periodicals 1, 2
printing 14, 55
publication year
specifying 30
publication year 30
publications in IEEE Xplore
finding 24
searching within 16, 20, 27
publications in IEEE Xplore 1
purchasing a document 56

Q

Quick Find
finding publications 24
in IEEE Enterprise 23
Quick Find 24
quotation marks 38

R

record structure 2
reference list 51
refining a search 38, 41, 42
results
formatting search 45
view selected items 44
reusing previous searches 36
rights and permissions 53
Rightslink service 53

S

search examples
advanced 46
author 49
complex free-form 48
free-form 47
search examples 46, 47, 48, 49
search function denials 59
search results
formatting 45
organizing 30, 45
view selected items 44
search text 37
searching
advanced 28
basic 26
broadening 38, 42
collections 29
complex expressions 48
CrossRef Search Pilot 36
exact word or phrase 38
examples 46, 47, 48
fields 28
IEEE Enterprise filing cabinet 23
multiple criteria 40
narrowing 38, 42
nesting operations 40
operators 33
order of search terms 40
publications 16, 20, 27
refining 38, 41, 42
reusing previous searches 36
specifying the publication year 30
spelling variations with wildcard characters . 38
tips 37, 38, 40, 41, 42
searching 25
session history 36
special characters and symbols 37
spelling variations 38
standards
finding 24
publication page 22
standards 1, 2
structured advanced search
examples 46
structured advanced search 28
subscription packages 2
symbols
special 37
symbols 37

T

table of contents, conference proceedings 21
table of contents, forthcoming articles 18
table of contents, journals and magazines 17
tips for searching 37, 38, 40, 41, 42
Top 100 Documents 5

U

updates to IEEE Xplore content 4
uppercase letters 37
username 11

NVIDIA Ex1023
Page O-76

using field codes 32
using search operators....33

V

variations of spelling....38
variations of words 38
viewing Abstract or AbstractPlus denials 59
viewing Abstract or AbstractPlus 51
viewing field codes 32
viewing HTML full-text denials 58
viewing HTML full-text....54
viewing IEEE Xplore Home page....6, 7, 8
viewing PDF full text denials 58
viewing PDF full text....54
viewing search operators 33
viewing search results selected 44
viewing session history 36
viewing table of contents 17, 18, 21
viewing top 100 documents 5

W

wildcard characters....38
word (operator) 38
word variations....38
words, exact....38

Y

year specifying 30
year....30

Z

zooming a page image 54

ATTACHMENT P

The Wayback Machine - https://web.archive.org/web/20050329031255/http://ieeexplore.ieee.org:80/Xplore/guesthome.jsp

IEEE Xplore RELEASE 2.0



Welcome to IEEE Xplore

... delivering full text access to the world's highest quality technical literature in electrical engineering, computer science, and electronics.

Login

Username

Password

Forgot your password?

Browse

» Journals & Magazines

» Conference Proceedings

» Standards

Basic Search

Login For: » Advanced Search

» Author Search

» CrossRef Search

Content Updates

Browse the latest update to see recently added content.

» Latest Content Update

Alerts

Register and access your tables of contents alerts.

» Visit Alerts

Top 100 Documents

Find out the most accessed documents for the month.

» View Top 100

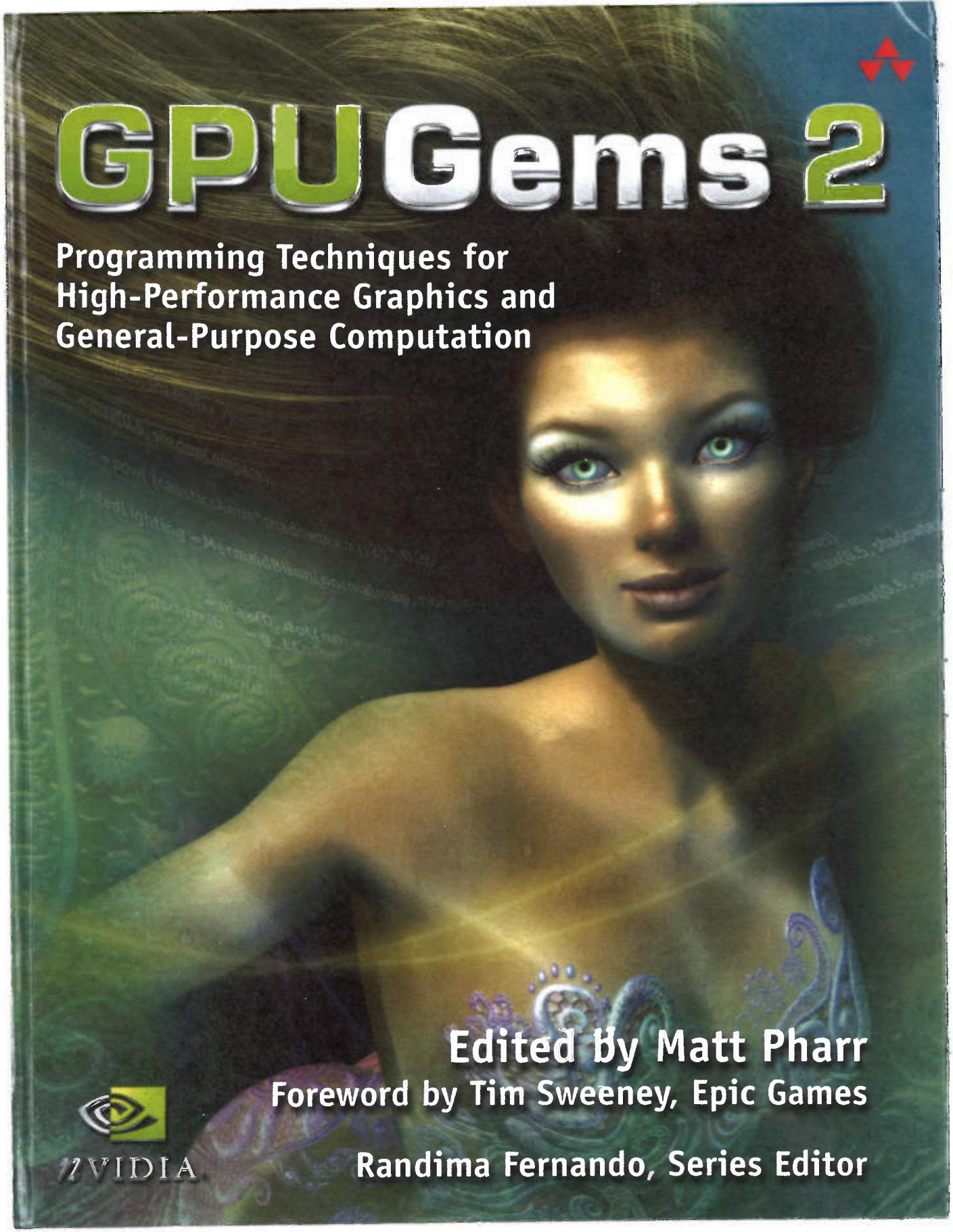
What's New @ IEEE
10 Free Newsletters
Subscribe Now

Cookies Disabled

Indexed by Inspec

© Copyright 2005 IEEE – All Rights Reserved

ATTACHMENT Q

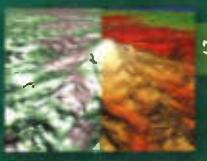

GPU Gems 2
Programming Techniques for
High-Performance Graphics and
General-Purpose Computation
Edited by Matt Pharr
Foreword by Tim Sweeney, Epic Games
Randima Fernando, Series Editor
NVIDIA

GEOMETRIC COMPLEXITY



7

Toward Photorealism in Virtual Botany
David Whatley
(Simutronics Corporation)



27

Terrain Rendering Using GPU-Based Geometry Clipmaps
Arul Asirvatham and Hugues Hoppe
(Microsoft Research)



47

Instance Geometry Instancing
Francesco Carucci
(Lionhead Studios)



69

Segment Buffering
Jon Olick
(2015)

SHADING, LIGHTING, AND SHADOWS



143

Deferred Shading in S.T.A.L.K.E.R.
Oles Shishkovtsov
(GSC Game World)



167

Real Time Computation of Dynamic Irradiance Environment Maps
Gary King
(NVIDIA)



177

Approximate Bidirectional Texture Functions
Jan Kautz
(MIT)



189

Tile-Based Texture Mapping
Li-Yi Wei
(NVIDIA)



269

Efficient Soft-Edged Shadows Using Pixel Shader Branching
Yury Uralsky
(NVIDIA)



283

Using Vertex Texture Displacement for Realistic Water Rendering
Yuri Kryachko
(1C: Maddox Games)



295

Geometric Refractive Simulation
Tiago Sousa
(Crytek)

HIGH-QUALITY RENDERING



313

Real-Time Higher-Order Texture Filtering
Christian Sigg *(ETH Zurich)*
and Markus Hadwiger
(VRVis Research Center)



393

GPU Image Processing in Apple's Motion
Pete Warden
(Apple Computer)



409

Implementing Improved Perlin Noise
Simon Green
(NVIDIA)



417

Advanced High-Quality Filtering
Justin Novosad
(discreet)



437

Mipmap-Level Measurement
Iain Cantlay
(Climax Entertainment)

IMAGE-ORIENTED COMPUTING



595

Octree Textures on the GPU
Sylvain Lefebvre, Samuel Hornus,
and Fabrice Neyret
(GRAVIR/IMAG – INRIA)



615

High Quality Global Illumination Rendering Using Rasterization
Toshiya Hachisuka
(The University of Tokyo)



635

Global Illumination Using Progressive Refinement Radiosity
Greg Coombe *(UNC at Chapel Hill)*
and Mark Harris *(NVIDIA)*



649

Computer Vision on the GPU
James Fung
(University of Toronto)



667

Deferred Filtering: Rendering from Difficult Data Formats
Joe Kniss *(Univ. of Utah)*, Aaron Lefohn,
and Nathaniel Fout *(UC Davis)*



677

Conservative Rasterization
Jon Hasselgren, Tomas
Akenine-Moller, and Lennart
Ohlsson *(Lund University)*

SIMULATION AND NUMERICAL ALGORITHMS



GPU Computing for Protein Structure Prediction
Paulius Micikevicius
(Armstrong Atlantic State University)



703

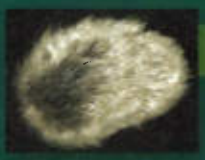
A GPU Framework for Solving Systems of Linear Equations
Jens Kruger and Rudiger Westermann
(Technische Universitat München)


75
Optimizing Resource Management with Multithreading
Oliver Hoeller and Kurt Pelzer
(Piranha Bytes)

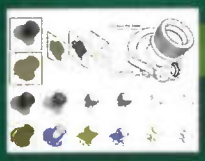

91
Hardware Occlusion Queries Made Useful
Michael Wimmer and Jiri Bittner
(Vienna University of Technology)


109
Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping
Michael Bunnell
(NVIDIA)


123
Per-Pixel Displacement Mapping with Distance Functions
William Donnelly
(University of Waterloo)


201
Implementing the mental images Phenomena Renderer on the GPU
Martin-Karl Lefrançois
(mental images)


223
Dynamic Ambient Occlusion and Indirect Lighting
Michael Bunnell
(NVIDIA)

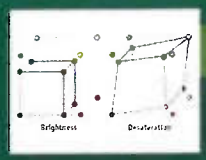

235
Blueprint Rendering and "Sketchy Drawings"
Marc Nienhaus and Jürgen Döllner
(University of Potsdam)

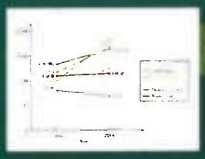

253
Accurate Atmospheric Scattering
Sean O'Neil

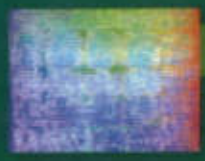

331
High-Quality Antialiased Rasterization
Dan Wexler and Eric Enderton
(NVIDIA)


345
Fast Prefiltered Lines
Eric Chan and Frédo Durand
(MIT)


361
Hair Animation and Rendering in the Nalu Demo
Hubert Nguyen and William Donnelly
(NVIDIA)

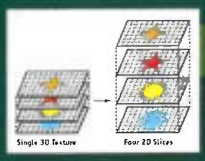

381
Using Lookup Tables to Accelerate Color Transformations
Jeremy Selan
(Sony Pictures Imageworks)


457
Streaming Architectures and Technology Trends
John Owens
(UC Davis)

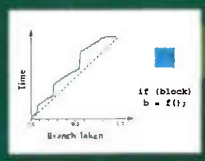

471
The GeForce 6 Series GPU Architecture
Emmett Kilgariff and Randima Fernando
(NVIDIA)


493
Mapping Computational Concepts to GPUs
Mark Harris
(NVIDIA)


509
Taking the Plunge into GPU Computing
Ian Buck
(Stanford University)

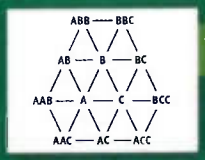


521
Implementing Efficient Parallel Data Structures on GPUs
Aaron Lefohn *(UC Davis)*, Joe Kniss *(Univ. of Utah)*, John Owens *(UC Davis)*

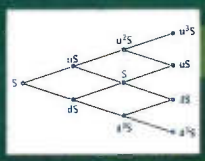


547
GPU Flow-Control Idioms
Mark Harris *(NVIDIA)* and Ian Buck *(Stanford University)*

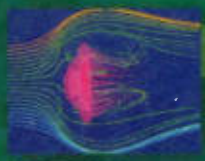

557
GPU Program Optimization
Cliff Woolley
(University of Virginia)

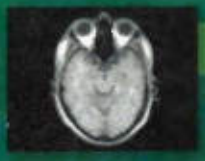


573
Stream Reduction Operations for GPGPU Applications
Daniel Horn
(Stanford University)


719
Options Pricing on the GPU
Craig Kolb and Matt Pharr
(NVIDIA)


733
Improved GPU Sorting
Peter Kipfer and Rüdiger Westermann
(Technische Universität München)


747
Flow Simulation with Complex Boundaries
Wei Li *(Siemens Corporate Research)*, Zhe Fan, Xiaoming Wei, and Arie Kaufman *(Stony Brook University)*


765
Medical Image Reconstruction with the FFT
Thilaka Sumanaweera and Donald Liu
(Siemens Medical Solutions USA)

GENERAL-PURPOSE COMPUTATION ON GPUS: A PRIMER

GPU Gems 2

GPU Gems 2

Programming Techniques for High-Performance Graphics and General-Purpose Computation

Edited by Matt Pharr

Randima Fernando, Series Editor

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

About the Cover: The Nalu character was created by the NVIDIA Demo Team to showcase the rendering power of the GeForce 6800 GPU. The demo shows off advanced hair shading and shadowing algorithms, as well as iridescence and bioluminescence. Soft shafts of light from the water surface are blocked by her body, and her skin is lit by the light refracted through the water's surface, with her body and hair casting soft shadows on her as she swims.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

NVIDIA makes no warranty or representation that the techniques described herein are free from any Intellectual Property claims. The reader assumes all risk of any such claims based on his or her use of these techniques.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales
international@pearsoned.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data
GPU gems 2 : programming techniques for high-performance graphics and general-purpose computation / edited by Matt Pharr ; Randima Fernando, series editor.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-33559-7 (hardcover : alk. paper)
1. Computer graphics. 2. Real-time programming. I. Pharr, Matt. II. Fernando, Randima.

T385.G688 2005
006.66—dc22

2004030181

GeForce™ and NVIDIA Quadro® are trademarks or registered trademarks of NVIDIA Corporation.
Nalu, Timbury, and Clear Sailing images © 2004 NVIDIA Corporation.
mental images and mental ray are trademarks or registered trademarks of mental images, GmbH.

Copyright © 2005 by NVIDIA Corporation.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
One Lake Street
Upper Saddle River, NJ 07458

ISBN 0-321-33559-7

Text printed in the United States on recycled paper at Quebecor World Taunton in Taunton, Massachusetts.

First printing, March 2005

To everyone striving to make
today's best computer graphics
look primitive tomorrow

NVIDIA Ex1023
Page Q-8

Contents

Foreword xxix
Preface xxxi
Contributors xxxv

PART I GEOMETRIC COMPLEXITY 1

Chapter 1
Toward Photorealism in Virtual Botany 7
David Whatley, Simutronics Corporation

1.1 Scene Management 7
1.1.1 The Planting Grid 8
1.1.2 Planting Strategy 9
1.1.3 Real-Time Optimization 10
1.2 The Grass Layer 11
1.2.1 Simulating Alpha Transparency via Dissolve 13
1.2.2 Variation 15
1.2.3 Lighting 15
1.2.4 Wind 17
1.3 The Ground Clutter Layer 17
1.4 The Tree and Shrub Layers 18
1.5 Shadowing 20
1.6 Post-Processing 22
1.6.1 Sky Dome Blooming 23
1.6.2 Full-Scene Glow 24
1.7 Conclusion 24
1.8 References 24

NVIDIA Ex1023
Page Q-9

Chapter 2
Terrain Rendering Using GPU-Based Geometry Clipmaps 27
Arul Asirvatham, Microsoft Research
Hugues Hoppe, Microsoft Research

2.1 Review of Geometry Clipmaps 27
2.2 Overview of GPU Implementation 30
2.2.1 Data Structures 31
2.2.2 Clipmap Size 31
2.3 Rendering 32
2.3.1 Active Levels 32
2.3.2 Vertex and Index Buffers 33
2.3.3 View Frustum Culling 35
2.3.4 `DrawPrimitive` Calls 35
2.3.5 The Vertex Shader 36
2.3.6 The Pixel Shader 38
2.4 Update 39
2.4.1 Upsampling 40
2.4.2 Residuals 42
2.4.3 Normal Map 42
2.5 Results and Discussion 43
2.6 Summary and Improvements 43
2.6.1 Vertex Textures 44
2.6.2 Eliminating Normal Maps 44
2.6.3 Memory-Free Terrain Synthesis 44
2.7 References 44

Chapter 3
Inside Geometry Instancing 47
Francesco Carucci, Lionhead Studios

3.1 Why Geometry Instancing? 48
3.2 Definitions 49
3.2.1 Geometry Packet 49
3.2.2 Instance Attributes 49
3.2.3 Geometry Instance 50
3.2.4 Render and Texture Context 50
3.2.5 Geometry Batch 50

NVIDIA Ex1023
Page Q-10

3.3 Implementation . . . 53
3.3.1 Static Batching . . . 54
3.3.2 Dynamic Batching . . . 56
3.3.3 Vertex Constants Instancing . . . 57
3.3.4 Batching with the Geometry Instancing API . . . 61
3.4 Conclusion . . . 65
3.5 References . . . 67

Chapter 4
Segment Buffering . . . 69
Jon Olick, 2015

4.1 The Problem Space . . . 69
4.2 The Solution . . . 70
4.3 The Method . . . 71
4.3.1 Segment Buffering, Step 1 . . . 71
4.3.2 Segment Buffering, Step 2 . . . 71
4.3.3 Segment Buffering, Step 3 . . . 72
4.4 Improving the Technique . . . 72
4.5 Conclusion . . . 72
4.6 References . . . 73

Chapter 5
Optimizing Resource Management with Multistreaming . . . 75
Oliver Hoeller, Piranha Bytes
Kurt Pelzer, Piranha Bytes

5.1 Overview . . . 76
5.2 Implementation . . . 77
5.2.1 Multistreaming with DirectX 9.0 . . . 78
5.2.2 Resource Management . . . 81
5.2.3 Processing Vertices . . . 83
5.3 Conclusion . . . 89
5.4 References . . . 90

Chapter 6
Hardware Occlusion Queries Made Useful . . . 91
Michael Wimmer, Vienna University of Technology
Jiří Bittner, Vienna University of Technology

6.1 Introduction . . . 91
6.2 For Which Scenes Are Occlusion Queries Effective? . . . 92

NVIDIA Ex1023
Page Q-11

6.3 What Is Occlusion Culling? . . . 93
6.4 Hierarchical Stop-and-Wait Method . . . 94
6.4.1 The Naive Algorithm, or Why Use Hierarchies at All? . . . 94
6.4.2 Hierarchies to the Rescue! . . . 95
6.4.3 Hierarchical Algorithm . . . 95
6.4.4 Problem 1: Stalls . . . 96
6.4.5 Problem 2: Query Overhead . . . 97
6.5 Coherent Hierarchical Culling . . . 97
6.5.1 Idea 1: Being Smart and Guessing . . . 97
6.5.2 Idea 2: Pull Up, Pull Up . . . 98
6.5.3 Algorithm . . . 99
6.5.4 Implementation Details . . . 100
6.5.5 Why Are There Fewer Stalls? . . . 103
6.5.6 Why Are There Fewer Queries? . . . 104
6.5.7 How to Traverse the Hierarchy . . . 104
6.6 Optimizations . . . 105
6.6.1 Querying with Actual Geometry . . . 105
6.6.2 Z-Only Rendering Pass . . . 105
6.6.3 Approximate Visibility . . . 105
6.6.4 Conservative Visibility Testing . . . 106
6.7 Conclusion . . . 106
6.8 References . . . 108

Chapter 7
Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping . . . 109
Michael Bunnell, NVIDIA Corporation

7.1 Subdivision Surfaces . . . 109
7.1.1 Some Definitions . . . 110
7.1.2 Catmull-Clark Subdivision . . . 110
7.1.3 Using Subdivision for Tessellation . . . 111
7.1.4 Patching the Surface . . . 112
7.1.5 The GPU Tessellation Algorithm . . . 114
7.1.6 Watertight Tessellation . . . 118
7.2 Displacement Mapping . . . 119
7.2.1 Changing the Flatness Test . . . 120
7.2.2 Shading Using Normal Mapping . . . 120
7.3 Conclusion . . . 122
7.4 References . . . 122

NVIDIA Ex1023
Page Q-12

Chapter 8
Per-Pixel Displacement Mapping with Distance Functions 123
William Donnelly, University of Waterloo

8.1 Introduction 123
8.2 Previous Work 125
8.3 The Distance-Mapping Algorithm 126
8.3.1 Arbitrary Meshes 129
8.4 Computing the Distance Map 130
8.5 The Shaders 130
8.5.1 The Vertex Shader 130
8.5.2 The Fragment Shader 130
8.5.3 A Note on Filtering 132
8.6 Results 132
8.7 Conclusion 134
8.8 References 135

PART II SHADING, LIGHTING, AND SHADOWS 137

Chapter 9
Deferred Shading in *S.T.A.L.K.E.R.* 143
Oles Shishkovtsov, GSC Game World

9.1 Introduction 143
9.2 The Myths 145
9.3 Optimizations 147
9.3.1 What to Optimize 147
9.3.2 Lighting Optimizations 148
9.3.3 G-Buffer-Creation Optimizations 151
9.3.4 Shadowing Optimizations 153
9.4 Improving Quality 154
9.4.1 The Power of "Virtual Position" 155
9.4.2 Ambient Occlusion 156
9.4.3 Materials and Surface-Light Interaction 157
9.5 Antialiasing 158
9.5.1 Efficient Tone Mapping 161
9.5.2 Dealing with Transparency 162
9.6 Things We Tried but Did Not Include in the Final Code 162
9.6.1 Elevation Maps 163
9.6.2 Real-Time Global Illumination 163
9.7 Conclusion 164
9.8 References 165

NVIDIA Ex1023
Page Q-13

Chapter 10
Real-Time Computation of Dynamic Irradiance Environment Maps 167
Gary King, NVIDIA Corporation

10.1 Irradiance Environment Maps 167
10.2 Spherical Harmonic Convolution 170
10.3 Mapping to the GPU 172
10.3.1 Spatial to Frequency Domain 172
10.3.2 Convolution and Back Again 173
10.4 Further Work 175
10.5 Conclusion 176
10.6 References 176

Chapter 11
Approximate Bidirectional Texture Functions 177
Jan Kautz, Massachusetts Institute of Technology

11.1 Introduction 177
11.2 Acquisition 179
11.2.1 Setup and Acquisition 179
11.2.2 Assembling the Shading Map 179
11.3 Rendering 181
11.3.1 Detailed Algorithm 181
11.3.2 Real-Time Rendering 182
11.4 Results 184
11.4.1 Discussion 186
11.5 Conclusion 187
11.6 References 187

Chapter 12
Tile-Based Texture Mapping 189
Li-Yi Wei, NVIDIA Corporation

12.1 Our Approach 191
12.2 Texture Tile Construction 191
12.3 Texture Tile Packing 192
12.4 Texture Tile Mapping 195
12.5 Mipmap Issues 197
12.6 Conclusion 198
12.7 References 199

NVIDIA Ex1023
Page Q-14

Chapter 13
Implementing the mental images Phenomena Renderer on the GPU 201
Martin-Karl Lefrançois, mental images

13.1 Introduction 201
13.2 Shaders and Phenomena 202
13.3 Implementing Phenomena Using Cg 205
13.3.1 The Cg Vertex Program and the Varying Parameters 205
13.3.2 The `main()` Entry Point for Fragment Shaders 207
13.3.3 The General Shader Interfaces 207
13.3.4 Example of a Simple Shader 208
13.3.5 Global State Variables 211
13.3.6 Light Shaders 211
13.3.7 Texture Shaders 215
13.3.8 Bump Mapping 216
13.3.9 Environment and Volume Shaders 217
13.3.10 Shaders Returning Structures 218
13.3.11 Rendering Hair 220
13.3.12 Putting It All Together 220
13.4 Conclusion 221
13.5 References 222

Chapter 14
Dynamic Ambient Occlusion and Indirect Lighting 223
Michael Bunnell, NVIDIA Corporation

14.1 Surface Elements 223
14.2 Ambient Occlusion 225
14.2.1 The Multipass Shadowing Algorithm 226
14.2.2 Improving Performance 228
14.3 Indirect Lighting and Area Lights 231
14.4 Conclusion 232
14.5 References 233

NVIDIA Ex1023
Page Q-15

Chapter 15
Blueprint Rendering and "Sketchy Drawings" 235

Marc Nienhaus, University of Potsdam, Hasso-Plattner-Institute
Jürgen Döllner, University of Potsdam, Hasso-Plattner-Institute

15.1 Basic Principles 236
15.1.1 Intermediate Rendering Results 236
15.1.2 Edge Enhancement 236
15.1.3 Depth Sprite Rendering 237
15.2 Blueprint Rendering 238
15.2.1 Depth Peeling 238
15.2.2 Extracting Visible and Nonvisible Edges 241
15.2.3 Composing Blueprints 241
15.2.4 Depth Masking 242
15.2.5 Visualizing Architecture Using Blueprint Rendering 244
15.3 Sketchy Rendering 244
15.3.1 Edges and Color Patches 245
15.3.2 Applying Uncertainty 245
15.3.3 Adjusting Depth 247
15.3.4 Variations of Sketchy Drawing 247
15.3.5 Controlling Uncertainty 248
15.3.6 Reducing the Shower-Door Effect 250
15.4 Conclusion 251
15.5 References 252

Chapter 16
Accurate Atmospheric Scattering 253

Sean O'Neil

16.1 Introduction 253
16.2 Solving the Scattering Equations 254
16.2.1 Rayleigh Scattering vs. Mie Scattering 255
16.2.2 The Phase Function 256
16.2.3 The Out-Scattering Equation 256
16.2.4 The In-Scattering Equation 257
16.2.5 The Surface-Scattering Equation 257
16.3 Making It Real-Time 258
16.4 Squeezing It into a Shader 260
16.4.1 Eliminating One Dimension 260
16.4.2 Eliminating the Other Dimension 261

NVIDIA Ex1023
Page Q-16

16.5 Implementing the Scattering Shaders 262
16.5.1 The Vertex Shader 262
16.5.2 The Fragment Shader 264
16.6 Adding High-Dynamic-Range Rendering 265
16.7 Conclusion 266
16.8 References 267

Chapter 17
Efficient Soft-Edged Shadows Using Pixel Shader Branching 269
Yury Uralsky, NVIDIA Corporation

17.1 Current Shadowing Techniques 270
17.2 Soft Shadows with a Single Shadow Map 271
17.2.1 Blurring Hard-Edged Shadows 271
17.2.2 Improving Efficiency 274
17.2.3 Implementation Details 277
17.3 Conclusion 281
17.4 References 282

Chapter 18
Using Vertex Texture Displacement for Realistic Water Rendering 283
Yuri Kryachko, 1C:Maddox Games

18.1 Water Models 283
18.2 Implementation 284
18.2.1 Water Surface Model 284
18.2.2 Implementation Details 285
18.2.3 Sampling Height Maps 286
18.2.4 Quality Improvements and Optimizations 288
18.2.5 Rendering Local Perturbations 292
18.3 Conclusion 294
18.4 References 294

Chapter 19
Generic Refraction Simulation 295
Tiago Sousa, Crytek

19.1 Basic Technique 296
19.2 Refraction Mask 297

NVIDIA Ex1023
Page Q-17

19.3 Examples 300
19.3.1 Water Simulation 300
19.3.2 Glass Simulation 303
19.4 Conclusion 305
19.5 References 305

PART III HIGH-QUALITY RENDERING 307

Chapter 20
Fast Third-Order Texture Filtering 313
Christian Sigg, ETH Zurich
Markus Hadwiger, VRVis Research Center

20.1 Higher-Order Filtering 314
20.2 Fast Recursive Cubic Convolution 315
20.3 Mipmapping 320
20.4 Derivative Reconstruction 324
20.5 Conclusion 327
20.6 References 328

Chapter 21
High-Quality Antialiased Rasterization 331
Dan Wexler, NVIDIA Corporation
Eric Enderton, NVIDIA Corporation

21.1 Overview 331
21.2 Downsampling 334
21.2.1 Comparison to Existing Hardware and Software 334
21.2.2 Downsampling on the GPU 336
21.3 Padding 336
21.4 Filter Details 337
21.5 Two-Pass Separable Filtering 338
21.6 Tiling and Accumulation 339
21.7 The Code 339
21.7.1 The Rendering Loop 340
21.7.2 The Downsample Class 341
21.7.3 Implementation Details 343
21.8 Conclusion 344
21.9 References 344

NVIDIA Ex1023
Page Q-18

Chapter 22
Fast Prefiltered Lines 345
Eric Chan, Massachusetts Institute of Technology
Frédo Durand, Massachusetts Institute of Technology

22.1 Why Sharp Lines Look Bad. 345
22.2 Bandlimiting the Signal. 347
22.2.1 Prefiltering. 347
22.3 The Preprocess 349
22.4 Runtime 351
22.4.1 Line Setup (CPU) 352
22.4.2 Table Lookups (GPU) 353
22.5 Implementation Issues. 355
22.5.1 Drawing Fat Lines 355
22.5.2 Compositing Multiple Lines 355
22.6 Examples. 356
22.7 Conclusion 358
22.8 References 359

Chapter 23
Hair Animation and Rendering in the Nalu Demo. 361
Hubert Nguyen, NVIDIA Corporation
William Donnelly, NVIDIA Corporation

23.1 Hair Geometry 362
23.1.1 Layout and Growth. 362
23.1.2 Controlling the Hair 362
23.1.3 Data Flow 364
23.1.4 Tessellation 364
23.1.5 Interpolation. 364
23.2 Dynamics and Collisions 366
23.2.1 Constraints 366
23.2.2 Collisions. 367
23.2.3 Fins. 368
23.3 Hair Shading. 369
23.3.1 A Real-Time Reflectance Model for Hair. 369
23.3.2 Real-Time Volumetric Shadows in Hair. 375
23.4 Conclusion and Future Work 378
23.5 References. 380

NVIDIA Ex1023
Page Q-19

Chapter 24
Using Lookup Tables to Accelerate Color Transformations 381
Jeremy Selan, Sony Pictures Imageworks

24.1 Lookup Table Basics 381
24.1.1 One-Dimensional LUTs 382
24.1.2 Three-Dimensional LUTs 383
24.1.3 Interpolation 385
24.2 Implementation 386
24.2.1 Strategy for Mapping LUTs to the GPU 386
24.2.2 Cg Shader 386
24.2.3 System Integration 389
24.2.4 Extending 3D LUTs for Use with High-Dynamic-Range Imagery 390
24.3 Conclusion 392
24.4 References 392

Chapter 25
GPU Image Processing in Apple's Motion 393
Pete Warden, Apple Computer

25.1 Design 393
25.1.1 Loves and Loathings 394
25.1.2 Pick a Language 396
25.1.3 CPU Fallback 396
25.2 Implementation 397
25.2.1 GPU Resource Limits 397
25.2.2 Division by Zero 399
25.2.3 Loss of Vertex Components 400
25.2.4 Bilinear Filtering 400
25.2.5 High-Precision Storage 405
25.3 Debugging 406
25.4 Conclusion 407
25.5 References 408

Chapter 26
Implementing Improved Perlin Noise 409
Simon Green, NVIDIA Corporation

26.1 Random but Smooth 409
26.2 Storage vs. Computation 410

NVIDIA Ex1023
Page Q-20

26.3 Implementation Details 411
26.3.1 Optimization 415
26.4 Conclusion 415
26.5 References 416

Chapter 27
Advanced High-Quality Filtering 417
Justin Novosad, discreet

27.1 Implementing Filters on GPUs 417
27.1.1 Accessing Image Samples 418
27.1.2 Convolution Filters 419
27.2 The Problem of Digital Image Resampling 422
27.2.1 Background 423
27.2.2 Antialiasing 423
27.2.3 Image Reconstruction 427
27.3 Shock Filtering: A Method for Deblurring Images 430
27.4 Filter Implementation Tips 433
27.5 Advanced Applications 433
27.5.1 Time Warping 433
27.5.2 Motion Blur Removal 434
27.5.3 Adaptive Texture Filtering 434
27.6 Conclusion 434
27.7 References 435

Chapter 28
Mipmap-Level Measurement 437
Iain Cantlay, Climax Entertainment

28.1 Which Mipmap Level Is Visible? 438
28.2 GPU to the Rescue 439
28.2.1 Counting Pixels 439
28.2.2 Practical Considerations in an Engine 442
28.2.3 Extensions 445
28.3 Sample Results 447
28.4 Conclusion 448
28.5 References 449

NVIDIA Ex1023
Page Q-21

PART IV GENERAL-PURPOSE COMPUTATION ON GPUS: A PRIMER 451

Chapter 29
Streaming Architectures and Technology Trends 457
John Owens, University of California, Davis

29.1 Technology Trends 457
29.1.1 Core Technology Trends 458
29.1.2 Consequences 458
29.2 Keys to High-Performance Computing 461
29.2.1 Methods for Efficient Computation 461
29.2.2 Methods for Efficient Communication 462
29.2.3 Contrast to CPUs 463
29.3 Stream Computation 464
29.3.1 The Stream Programming Model 464
29.3.2 Building a Stream Processor 466
29.4 The Future and Challenges 468
29.4.1 Challenge: Technology Trends 468
29.4.2 Challenge: Power Management 468
29.4.3 Challenge: Supporting More Programmability and Functionality 469
29.4.4 Challenge: GPU Functionality Subsumed by CPU (or Vice Versa)? 470
29.5 References 470

Chapter 30
The GeForce 6 Series GPU Architecture 471
Emmett Kilgariff, NVIDIA Corporation
Randima Fernando, NVIDIA Corporation

30.1 How the GPU Fits into the Overall Computer System 471
30.2 Overall System Architecture 473
30.2.1 Functional Block Diagram for Graphics Operations 473
30.2.2 Functional Block Diagram for Non-Graphics Operations . . 478
30.3 GPU Features 481
30.3.1 Fixed-Function Features. 481
30.3.2 Shader Model 3.0 Programming Model. 483
30.3.3 Supported Data Storage Formats. 488
30.4 Performance 488

NVIDIA Ex1023
Page Q-22

30.5 Achieving Optimal Performance 490
30.5.1 Use Z-Culling Aggressively 490
30.5.2 Exploit Texture Math When Loading Data 490
30.5.3 Use Branching in Fragment Programs Judiciously 490
30.5.4 Use fp16 Intermediate Values Wherever Possible 491
30.6 Conclusion 491

Chapter 31
Mapping Computational Concepts to GPUs 493
Mark Harris, NVIDIA Corporation

31.1 The Importance of Data Parallelism 493
31.1.1 What Kinds of Computation Map Well to GPUs? 494
31.1.2 Example: Simulation on a Grid 495
31.1.3 Stream Communication: Gather vs. Scatter 496
31.2 An Inventory of GPU Computational Resources 497
31.2.1 Programmable Parallel Processors 497
31.3 CPU-GPU Analogies 500
31.3.1 Streams: GPU Textures = CPU Arrays 500
31.3.2 Kernels: GPU Fragment Programs = CPU "Inner Loops" . . 500
31.3.3 Render-to-Texture = Feedback 501
31.3.4 Geometry Rasterization = Computation Invocation 501
31.3.5 Texture Coordinates = Computational Domain 501
31.3.6 Vertex Coordinates = Computational Range 502
31.3.7 Reductions 502
31.4 From Analogies to Implementation 503
31.4.1 Putting It All Together: A Basic GPGPU Framework 503
31.5 A Simple Example 505
31.6 Conclusion 508
31.7 References 508

Chapter 32
Taking the Plunge into GPU Computing 509
Ian Buck, Stanford University

32.1 Choosing a Fast Algorithm 509
32.1.1 Locality, Locality, Locality 510
32.1.2 Letting Computation Rule 511
32.1.3 Considering Download and Readback 512
32.2 Understanding Floating Point 513
32.2.1 Address Calculation 514

NVIDIA Ex1023
Page Q-23

32.3 Implementing Scatter 515
32.3.1 Converting to Gather 515
32.3.2 Address Sorting 516
32.3.3 Rendering Points 518
32.4 Conclusion 518
32.5 References 519

Chapter 33
Implementing Efficient Parallel Data Structures on GPUs 521
Aaron Lefohn, University of California, Davis
Joe Kniss, University of Utah
John Owens, University of California, Davis

33.1 Programming with Streams 521
33.2 The GPU Memory Model 524
33.2.1 Memory Hierarchy 524
33.2.2 GPU Stream Types 525
33.2.3 GPU Kernel Memory Access 527
33.3 GPU-Based Data Structures 528
33.3.1 Multidimensional Arrays 528
33.3.2 Structures 534
33.3.3 Sparse Data Structures 535
33.4 Performance Considerations 540
33.4.1 Dependent Texture Reads 540
33.4.2 Computational Frequency 541
33.4.3 Pbuffer Survival Guide 541
33.5 Conclusion 543
33.6 References 544

Chapter 34
GPU Flow-Control Idioms 547
Mark Harris, NVIDIA Corporation
Ian Buck, Stanford University

34.1 Flow-Control Challenges 547
34.2 Basic Flow-Control Strategies 549
34.2.1 Predication 549
34.2.2 Moving Branching up the Pipeline 549
34.2.3 Z-Cull 550
34.2.4 Branching Instructions 553
34.2.5 Choosing a Branching Mechanism 553
34.3 Data-Dependent Looping with Occlusion Queries 554
34.4 Conclusion 555

NVIDIA Ex1023
Page Q-24

Chapter 35
GPU Program Optimization. 557
Cliff Woolley, University of Virginia

35.1 Data-Parallel Computing. 557
35.1.1 Instruction-Level Parallelism 558
35.1.2 Data-Level Parallelism 560
35.2 Computational Frequency 561
35.2.1 Precomputation of Loop Invariants 563
35.2.2 Precomputation Using Lookup Tables 564
35.2.3 Avoid Inner-Loop Branching 566
35.2.4 The Swizzle Operator 566
35.3 Profiling and Load Balancing 568
35.4 Conclusion 570
35.5 References 570

Chapter 36
Stream Reduction Operations for GPGPU Applications 573
Daniel Horn, Stanford University

36.1 Filtering Through Compaction 574
36.1.1 Running Sum Scan 574
36.1.2 Scatter Through Search/Gather 575
36.1.3 Filtering Performance. 579
36.2 Motivation: Collision Detection 579
36.3 Filtering for Subdivision Surfaces 583
36.3.1 Subdivision on Streaming Architectures. 584
36.4 Conclusion 587
36.5 References 587

PART V IMAGE-ORIENTED COMPUTING 591

Chapter 37
Octree Textures on the GPU 595
Sylvain Lefebvre, GRAVIR/IMAG – INRIA
Samuel Hornus, GRAVIR/IMAG – INRIA
Fabrice Neyret, GRAVIR/IMAG – INRIA

37.1 A GPU-Accelerated Hierarchical Structure: The N^3-Tree. 597
37.1.1 Definition 597
37.1.2 Implementation. 598

NVIDIA Ex1023
Page Q-25

37.2 Application 1: Painting on Meshes 602
37.2.1 Creating the Octree 603
37.2.2 Painting 604
37.2.3 Rendering 604
37.2.4 Converting the Octree Texture to a Standard 2D Texture . . 607
37.3 Application 2: Surface Simulation 611
37.4 Conclusion 612
37.5 References 613

Chapter 38
High-Quality Global Illumination Rendering Using Rasterization 615
Toshiya Hachisuka, The University of Tokyo

38.1 Global Illumination via Rasterization 616
38.2 Overview of Final Gathering 617
38.2.1 Two-Pass Methods 617
38.2.2 Final Gathering 618
38.2.3 Problems with Two-Pass Methods 619
38.3 Final Gathering via Rasterization 621
38.3.1 Clustering of Final Gathering Rays 621
38.3.2 Ray Casting as Multiple Parallel Projection 623
38.4 Implementation Details 625
38.4.1 Initialization 625
38.4.2 Depth Peeling 626
38.4.3 Sampling 627
38.4.4 Performance 627
38.5 A Global Illumination Renderer on the GPU 627
38.5.1 The First Pass 628
38.5.2 Generating Visible Points Data 628
38.5.3 The Second Pass 629
38.5.4 Additional Solutions 629
38.6 Conclusion 632
38.7 References 632

Chapter 39
Global Illumination Using Progressive Refinement Radiosity 635
Greg Coombe, University of North Carolina at Chapel Hill
Mark Harris, NVIDIA Corporation

39.1 Radiosity Foundations 636
39.1.1 Progressive Refinement 637

NVIDIA Ex1023
Page Q-26

39.2 GPU Implementation 638
39.2.1 Visibility Using Hemispherical Projection 639
39.2.2 Form Factor Computation 641
39.2.3 Choosing the Next Shooter 643
39.3 Adaptive Subdivision 643
39.3.1 Texture Quadtree 644
39.3.2 Quadtree Subdivision 644
39.4 Performance 645
39.5 Conclusion 645
39.6 References 647

Chapter 40
Computer Vision on the GPU 649
James Fung, University of Toronto

40.1 Introduction 649
40.2 Implementation Framework 650
40.3 Application Examples 651
40.3.1 Using Sequences of Fragment Programs for Computer Vision 651
40.3.2 Summation Operations 655
40.3.3 Systems of Equations for Creating Image Panoramas 658
40.3.4 Feature Vector Computations 661
40.4 Parallel Computer Vision Processing 664
40.5 Conclusion 664
40.6 References 665

Chapter 41
Deferred Filtering: Rendering from Difficult Data Formats 667
Joe Kniss, University of Utah
Aaron Lefohn, University of California, Davis
Nathaniel Fout, University of California, Davis

41.1 Introduction 667
41.2 Why Defer? 668
41.3 Deferred Filtering Algorithm 669
41.4 Why It Works 673
41.5 Conclusions: When to Defer 673
41.6 References 674

NVIDIA Ex1023
Page Q-27

Chapter 42
Conservative Rasterization 677
Jon Hasselgren, Lund University
Tomas Akenine-Möller, Lund University
Lennart Ohlsson, Lund University

42.1 Problem Definition 678
42.2 Two Conservative Algorithms 679
42.2.1 Clip Space 681
42.2.2 The First Algorithm 681
42.2.3 The Second Algorithm 683
42.3 Robustness Issues 686
42.4 Conservative Depth 687
42.5 Results and Conclusions 689
42.6 References 690

PART VI SIMULATION AND NUMERICAL ALGORITHMS 691

Chapter 43
GPU Computing for Protein Structure Prediction 695
Paulius Micikevicius, Armstrong Atlantic State University

43.1 Introduction 695
43.2 The Floyd-Warshall Algorithm and Distance-Bound Smoothing 697
43.3 GPU Implementation 698
43.3.1 Dynamic Updates 698
43.3.2 Indexing Data Textures 698
43.3.3 The Triangle Approach 699
43.3.4 Vectorization 699
43.4 Experimental Results 701
43.5 Conclusions and Further Work 701
43.6 References 702

Chapter 44
A GPU Framework for Solving Systems of Linear Equations 703
Jens Krüger, Technische Universität München
Rüdiger Westermann, Technische Universität München

44.1 Overview 703
44.2 Representation 704
44.2.1 The "Single Float" Representation 704
44.2.2 Vectors 704
44.2.3 Matrices 706

NVIDIA Ex1023
Page Q-28

44.3 Operations 708
44.3.1 Vector Arithmetic 709
44.3.2 Vector Reduce 709
44.3.3 Matrix-Vector Product 710
44.3.4 Putting It All Together 712
44.3.5 Conjugate Gradient Solver 713
44.4 A Sample Partial Differential Equation 714
44.4.1 The Crank-Nicholson Scheme 716
44.5 Conclusion 718
44.6 References 718

Chapter 45
Options Pricing on the GPU 719
Craig Kolb, NVIDIA Corporation
Matt Pharr, NVIDIA Corporation

45.1 What Are Options? 719
45.2 The Black-Scholes Model 721
45.3 Lattice Models 725
45.3.1 The Binomial Model 725
45.3.2 Pricing European Options 726
45.4 Conclusion 730
45.5 References 731

Chapter 46
Improved GPU Sorting 733
Peter Kipfer, Technische Universität München
Rüdiger Westermann, Technische Universität München

46.1 Sorting Algorithms 733
46.2 A Simple First Approach 734
46.3 Fast Sorting 735
46.3.1 Implementing Odd-Even Merge Sort 737
46.4 Using All GPU Resources 738
46.4.1 Implementing Bitonic Merge Sort 743
46.5 Conclusion 745
46.6 References 746

NVIDIA Ex1023
Page Q-29

Chapter 47
Flow Simulation with Complex Boundaries 747
Wei Li, Siemens Corporate Research
Zhe Fan, Stony Brook University
Xiaoming Wei, Stony Brook University
Arie Kaufman, Stony Brook University

47.1 Introduction 747
47.2 The Lattice Boltzmann Method 748
47.3 GPU-Based LBM 749
47.3.1 Algorithm Overview 749
47.3.2 Packing 751
47.3.3 Streaming 752
47.4 GPU-Based Boundary Handling 753
47.4.1 GPU-Based Voxelization 754
47.4.2 Periodic Boundaries 756
47.4.3 Outflow Boundaries 756
47.4.4 Obstacle Boundaries 757
47.5 Visualization 759
47.6 Experimental Results 760
47.7 Conclusion 761
47.8 References 763

Chapter 48
Medical Image Reconstruction with the FFT 765
Thilaka Sumanaweera, Siemens Medical Solutions USA
Donald Liu, Siemens Medical Solutions USA

48.1 Background 765
48.2 The Fourier Transform 766
48.3 The FFT Algorithm 767
48.4 Implementation on the GPU 768
48.4.1 Approach 1: Mostly Loading the Fragment Processor 770
48.4.2 Approach 2: Loading the Vertex Processor, the Rasterizer, and the Fragment Processor 772
48.4.3 Load Balancing 775
48.4.4 Benchmarking Results 775
48.5 The FFT in Medical Imaging 776
48.5.1 Magnetic Resonance Imaging 776
48.5.2 Results in MRI 778
48.5.3 Ultrasonic Imaging 780
48.6 Conclusion 783
48.7 References 784

Index 785

NVIDIA Ex1023
Page Q-30

Foreword

Before the advent of dedicated PC graphics hardware, the industry's first 3D games used CPU-based software rendering. I wrote the first Unreal Engine in that era, inspired by John Carmack's pioneering programming work on *Doom* and *Quake*. Despite slow CPUs and low resolutions, the mid-1990s became a watershed time for graphics and gaming. New visual effects appeared almost monthly, marked by milestones like *Quake*'s light mapping and shadowing and Unreal's colored lighting and volumetric fog. That era faded away as fixed-function 3D accelerators appeared. Deprived of the programmability that drove innovation and differentiation, 3D games grew indistinct.

Today, a new Renaissance in 3D graphics is under way, driven by fully programmable GPUs—*graphics processing units*—that deliver thousands of times the graphics power available just ten years ago. Combining incredible parallel computing power with modern, high-level programming languages, today's GPUs have unleashed a Cambrian Explosion of innovation and creativity. Real-time soft shadowing, accurate lighting models, and realistic material interactions are readily achievable. But the most important gain of programmability is that you can do *anything* with a GPU so long as you can find an algorithm to express your idea. *GPU Gems 2* demonstrates many such ideas-turned-algorithms.

Let us take a moment to review the set of resources available to today's graphics programmer. First, you have access to a GPU that can perform tens of billions of floating-point calculations per second in programmable shading algorithms. It's your workhorse; if you can move your problem into the realm of pixels and vertices, then you can harness the GPU's immense power. Second, you have a CPU, the system's general-purpose computing engine. The CPU sends commands to the graphics processing unit, manages resources, and interacts with the outside world. Finally, you have access to artistic content—texture maps, meshes, and other multimedia data that the GPU can combine, filter, and procedurally modify during rendering.

The Gems in this book employ these resources in novel ways to render realistic scenes, process images, and produce special effects. In doing so, many of the previous era's

NVIDIA Ex1023
Page Q-31



graphics rules may be broken. GPUs are fast and flexible enough that you may render a given object many times, decomposing a scene into its components—lighting, shadowing, reflections, post-processing effects, and so on. You can employ the GPU for decidedly non-graphics tasks like collision detection, physics, and numerical computation; and within texture maps you can encode arbitrary data, such as vectors, positions, or lookup tables used by shader programs. And while visual realism is now achievable on GPUs, it is not your only option: nonphotorealistic rendering techniques are available, such as cel shading, exaggerated motion blur and light blooms, and other effects seen frequently in Hollywood productions.

Seven years after I wrote Unreal's original software renderer, my company began developing a new game engine, Unreal Engine 3, designed for the capabilities of today's modern GPUs. It has been an incredible experience! Where we once built 300-polygon scenes with static lighting and texture maps, we now combine dynamic per-pixel lighting and shadowing with realistic material effects in million-polygon scenes. We've seen an explosive growth in the power and flexibility available to programmers and artists alike. But while much has changed in graphics development, several truths have remained: that graphics requires a unique combination of engineering, artistry, and invention unmatched in other fields; that innovation moves at an incredible pace as hardware performance increases exponentially; and that graphics programming is *a heck of a lot of fun!*



Here in *GPU Gems 2*, you'll find a wealth of knowledge and insight, plus many just plain neat ideas, which can be readily applied on today's graphics hardware. But the techniques here are only a starting point on your adventure—the real fun and opportunity lie in finding new ways to customize and combine these Gems and to invent new ones.

Tim Sweeney
Founder and Technical Director, Epic Games

Screenshots from Unreal Engine 3 Technology Demo, http://www.unrealtechnology.com

NVIDIA Ex1023
Page Q-32

Preface

The first volume of *GPU Gems* was conceived in the spring of 2003, soon after the arrival of the first generation of fully programmable GPUs. The resulting book was released less than a year later and quickly became a best seller, providing a snapshot of the best ideas for making the most of the capabilities of the latest programmable graphics hardware.

GPU programming is a rapidly changing field, and the time is already ripe for a sequel. In the handful of years since programmable graphics processors first became available, they have become faster and more flexible at an incredible pace. Early programmable GPUs supported programmability only at the vertex level, while today complex per-pixel programs are common. A year ago, real-time GPU programs were typically tens of instructions long, while this year's GPUs handle complex programs hundreds of instructions long and still render at interactive rates. Programmable graphics has even transcended the PC and is rapidly spreading to consoles, handheld gaming devices, and mobile phones.

Until recently, performance-conscious developers might have considered writing their GPU programs in assembly language. These days, however, high-level GPU programming languages are ubiquitous. It is extremely rare for developers to bother writing assembly for GPUs anymore, thanks both to improvements in compilers and to the rapidly increasing capabilities of GPUs. (In contrast, it took many more years before game developers switched from writing their games in CPU assembly language to using higher-level languages.)

This sort of rapid change makes a "gems"-style book a natural fit for assembling the state of the art and disseminating it to the developer community. Featuring chapters written by acknowledged experts, *GPU Gems 2* provides broad coverage of the most exciting new ideas in the field.

Innovations in graphics hardware and programming environments have inspired further innovations in how to use programmability. While programmable shading has long been a staple of offline software rendering, the advent of programmability on GPUs has

NVIDIA Ex1023
Page Q-33

led to the invention of a wide variety of new techniques for programmable shading. Going far beyond procedural pattern generation and texture composition, the state of the art of using shaders on GPUs is rapidly breaking completely new ground, leading to novel techniques for animation, lighting, particle systems, and much more.

Indeed, the flexibility and speed of GPUs have fostered considerable interest in doing computations on GPUs that go beyond computer graphics: general-purpose computation on GPUs, or "GPGPU." This volume of the *GPU Gems* series devotes a significant number of chapters to this new topic, including an overview of GPGPU programming techniques as well as in-depth discussions of a number of representative applications and key algorithms. As GPUs continue to increase in performance more quickly than CPUs, these topics will gain in importance for more and more programmers because GPUs will provide superior results for many computationally intensive applications.

With this background, we sent out a public call for participation in *GPU Gems 2*. The response was overwhelming: more than 150 chapters were proposed in the short time that submissions were open, covering a variety of topics related to GPU programming. We were able to include only about a third of them in this volume; many excellent submissions could not be included purely because of constraints on the physical size of the book. It was difficult for the editors to whittle down the chapters to the 48 included here, and we would like to thank everyone who submitted proposals.

The accepted chapters went through a rigorous review process in which the book's editors, the authors of other chapters in the same part of the book, and in some cases additional reviewers from NVIDIA carefully read them and suggested improvements or changes. In almost every case, this step noticeably improved the final chapter, due to the high-quality feedback provided by the reviewers. We thank all of the reviewers for the time and effort they put into this important part of the production process.

Intended Audience

We expect readers to be familiar with the fundamentals of computer graphics and GPU programming, including graphics APIs such as Direct3D and OpenGL, as well as GPU languages such as HLSL, GLSL, and Cg. Readers interested in GPGPU programming may find it helpful to have some basic familiarity with parallel programming concepts.

Developers of games, visualization applications, and other interactive applications, as well as researchers in computer graphics, will find *GPU Gems 2* an invaluable daily resource. In particular, those developing for next-generation consoles will find a wealth of timely and applicable content.

NVIDIA Ex1023
Page Q-34

Trying the Examples

GPU Gems 2 comes with a CD-ROM that includes code samples, movies, and other demonstrations of the techniques described in the book. This CD is a valuable supplement to the ideas explained in the book. In many cases, the working examples provided by the authors will provide additional enlightenment. You can find sample chapters, updated CD content, supplementary materials, and more at the book's Web site, **http://developer.nvidia.com/GPUGems2/**.

Acknowledgments

An enormous amount of work by many different people went into this book. First, the contributors wrote a great collection of chapters on a tight schedule. Their efforts have made this collection as valuable, timely, and thought provoking as it is.

The section editors—Kevin Bjorke, Cem Cebenoyan, Simon Green, Mark Harris, Craig Kolb, and Matthias Wloka—put in many hours of hard work on this project, working with authors to polish their chapters and their results until they shone, consulting with them about best practices for GPU programming, and gently reminding them of deadlines. Without their focus and dedication, we'd still be working through the queue of submissions. Chris Seitz also kindly took care of many legal, logistical, and business issues related to the book's production.

Many others at NVIDIA also contributed to *GPU Gems 2*. We thank Spender Yuen once again for his patience while doing a wonderful job on the book's diagrams, as well as on the cover. Helen Ho also helped with the illustrations as their number grew to more than 150. We are grateful to Caroline Lie and her team for their continual support of our projects. Similarly, Teresa Saffaie and Catherine Kilkenny have always been ready and willing to provide help with copyediting as our projects develop. Jim Black coordinated communication with a number of developers and contributors, including Tim Sweeney, to whom we are grateful for writing a wonderfully focused and astute Foreword.

At Addison-Wesley Professional, Peter Gordon, Julie Nahil, and Kim Boedigheimer oversaw this project and helped to expedite the production pipeline so we could release this book in as timely a manner as possible. Christopher Keane's copyediting skills and Jules Keane's assistance improved the content immeasurably, and Curt Johnson helped to market the book when it was finally complete.

The support of several members of NVIDIA's management team was instrumental to this project's success. Mark Daly and Dan Vivoli saw the value of putting together a

NVIDIA Ex1023
Page Q-35

second volume in the *GPU Gems* series and supported this book throughout. Nick Triantos allowed Matt the time to work on this project and gave feedback on a number of the GPGPU chapters. Jonah Alben and Tony Tamasi provided insightful perspectives and valuable feedback about the chapter on the GeForce 6 Series architecture. We give sincere thanks to Jen-Hsun Huang for commissioning this project and fostering the innovative, challenging, and forward-thinking environment that makes NVIDIA such an exhilarating place to work.

Finally, we thank all of our colleagues at NVIDIA for continuing to push the envelope of computer graphics day by day; their efforts make projects like this possible.

Matt Pharr
NVIDIA Corporation

Randima (Randy) Fernando
NVIDIA Corporation

NVIDIA Ex1023
Page Q-36

Contributors



Tomas Akenine-Möller, Lund University

Tomas Akenine-Möller is an associate professor in the department of computer science at Lund University in Sweden. His main interests lie in real-time rendering, graphics on mobile devices, and shadows.



Arul Asirvatham, Microsoft Research

Arul Asirvatham is a Ph.D. student in the School of Computing, University of Utah. He received a B.Tech. in computer science and engineering in 2002 from the Indian Institute of Information Technology in India. His primary research interest is digital geometry processing; he has been working on mesh parameterization techniques. He is also interested in real-time computer graphics. Currently he is focusing on rendering huge terrain data sets interactively.



Jiří Bittner, Vienna University of Technology

Jiří Bittner is currently affiliated with the Institute of Computer Graphics and Algorithms of the Vienna University of Technology. He received his Ph.D. in 2003 from the department of computer science and engineering of the Czech Technical University in Prague. His research interests include visibility computations, efficient real-time rendering techniques, global illumination, and computational geometry.



Kevin Bjorke, NVIDIA Corporation

Kevin Bjorke is a member of the Developer Technology group at NVIDIA. He was a section editor and authored several chapters for *GPU Gems*. He has an extensive and award-winning production background in live-action and computer-animated films, television, advertising, theme park rides, and, of course, games. Kevin has been a regular speaker at events such as Game Developers Conference (GDC) and ACM SIGGRAPH since the mid-1980s. His current work at NVIDIA involves exploring and harnessing the power of programmable shading for high-quality real-world applications.

NVIDIA Ex1023
Page Q-37



Ian Buck, Stanford University

Ian Buck is completing his Ph.D. in computer science at the Stanford Computer Graphics Lab, researching general-purpose computing models for GPUs. He received a B.S.E. in computer science from Princeton University in 1999 and received fellowships from the Stanford School of Engineering and NVIDIA. His research focuses on programming language design for graphics hardware as well as general-computing applications that map to graphics hardware architectures.



Michael Bunnell, NVIDIA Corporation

Michael Bunnell graduated from Southern Methodist University with degrees in computer science and electrical engineering. He wrote the Megamax C compiler for the Macintosh, Atari ST, and Apple IIGS before cofounding what is now LynuxWorks. After working on real-time operating systems for nine years, he moved to Silicon Graphics, focusing on image-processing, video, and graphics software. Next, he worked at Gigapixel, then at 3dfx, and now at NVIDIA, where, interestingly enough, he is working on compilers again—this time, shader compilers.



Iain Cantlay, Climax Entertainment

Iain Cantlay is currently a senior engineer at Climax, where he was responsible for the graphical aspects of the *Leviathan* MMO engine and *Warhammer Online*. His current projects include *MotoGP 3* (to be published for Xbox and PC by THQ in 2005). Iain is passionate about exploiting the best visuals from the latest technology, but natural phenomena interest him most: terrain, skies, clouds, vegetation, and water.



Francesco Carucci, Lionhead Studios

Francesco Carucci graduated from the Politecnico di Torino in Italy with a degree in software engineering. When he was eight, rather than make pizza (like every good Italian), he decided to make video games, and he tried to animate a running character in BASIC on an Intellivision. He is now writing code to animate running characters at Lionhead, working on the latest rendering technology for *Black & White 2*. He contributed to various Italian technical 3D sites and to *ShaderX2*. His main interests include lighting and shadowing algorithms, 3D software construction, and the latest 3D hardware architectures. And when he needs help, he writes shaders for food.



Cem Cebenoyan, NVIDIA Corporation

Cem Cebenoyan is a software engineer working in the Developer Technology group at NVIDIA. He was an author and section editor for *GPU Gems*. He spends his days researching graphics techniques and helping game developers get the most out of graphics hardware. He has spoken at past Game Developer Conferences on character animation, graphics performance, and nonphotorealistic rendering. Before joining NVIDIA, he was a student and research assistant in the Graphics, Visualization, and Usability Lab at the Georgia Institute of Technology.

NVIDIA Ex1023
Page Q-38



Eric Chan, Massachusetts Institute of Technology

Eric Chan is a Ph.D. student in the Computer Science and Artificial Intelligence Laboratory at M.I.T. He fiddles with graphics architectures, shading languages, and real-time rendering algorithms. He has recently developed efficient methods for rendering hard and soft shadows. Before attending graduate school, Eric was a research staff member in the Stanford Computer Graphics Laboratory. As part of the Real-Time Programmable Shading team, he wrote compiler back ends for the NV30 and R300 fragment architectures and developed a pass-decomposition algorithm for virtualizing hardware resources. Eric enjoys photography and spends an unreasonable amount of his free time behind the camera.



Greg Coombe, The University of North Carolina at Chapel Hill

Greg Coombe is a graduate student at the University of North Carolina at Chapel Hill. He received a B.S. in mathematics and a B.S. in computer science from the University of Utah in 2000. Greg's research interests include global illumination, graphics hardware, nonphotorealistic rendering, virtual environments, and 3D modeling. During the course of his graduate studies, he has worked briefly at Intel, NVIDIA, and Vicious Cycle Software. Greg was the recipient of the NVIDIA Graduate Fellowship in 2003 and 2004.



Jürgen Döllner, University of Potsdam, Hasso-Plattner-Institute

Jürgen Döllner, a professor at the Hasso-Plattner-Institute of the University of Potsdam, directs the computer graphics and visualization division. He has studied mathematics and computer science and received a Ph.D. in computer science. He researches and teaches in real-time computer graphics and spatial visualization.



William Donnelly, NVIDIA Corporation and University of Waterloo

William Donnelly is a fourth-year undergraduate in computer science and mathematics at the University of Waterloo in Ontario. He interned with Okino Computer Graphics, where he worked on global illumination and volumetric rendering; and with NVIDIA's Demo Team, where he worked on the "Last Chance Gas" and "Nalu" demos. He has been destined for greatness in computer graphics since mastering the art of the Bezier spline at age ten.



Frédo Durand, Massachusetts Institute of Technology

Frédo Durand received a Ph.D. from Grenoble University in France in 1999, where he worked on both theoretical and practical aspects of 3D visibility. From 1999 until 2002, he was a postdoc in the M.I.T. Computer Graphics Group, where he is now an assistant professor. His research interests span most aspects of picture generation and creation, including realistic graphics, real-time rendering, nonphotorealistic rendering, and computational photography. He received a Eurographics Young Researcher Award in 2004. *(Digital drawing courtesy of Victor Ostromoukhov)*

NVIDIA Ex1023
Page Q-39



Eric Enderton, NVIDIA Corporation

Eric Enderton is a senior engineer at NVIDIA, where he is working on the Gelato film renderer. After studying computer science at the University of California, Berkeley, Eric spent a decade developing rendering and animation software at Industrial Light & Magic, and he later consulted at other studios. His film credits include *Terminator 2*; *Jurassic Park*; and *Star Wars, Episode I: The Phantom Menace*.



Zhe Fan, Stony Brook University

Zhe Fan is a Ph.D. candidate in the computer science department at Stony Brook University. He received a B.S. in computer science from the University of Science and Technology of China in 1998 and an M.S. in computer science from the Chinese Academy of Sciences in 2001. His current research interests include GPU clusters for general-purpose computation, parallel graphics and visualization, and modeling of amorphous phenomena.



Randima Fernando, NVIDIA Corporation

Randima (Randy) Fernando has loved computer graphics since age eight. Working in NVIDIA's Developer Technology group, he helps teach developers how to take advantage of the latest GPU technology. Randy has a B.S. in computer science and an M.S. in computer graphics, both from Cornell University. He has published research in SIGGRAPH and is coauthor, with Mark Kilgard, of *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. He edited *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* and is the *GPU Gems* series editor.



Nathaniel Fout, University of California, Davis

Nathaniel Fout received a B.S. in chemical engineering and an M.S. in computer science from the University of Tennessee in 2002 and 2003, respectively. He is a Ph.D. student in computer science at the University of California, Davis, where he is a member of the Institute for Data Analysis and Visualization. His research interests include volumetric compression for rendering, multivariate and comparative visualization, and tensor visualization.



James Fung, University of Toronto

James Fung is completing his Ph.D. in engineering. He received a B.A.Sc. in engineering science and an M.S. in electrical engineering from the University of Toronto. His research interests include wearable computing, mediated reality, and exploring new types of musical instrument interfaces based on EEG brain-wave signal processing. His most recent work has been the development of the GPU-based computer vision and mediated reality library called OpenVIDIA.

NVIDIA Ex1023
Page Q-40



Simon Green, NVIDIA Corporation

Simon Green is a senior software engineer in the Developer Technology group at NVIDIA. He started graphics programming on the Sinclair ZX-81, which had 1 kB of RAM and a screen resolution of 64 × 48 pixels. He received a B.S. in computer science from the University of Reading, in the United Kingdom, in 1994. Since 1999 Simon has found a stable home at NVIDIA, where he develops new rendering techniques and helps application developers take maximum advantage of GPU hardware. He is a frequent presenter at GDC, has written for *Amiga Shopper* and *Wired* magazines, and was a section editor for *GPU Gems*. His research interests include cellular automata, general-purpose computation on GPUs, and analog synthesizers.



Toshiya Hachisuka, University of Tokyo

Toshiya Hachisuka is an undergraduate in the Department of Systems Innovation at the University of Tokyo. He also works as a programmer for MagicPictures, integrating cutting-edge research results into current computer graphics software. He has studied computer graphics since age ten. His current research interests are physically based rendering, physically based modeling, real-time rendering techniques, and general-purpose computation on GPUs.



Markus Hadwiger, VRVis Research Center

Markus Hadwiger received his Ph.D. in computer science from the Vienna University of Technology in 2004, where he concentrated on high-quality real-time volume rendering and texture filtering with graphics hardware, in cooperation with the VRVis Research Center. He has been a researcher at VRVis since 2000, working in the Basic Research on Visualization group and the Medical Visualization group (since 2004). From 1996 to 2001, he was also the lead programmer of the cross-platform 3D space-shooter game *Parsec*, which is now an open source project.



Mark Harris, NVIDIA Corporation

Mark Harris received a B.S. from the University of Notre Dame in 1998 and a Ph.D. in computer science from the University of North Carolina at Chapel Hill (UNC) in 2003. At UNC, Mark's research covered a wide variety of computer graphics topics, including real-time cloud simulation and rendering, general-purpose computation on GPUs, global illumination, nonphotorealistic rendering, and virtual environments. Mark is now a member of NVIDIA's Developer Technology team based in the United Kingdom.



Jon Hasselgren, Lund University

Jon Hasselgren received an M.Sc. from Lund University. He now pursues graduate studies in the computer science department, where he researches graphics for mobile phones.

NVIDIA Ex1023
Page Q-41



Oliver Hoeller, Piranha Bytes

Oliver Hoeller is a senior software engineer at Piranha Bytes, which developed the RPGs *Gothic I* and *Gothic II*. Previously he was director of development at H2Labs/Codecult, where he was responsible for development and architecture design of the Codecreatures game system. He was an active member of the German demo scene in the 1980s and early 1990s. After exploring different areas—developing music software, creating a security program, and working as a Web services consultant—Oliver returned to his roots and now guarantees a high level of visual quality for Piranha Bytes' forthcoming *Gothic III*.



Hugues Hoppe, Microsoft Research

Hugues Hoppe is a senior researcher in the Computer Graphics Group at Microsoft Research. His primary interests lie in the acquisition, representation, and rendering of geometric models. He received the 2004 ACM SIGGRAPH Achievement Award for his pioneering work on surface reconstruction, progressive meshes, geometry texturing, and geometry images. His publications include twenty papers at ACM SIGGRAPH, and he is associate editor of *ACM Transactions on Graphics*. He received a B.S. in electrical engineering in 1989 and a Ph.D. in computer science in 1994 from the University of Washington.



Daniel Horn, Stanford University

Daniel Horn is a Ph.D. candidate at the Stanford Computer Graphics Lab; he received his B.S. from the University of California, Berkeley. While Daniel focuses on programming graphics hardware and real-time graphics, theory and compilers have always interested him deeply, and he tries to incorporate knowledge from those fields into his graphics research. In his spare time, Daniel enjoys hacking with his brother, Patrick, on their open source space sim, *Vega Strike*. He also enjoys roaming with friends in the Bay Area's many natural parks, from Palo Alto's Foothills Park to Berkeley's Tilden Park.



Samuel Hornus, GRAVIR/IMAG–INRIA

Samuel Hornus is a Ph.D. candidate at INRIA in Grenoble, France. He is a former student of the Ecole Normale Supérieure de Cachan. His research focuses on 3D visibility problems, as well as other aspects of computer graphics, such as texture authoring, interactive walkthroughs, real-time shadows, realistic rendering, implicit surfaces, and image-based modeling.

NVIDIA Ex1023
Page Q-42



Arie Kaufman, Stony Brook University

Arie Kaufman is the director of the Center for Visual Computing, a distinguished professor and chair of the Computer Science Department, and distinguished professor of radiology at Stony Brook University. He received a B.S. in mathematics and physics from the Hebrew University of Jerusalem in 1969; an M.S. in computer science from the Weizmann Institute of Science, Rehovot, Israel, in 1973; and a Ph.D. in computer science from the Ben-Gurion University, Israel, in 1977. Kaufman has conducted research and consulted for more than thirty years, with numerous publications in volume visualization; graphics architectures, algorithms, and languages; virtual reality; user interfaces; and multimedia.



Jan Kautz, Massachusetts Institute of Technology

Jan Kautz is a postdoctoral researcher at M.I.T. He is particularly interested in realistic shading and lighting, hardware-accelerated rendering, textures and reflection properties, and interactive computer graphics. He received his Ph.D. in computer science from the Max-Planck-Institut für Informatik in Germany; a diploma in computer science from the University of Erlangen in Germany; and an M.Math. from the University of Waterloo in Ontario.



Emmett Kilgariff, NVIDIA Corporation

Emmett Kilgariff is a director of architecture in the GPU group at NVIDIA, where he has contributed to the design of many GeForce chips, including the GeForce 6 and GeForce 7 Series. He has more than twenty years of experience designing graphics hardware, at Sun Microsystems, Silicon Graphics, 3dfx, and many small companies whose memories have faded over time.



Gary King, NVIDIA Corporation

Unscrupulous. Unconventional. Uncouth. Unkempt. All are accurate adjectives for the worst thing to happen to the graphics industry since Execute Buffers. A master of GPU arcana, lore, and the occult, he spends his days at NVIDIA crafting increasingly ingeniously nefarious rendering techniques, imbuing next-generation architectures with unholy energies, worshipping the Dark Lord, and kicking puppies.



Peter Kipfer, Technische Universität München

Peter Kipfer is a postdoctoral researcher in the Computer Graphics and Visualization Group at the Technische Universität München. He received his Ph.D. from the University of Erlangen-Nürnberg in 2003 for his work on parallel and distributed visualization and rendering within the KONWIHR supercomputing project. His current research focuses on general-purpose computing and geometry processing on the GPU.

NVIDIA Ex1023
Page Q-43



Joe Kniss, University of Utah

Joe Kniss is a Ph.D. student in computer science at the University of Utah, where he is a member of the Scientific Computing and Imaging Institute. His research interests include nonpolygonal rendering, light transport in participating media, user-interface design, and all things GPU. He is a Department of Energy High-Performance Computer Science graduate fellow.



Craig Kolb, NVIDIA Corporation

Craig Kolb has been interested in computer graphics since he began writing games on his high school's sub-megaflop PDP-11. He received a B.A. and an M.Sc. from Princeton, where he wrote the first version of rayshade, a popular ray tracer, as part of his senior thesis. He spent the 1990s waiting for frames to render: first as a research assistant to Benoit Mandelbrot at Yale, then as a Ph.D. candidate researching camera and rendering systems at Princeton and Stanford, and later as head of rendering development at Pixar Animation Studios. In 2000 he cofounded Exluna and now works in the Software Architecture group at NVIDIA finding novel ways to push multi-gigaflop GPUs to their limits.



Jens Krüger, Technische Universität München

Jens Krüger is a Ph.D. student in the Computer Graphics and Visualization Group at the Technische Universität München. Jens's current research focuses on GPU solutions to numerical problems, often arising in physically based simulations. He has published papers on GPU programming at conferences such as ACM SIGGRAPH and IEEE Visualization. In 2004 he received an ATI Fellowship, which honors outstanding graduate students in areas related to computer graphics and graphics systems.



Yuri Kryachko, 1C:Maddox Games

Yuri Kryachko is the 3D graphics and effects programmer on *IL-2 Sturmovik*, *WW2*, *IL-2 Sturmovik: Forgotten Battles*, *AEP*, and *Pacific Fighters*. He has been at Maddox Games since 1996, and he's been playing and creating PC games since writing his first 2D game in 1987. He received an M.S. from the department of applied mathematics of the Moscow State Engineering Physics Institute (Technical University). Previous game projects include *City3D–Drive Simulator (ELF)* in 1995 and *Helicopter Simulator* from 1995 to 1996.



Sylvain Lefebvre, GRAVIR/IMAG–INRIA

Sylvain Lefebvre is a final-year Ph.D. student at INRIA in Grenoble, France. He received an M.S. in computer science from the INPG University, Grenoble, in 2001. His research focuses on developing new texturing methods for creating, storing, and rendering highly detailed textures for real-time applications. Recently he has worked on landscape texturing, direct painting on meshes, and the progressive loading of texture maps. He is also interested in many aspects of game programming.

NVIDIA Ex1023
Page Q-44



Aaron Lefohn, University of California, Davis

Aaron Lefohn is a Ph.D. student in the computer science department at the University of California, Davis, and a graphics software engineer at Pixar Animation Studios. His current research focuses on data-parallel data structures and programming models and their application to high-quality interactive rendering. Aaron completed his M.S. in computer science at the University of Utah in 2003; he received an M.S. in theoretical chemistry from the University of Utah in 2001 and a B.A. in chemistry from Whitman College in 1997. Aaron is a National Science Foundation graduate fellow in computer science.



Martin-Karl Lefrançois, mental images

Martin-Karl Lefrançois is senior graphics software engineer at mental images in Berlin, maker of the mental ray renderer and other graphics software products. Under his lead, his team at mental images delivered automatic GPU support in mental ray 3.3 and is responsible for GPU acceleration support in all mental images products. After graduating with a degree in computer science and mathematics from the University of Sherbrooke in Quebec, he worked as a graphics developer for nearly ten years at Softimage in Montreal and Tokyo before leading the core game engine team at A2M.



Wei Li, Siemens Corporate Research

Wei Li is a research scientist at Siemens Corporate Research in Princeton, New Jersey. His current research focuses on texture-based volume rendering and general-purpose computation on the GPU. He received an M.S. and a Ph.D. in computer science from Stony Brook University in 2001 and 2004, respectively. He also received a B.S. and an M.S. in electrical engineering from Xi'an Jiaotong University in China in 1992 and 1995, respectively.



Donald Liu, Siemens Medical Solutions USA

Donald Liu received a B.Eng. from Qinghua University in Beijing in 1984; he received an M.Eng. and a D.Eng. from the University of Tokyo in 1988 and 1991, respectively. He was an assistant professor at Sophia University in Tokyo for a year before joining the faculty of the electrical engineering department at the University of Rochester in New York. Since 1997 he has been with the Siemens Medical Solutions Ultrasound Group in Issaquah, Washington, where he is currently a senior staff systems engineer. He is a senior member of IEEE and a recipient of the National Institutes of Health FIRST award. His research interests include analysis and correction of ultrasonic wavefront distortion, efficient image formation, and digital signal processing.

NVIDIA Ex1023
Page Q-45



Paulius Micikevicius, Armstrong Atlantic State University

Paulius Micikevicius received a B.S. in computer science from Midwestern State University in 1998 and a Ph.D. in computer science from the University of Central Florida (UCF) in 2002. He is an assistant professor at Armstrong Atlantic State University in Savannah, Georgia, as well as a research associate at the Media Convergence Laboratory at UCF. His research interests include real-time graphics, graphics processing for mixed/augmented reality experiences, and parallel computing and graph theory.



Fabrice Neyret, GRAVIR/IMAG–INRIA

Fabrice Neyret has worked on the R&D teams of several companies, including TDI in Paris and Alias|Wavefront in Toronto. He received a master's degree in applied mathematics, an engineering degree from Telecom Paris (ENST), and a Ph.D. in computer science. He did his postdoctoral work at the University of Toronto. He is currently a full-time CNRS researcher at GRAVIR lab in Grenoble, France. His research interests include natural phenomena (especially water and clouds), highly complex scenes (such as landscapes covered by forest), textures, local illumination and shaders, alternate representations (such as volumetric textures), phenomenological approaches, and, of course, getting the most out of GPUs. He is also involved in pedagogic software (such as MobiNet), scientific popularization, and writing short stories.



Hubert Nguyen, NVIDIA Corporation

Hubert Nguyen is a software engineer on the NVIDIA Demo Team. He spends his time searching for novel effects that show off the features of NVIDIA's latest GPUs. He most recently worked on "Nalu," NVIDIA's mermaid. Before joining NVIDIA, Hubert was at 3dfx interactive, the creators of Voodoo Graphics. Prior to 3dfx, Hubert was part of the R&D department of Cryo Interactive in Paris. Hubert started to develop 3D graphics programs when he was involved in the European demo scene. He holds a degree in computer science.



Marc Nienhaus, University of Potsdam, Hasso-Plattner-Institute

Marc Nienhaus is a Ph.D. candidate at the Hasso-Plattner-Institute of the University of Potsdam. He studied mathematics and computer science and has worked as a software engineer focusing on computer graphics. His research interests include real-time rendering, nonphotorealistic rendering, and depiction strategies for symbolizing dynamics.

NVIDIA Ex1023
Page Q-46



Justin Novosad, discreet

Justin Novosad is a software developer for discreet (a division of Autodesk). He received a bachelor's degree in computer engineering and a master's degree in medical imaging, both from École Polytechnique de Montréal, in 2001 and 2003, respectively. Justin is a member of the "effects" team at discreet, working on the Inferno, Flame, and Flint visual effects and digital compositing products. Before joining discreet in 2004, he was a research engineer at Sainte-Justine Hospital in Montreal, where he developed computer vision algorithms for the study of spinal deformities from X-ray data. His fields of interest include computer graphics, computer vision, machine learning, image processing, and applied mathematics. He is a cofounder of the ACM SIGGRAPH Montreal Professional Chapter.



Lennart Ohlsson, Lund University

Lennart Ohlsson is an assistant professor in the computer science department at Lund University. His primary research interest is software architecture for computer graphics.



Jon Olick, 2015

Jon Olick has been creating games since age 11. He is a senior software engineer specializing in graphics technology and engine design at 2015, where he has worked on titles such as *Medal of Honor: Allied Assault* and *Men of Valor: Vietnam.* He is now developing engine technology for future products.



Sean O'Neil

Sean O'Neil graduated from Georgia Tech in 1995 with a B.S. in computer science. He lives in Atlanta with his wife and two wonderful children. All of his full-time positions have been in the telecommunications industry, so for now graphics programming is just a hobby.



John Owens, University of California, Davis

John Owens is an assistant professor of electrical and computer engineering at the University of California, Davis, where he leads research projects in graphics hardware/software and wireless sensor networks. Prior to his appointment at Davis, he earned an M.S. and a Ph.D. in electrical engineering from Stanford University in 1997 and 2002, respectively. At Stanford he was an architect of the Imagine Stream Processor and a member of the Concurrent VLSI Architecture Group and the Computer Graphics Laboratory. He received a B.S. in electrical engineering and computer science from the University of California, Berkeley, in 1995.

NVIDIA Ex1023
Page Q-47



Kurt Pelzer, Piranha Bytes

Kurt Pelzer is a senior software engineer at Piranha Bytes, where he worked on the PC game *Gothic*, the top-selling *Gothic II* (awarded RPG of the Year in Germany during 2001 and 2002, respectively), and the add-on *Gothic II: The Night of the Raven.* Previously he was a senior programmer at Codecult and developed several real-time simulations and technology demos built on Codecult's 3D engine. Kurt has published in *GPU Gems*, *ShaderX2*, and *Game Programming Gems 4.*



Matt Pharr, NVIDIA Corporation

Matt Pharr is a senior software developer in the Software Architecture group at NVIDIA, where he works on Cg and interactive rendering techniques. He is coauthor, with Greg Humphreys, of *Physically Based Rendering: From Theory to Implementation.* Previously he was a cofounder of Exluna and a Ph.D. student in the Stanford Computer Graphics Lab, where he researched systems issues for rendering and theoretical foundations of rendering; he published a series of SIGGRAPH papers on these topics.



Jeremy Selan, Sony Pictures Imageworks

Jeremy Selan currently pioneers color and lighting tools at Sony Pictures Imageworks. His work has been utilized on numerous motion pictures, most recently on *Spider-Man 2.* Professionally, he maintains an active interest in colorimetry and digital cinema. He is a graduate of the Program of Computer Graphics and the School of Electrical and Computer Engineering at Cornell University. In his free time—drawn by a climate markedly superior to that of his hometown, Skokie, Illinois—Jeremy is an aspiring Santa Monica beach bum.



Oles Shishkovtsov, GSC Game World

Oles Shishkovtsov became interested in programming and graphics at age 13; by age 17 he had won two national competitions in programming and enrolled at the Junior Academy of Science in Ukraine. At 19 he started working for White Lynx as a software developer/graphics programmer, where he successfully completed three projects. Since 2000 he has worked for GSC Game World as an engine architect/team leader and has continued doing R&D in his free time. He has spent the last three years working on *S.T.A.L.K.E.R.: Shadows of Chernobyl.*



Christian Sigg, ETH Zurich

Christian Sigg received his degree in computational science and engineering from the Swiss Federal Institute of Technology Zurich. He became interested in computer graphics during a semester abroad at the University of Texas at Austin, where he worked on parallel volume rendering at the Computational Visualization Center. He is working on his Ph.D. at the ETH Zurich Computer Graphics Laboratory. His research interests lie in the area of algorithms for implicit surface representations using graphics hardware.

NVIDIA Ex1023
Page Q-48



Tiago Sousa, Crytek

Tiago Sousa is a self-taught game and graphics programmer who has worked at Crytek as an R&D software engineer for the last two years. He has contributed to most of the special effects in Crytek's games. In 1999, before joining Crytek, he cofounded a pioneering game development team in Portugal and studied computer science at a local university. He spends most of his time researching real-time and non-real-time graphics and reading all kinds of technical books.



Thilaka Sumanaweera, Siemens Medical Solutions USA

Thilaka Sumanaweera has been having fun with first GL and then OpenGL since the late 1980s, creating 2, 3, and 4D applications in computer vision, image processing, and medical imaging. He received his Ph.D. in electrical engineering from Stanford University in 1992. He then joined the Radiological Sciences Laboratory at Stanford's Radiology Department as a postdoc and a research associate developing CT/MRI image fusion and image-guided neurosurgery. Currently he is a Fellow in the Siemens Medical Solutions Ultrasound Division, working in the areas of volume rendering, motion detection and compensation, and image segmentation. He holds 24 patents for techniques related to medical imaging and visualization, and he has published extensively in medical journals.



Yury Uralsky, NVIDIA Corporation

Yury Uralsky became interested in games and computer graphics when the ZX Spectrum 48K was a dream machine and writing software rasterizers in assembly language was fun. He received an M.S. in computer science from the Moscow State Technical University in 2001. He worked as a graphics engine programmer for Eagle Dynamics, creating graphics for the flight simulator *LockOn: Modern Air Combat*. He joined the NVIDIA Developer Technology group in March 2004 and enjoys pushing 3D graphics forward in the NVIDIA Moscow office.



Pete Warden, Apple Computer

Pete Warden has worked as a graphics engine programmer on PC, PSX, PS2, GameCube, and Xbox titles, specializing in low-level assembler and vector unit programming on the PS2. He has also published 45 open source video filters that run in a variety of real-time video applications on Windows, Linux, and OS X, including After Effects, and helped to create the Freeframe open plug-in standard. Pete is now part of the team working on Apple's Motion video effects package, a fully GPU-based image-processing application. He has written many of its original video filters and also works on the rendering engine.

NVIDIA Ex1023
Page Q-49



Li-Yi Wei, NVIDIA Corporation

Li-Yi Wei is a 3D graphics architect at NVIDIA Corporation. He received a B.S. in electrical engineering from the National Taiwan University in 1993 and a Ph.D. in electrical engineering from Stanford University in 2001. He spends 1 percent of his time designing next-generation graphics hardware and the remaining 99 percent verifying that the design actually works. When not wreaking havoc on NVIDIA's chips, he enjoys researching various fields of computer graphics. He is a frequent contributor to SIGGRAPH and other academic conferences.



Xiaoming Wei, Stony Brook University

Xiaoming Wei is an assistant professor of computer science at Iona College in New Rochelle, New York. She received her Ph.D. in computer science from Stony Brook University in 2004. She received a B.Sc. from the Beijing University of Aeronautics and Astronautics in 1995 and an M.Sc. in computer science from Tsinghua University in Beijing in 1998. Her research interests include physically based modeling, natural phenomena modeling, and computer animation.



Rüdiger Westermann, Technische Universität München

Rüdiger Westermann studied computer science at the Technical University Darmstadt, Germany. He received a Ph.D. in computer science from the University of Dortmund, Germany. In 2002 he was appointed the chair of Computer Graphics and Visualization at the Technische Universität München. His research interests include general-purpose computing on GPUs, hardware-accelerated visualization and image synthesis, hierarchical methods in scientific visualization, volume rendering, flow visualization, and parallel graphics algorithms.



Daniel Wexler, NVIDIA Corporation

Daniel Wexler attended the University of California, Berkeley, where he studied with the graphics research group before leaving school to work at Sun Microsystems. He worked at Xaos Tools before joining the R&D team at Pacific Data Images (PDI) in 1995. After spending six years writing a new renderer and shading system for PDI, which was used on a variety of feature film projects including *Antz* and *Shrek*, he joined NVIDIA to work on hardware-based rendering systems with Larry Gritz and the rest of the NVIDIA architecture team.



David Whatley, Simutronics Corporation

David Whatley is president and CEO of Simutronics Corporation and a developer and publisher of online games. His passion for online gaming led him to found Simutronics in 1987, when he was 20 years old. David—chief designer and technology architect of most of the company's games—has won numerous awards, including Computer Gaming World's first Online Game of the Year award for *CyberStrike*. His current focus is on alternate techniques for more photorealistic rendering of 3D environments in games.

NVIDIA Ex1023
Page Q-50



Michael Wimmer, Vienna University of Technology

Michael Wimmer is an assistant professor at the Institute of Computer Graphics and Algorithms of the Vienna University of Technology, where he received an M.Sc. in 1997 and a Ph.D. in 2001. His current research interests are real-time rendering, virtual and augmented reality, computer games, and real-time visualization of urban environments; he has coauthored several scientific papers in these fields. He also teaches courses on 3D computer games and real-time rendering.



Matthias Wloka, NVIDIA Corporation

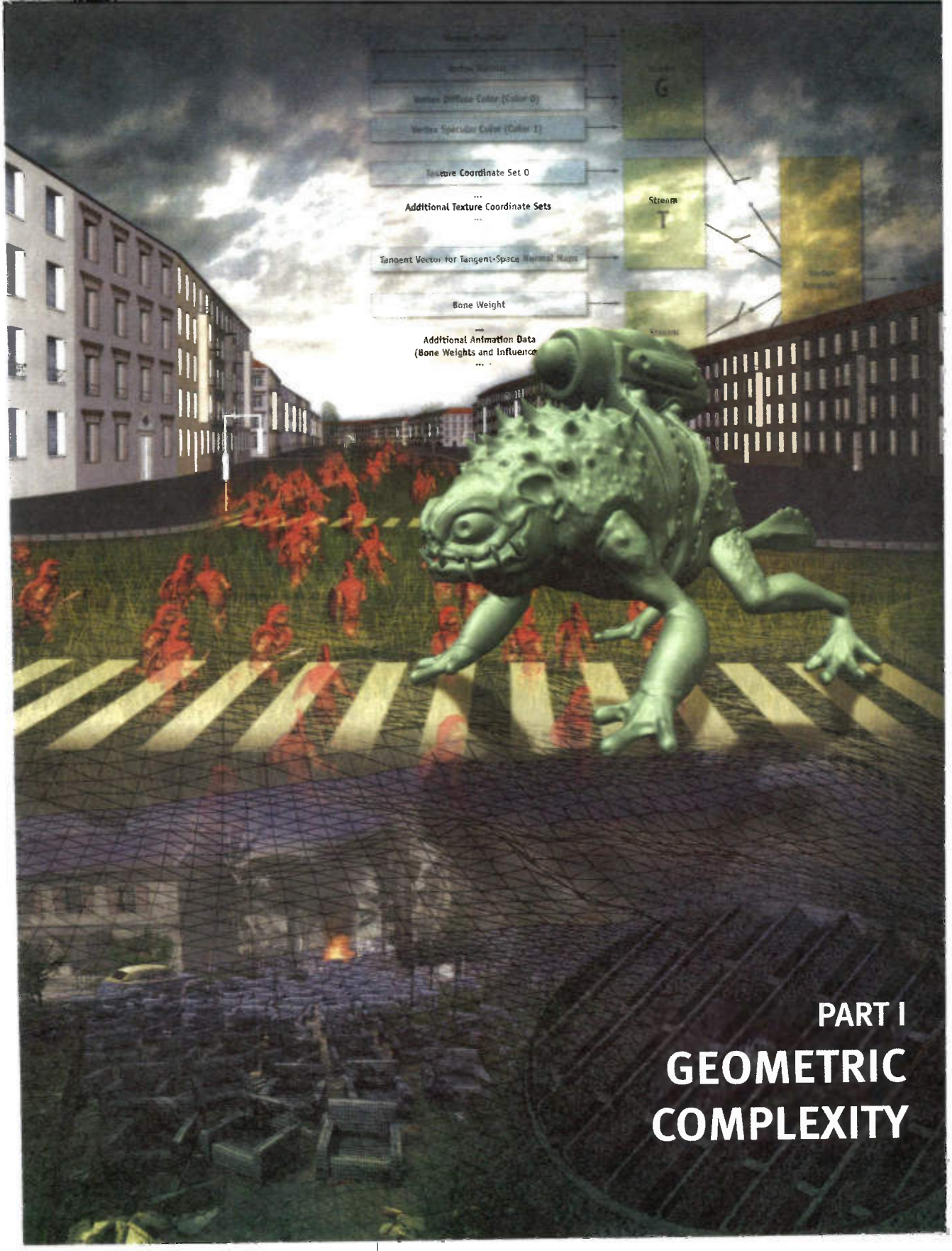
Matthias Wloka is a software engineer in the Developer Technology group at NVIDIA. His primary responsibility is to collaborate with game developers to enhance image quality and graphics performance of their games; he is also a regular contributor at game developer conferences, such as GDC. Matthias's passion for computer gaming started at age 15 when he discovered that his school's Commodore PET 2001 computers could also play Black Jack. He started writing his own games soon thereafter and continues to use the latest graphics hardware to explore the limits of interactive real-time rendering. Before joining NVIDIA, Matthias was a game developer at GameFX/THQ. He received an M.Sc. in computer science from Brown University in 1990 and a B.Sc from Christian-Albrechts-University in Kiel, Germany, in 1987.



Cliff Woolley, University of Virginia

Cliff Woolley is a Ph.D. student in computer science at the University of Virginia. His research interests include interactive rendering techniques, sparse sample reconstruction, and general-purpose computation using programmable graphics hardware. He received an M.C.S. in computer graphics from the University of Virginia in 2003 and a B.A. in computer science and theater at Washington and Lee University in 1999.

NVIDIA Ex1023
Page Q-51



PART I
GEOMETRIC COMPLEXITY

Today's games are visually more interesting and complex than ever before. Geometric complexity—how many objects are visible and how detailed each looks—is one of the dimensions in which games are making leaps and bounds.

Advances in technology are partly responsible for these leaps and bounds: CPUs, memory, and buses all have become faster, but specifically GPUs are undergoing significant change and are becoming ever more powerful—at a rate faster than Moore's Law.

These GPU changes include incorporating fixed-function processing for the vertex- and pixel-shading units, then generalizing those to be fully programmable. GPUs also have gained more units to process pixels and vertices in parallel: the GeForce 6800 Ultra, for example, incorporates 6 vertex shader units and 16 pixel pipelines.

Despite these performance advances, rendering complex scenes is still more difficult than simply dumping all geometry onto the GPU and forgetting about it. The simple approach tends to fail either because the generated GPU workload turns out to be excessive, or because the associated CPU overhead is prohibitive. This part of the book discusses the challenges today's games face in rendering complex geometric scenes.

Chapter 1, "Toward Photorealism in Virtual Botany" by **David Whatley** of Simutronics Corporation, provides a holistic view on how to render nature scenes. It explains the multitude of different techniques, from scene management and rendering various plant layers to post-processing effects, that Simutronics' *Hero's Journey* employs to generate complex and stunning visuals.

Rendering terrain is a good example of why simply dumping all available data to the GPU cannot work: the horizon represents a near-infinite amount of vertex data and thus workload. **Arul Asirvatham** and **Hugues Hoppe** of Microsoft Research use vertex texture fetches for a new highly efficient terrain-rendering algorithm. Their technique avoids overloading the GPU even as it shifts most work onto the GPU and away from the CPU, which too often is the bottleneck in modern games. **Chapter 2, "Terrain Rendering Using GPU-Based Geometry Clipmaps,"** provides all the implementation details.

NVIDIA Ex1023
Page Q-53

As already mentioned, another way to increase geometric complexity is to increase the number of visible objects in a scene. The straightforward solution of drawing each object independently of the others, however, quickly bogs down even a high-end system. It is much easier to efficiently draw ten objects that are one million triangles each, than it is to draw one million objects that are ten triangles each. **Francesco Carucci** of Lionhead Studios faces this very problem while developing *Black & White 2*, the sequel to Lionhead's critically acclaimed *Black & White*. **Chapter 3, "Inside Geometry Instancing,"** describes his solution: a framework of instancing techniques that applies to legacy GPUs as well as to GPUs supporting DirectX 9's instancing API. **Jon Olick** of 2015 provides further optimizations to the instancing technique that prove beneficial for 2015's title *Men of Valor: Vietnam*. Jon describes his findings in **Chapter 4, "Segment Buffering."**

Also, as games incorporate more and more data—more complex scenes of more complex meshes rendered in multiple, disparate passes supporting the gamut of differing functionality from legacy to current high-end GPUs—managing this glut of data efficiently becomes paramount. **Oliver Hoeller** and **Kurt Pelzer** of Piranha Bytes are currently working on Piranha Bytes' *Gothic III* engine. They share their solutions in **Chapter 5, "Optimizing Resource Management with Multistreaming."**

The best way to render lots of geometry to create geometric complexity is to avoid rendering the occluded parts. **Michael Wimmer** and **Jiří Bittner** of the Vienna University of Technology explore how best to apply that idea in **Chapter 6, "Hardware Occlusion Queries Made Useful."** Occlusion queries are a GPU feature that provides high-latency feedback on whether an object is visible or not after it is rendered. Unlike earlier occlusion-query culling techniques, Michael and Jiří's algorithm is pixel-perfect. That is, it introduces no rendering artifacts, generates a near-optimal set of visible objects to render, does not put unnecessary load on the GPU, and has minimal CPU overhead.

Similarly, increasing geometric detail only where visible and simplifying it when and where it isn't visible is a good way to avoid excessive GPU loads. View-dependent and adaptive subdivision schemes are an appealing solution that the

NVIDIA Ex1023
Page Q-54

offline-rendering world already employs to render their highly detailed models to subpixel geometric accuracy. Subdivision surfaces have not yet found a place in today's real-time applications, partly because they are not directly supported in graphics hardware. Rendering subdivision surfaces thus seems out of reach for real-time applications. Not so, says **Michael Bunnell** of NVIDIA Corporation. In **Chapter 7**, Michael shows how his implementation of **"Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping"** is already feasible on modern GPUs and results in movie-quality geometric detail at real-time rates.

Finally, faking geometric complexity with methods that are cheaper than actually rendering geometry allow for higher apparent complexity at faster speeds. Replacing geometry with textures that merely depict it used to be an acceptable trade-off—and in the case of grates and wire-mesh fences, often still is. Normal mapping is a more sophisticated fake that properly accounts for lighting information. Parallax mapping is the latest craze that attempts to also account for intra-object occlusions. **William Donnelly** of the University of Waterloo one-ups parallax mapping: he describes **"Per-Pixel Displacement Mapping with Distance Functions"** in **Chapter 8**. Displacement mapping provides correct intra-object occlusion information, yet minimally increases computation cost. His technique gives excellent results while taking full advantage of the latest programmable pixel-shading hardware. Even better, it is practical for applications today.

Matthias Wloka, NVIDIA Corporation

NVIDIA Ex1023
Page Q-55



Chapter 1

Toward Photorealism in Virtual Botany

David Whatley
Simutronics Corporation

Rendering natural scenes in real time, while leaving enough CPU and GPU resources for other game-engine requirements, is a difficult proposition. Images of botany require a great deal of visual depth and detail to be convincing. This chapter describes strategies for rendering more photorealistic natural scenes in a manner that is friendly to real-time game engines. The methods presented here work together to create a convincing illusion of grassy fields teeming with plants and trees, while not overwhelming either the CPU or the GPU. These techniques are used in Simutronics' *Hero's Journey*, as shown in Figure 1-1.

We begin by describing the foundation for managing scene data in large outdoor environments. Next, we provide details on how to maximize throughput of the GPU to achieve the required visual density in grass. Then we expand on these techniques to add ground clutter and larger-scale botany, such as trees. Finally, we tie the visuals together with shadowing and environmental effects.

1.1 Scene Management

Game engines must manage their rendering techniques to match the scope of the environment they hope to visualize. Game levels that feature nature scenes, made up of thousands of trees and bushes and perhaps millions of blades of grass, present significant data management problems that must be solved to allow rendering at interactive frame rates.

NVIDIA Ex1023
Page Q-56



Figure 1-1. Babbling Brook: A Nature Scene from *Hero's Journey*

Rendering a virtual nature scene convincingly is both an artistic and a technical challenge. We can approach the rendering of nature much like a painter: break down the elements into layers and treat each layer independently to ultimately create a unified whole. For example, a layer of grass, a layer of ground clutter, a layer of trees, and so on. All these layers share some common properties, which we can leverage to compress our data representation.

Our goal is to travel the game camera over long distances of convincing outdoor scenes without having to dedicate excessive memory resources to managing the task. With guided deterministic random-number generation, we have an algorithm that can "plant" all of the elements of nature in a reasonable manner while achieving the same visual results each time we revisit the same spot on the map. In an online game, everyone would see the same thing right down to the placement of a blade of grass without this placement being permanently stored in memory.

1.1.1 The Planting Grid

We establish a world-space fixed grid around the camera to manage the planting data for each layer of plants and other natural objects. Each grid cell contains all of the data

NVIDIA Ex1023
Page Q-57

to render its layer in the physical space it occupies. In particular, the cell data structure stores the corresponding vertex and index buffers, along with material information to represent what is drawn.

For each layer of botany, we establish a distance from the camera that the layer needs to generate visuals; this determines the size of our virtual grid. As the camera moves, the virtual grids travel with it. When a grid cell is no longer in the virtual grid, we discard it and add new cells where needed to maintain our full grid structure. As each cell is added, a planting algorithm is used to fill in the layer with the appropriate data for rendering. See Figure 1-2.

1.1.2 Planting Strategy

For each cell that is filled with natural objects, we need to pick suitable spots on the ground where those objects are placed. The heuristic used to choose these spots depends on the type of object being placed. Generally, we would like to pick random spots, at some desired density, and then see if the corresponding points on the ground are acceptable for what we are planting. In our implementation, a ground polygon's material determines what layers are applicable.

The obvious approach is to randomly cast rays straight down at the ground within the volume of the cell. Each time we hit a polygon, we check to see if it is suitable (Can grass be planted here? Is the slope too severe?). If we succeed, then we have a planted point. We continue until we reach the proper density.

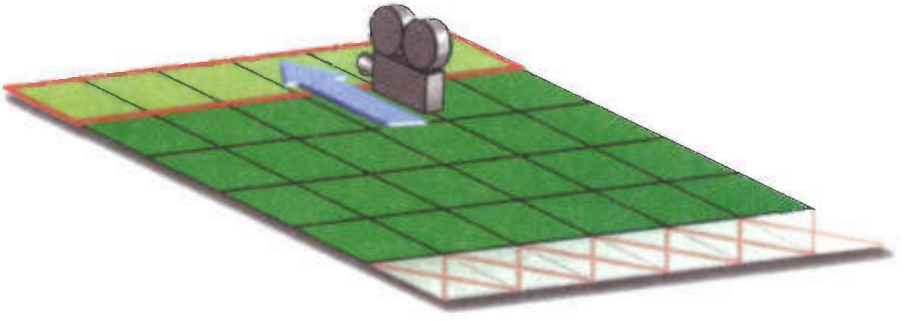


Figure 1-2. The Virtual Grid
Each layer consists of a fixed-size grid that is world-space aligned. The dark green cells represent active cells. The camera is moving forward, and the cells marked with an X are being discarded as new cells (shown in light green) are added to maintain our virtual grid size. A useful improvement in the implementation is to pool the grid cells and simply recycle them, because a new cell is always added when an old one is discarded.

NVIDIA Ex1023
Page Q-58

This approach yields good results but has significant problems. First, in grid cells where there are few suitable places to plant (for example, just the top of a polygon that is marked for grass), we can burn inordinate amounts of CPU time trying to randomly achieve our density requirement. So in the worst case, we must abandon our search if we reach some maximum limit of planting attempts. Second, we cannot handle overlapping terrain (such as a land bridge) with this approach.

A better approach is to collect all of the polygons that intersect the cell, discard all polygons inappropriate for planting, and then scan-convert them to find suitable spots for planting. This is similar to rasterizing a polygon for rendering, but instead each "pixel" of our traversal is a world-space potential planting point. We must be careful to keep the scan conversion rate appropriate to the density, while not exceeding the boundaries of the triangle. Further, at each planting point we select, it is important to offset along the plane of the polygon by some suitable random distance to eliminate repeating patterns. All of these values are adjustable coefficients that should be determined during design time. In our implementation, the designer can interactively tweak these values to achieve the desired result for the layer.

Finally, when scan-converting we also must take care to clip to the polygon edges (when offsetting) as well as to the cell's border, because the polygon may extend beyond it (and another cell is managing the planting there).

Planting in this manner can take place in real time or as part of offline level preprocessing. In the latter case, the grass planting spots should be stored in a highly compressed form; the data should be uncompressed at run time as each cell is added to the set of potentially visible cells by the moving camera.

1.1.3 Real-Time Optimization

If this planting operation is done in real time, care must be taken to ensure that planting is a fast operation. Collecting polygons in a grid cell can be done quickly by using an AABB tree or a similar data structure. Because many cells may need to be planted suddenly due to continuous camera movement, it is also effective to queue up this task so that we spend only a relatively fixed amount of CPU on the task for each frame. By extending the size of the grid, we can be reasonably sure that all the appropriate planting will take place before the contents of the cell come into view.

NVIDIA Ex1023
Page Q-59

1.2 The Grass Layer

Achieving interactive frame rates for endless fields of grass requires a careful balance of GPU techniques and algorithms. The key challenge is to create a visual that has high apparent visual complexity at relatively low computational and rendering cost. Doing so creates a convincing volume of grass. Here we introduce a technique similar to the one presented by Pelzer (2004) in "Rendering Countless Blades of Waving Grass." Our technique yields higher-quality and more-robust results at a reduced GPU and CPU burden. Figure 1-3 shows a scene rendered with our technique.

Obviously, drawing each grass blade is out of the question. But we can create such an illusion with clumps of grass, which are best represented by camera-facing quads with a suitable grass texture. Billboards of this nature create the illusion of volume at a minimal cost. However, a large field of grass can still require an excessive number of draw calls, so we must carefully structure our usage of the GPU to achieve sufficient volume and density.



Figure 1-3. A Convincing Grass Layer
To be convincing, the illusion of volume must be maintained even though the grass is drawn using camera-facing quads.

NVIDIA Ex1023
Page Q-60

GPUs work best when they are presented with large batch sizes to draw at once. Therefore, our goal is to figure out how to draw fields of grass with a relatively small number of draw calls to the API. The naive approach is to draw one billboard at a time. Instead, what we want is to draw as many as is practical in one draw call.

To achieve this, we use a technique whereby we create a vertex and an index buffer and fill it with a large number of grass billboards. We then draw all these billboards in one call. This algorithm is similar to speeding up a CPU loop by unrolling it.

For our purposes, each layer of grass—that is, all grass that uses the same texture and other parameters—is represented by a vertex and an index buffer pair per grid cell, as shown in Figure 1-4. For each clump of grass (or billboard) we plant, we write its positions into the vertex buffer and update the index buffer accordingly. We need four vertices and six indices per billboard. For each vertex, we set the position to the point where we have planted the grass clump. This means that all four vertices of a billboard have an identical position, but we offset this position in the vertex shader to create the proper camera-facing quad shape. Alternatively, if the grass texture fits within a triangular shape, we can save processing one vertex each. Even better, at this point, indices become unnecessary and can be skipped altogether without loss of performance; no vertex is ever reused out of the post-transformation-and-lighting cache when rendering this sort of triangle soup.

Once the vertex buffer is created and sent to video memory, we can draw each grid cell's worth of botany with a single draw call. On the GPU, we use a vertex shader to

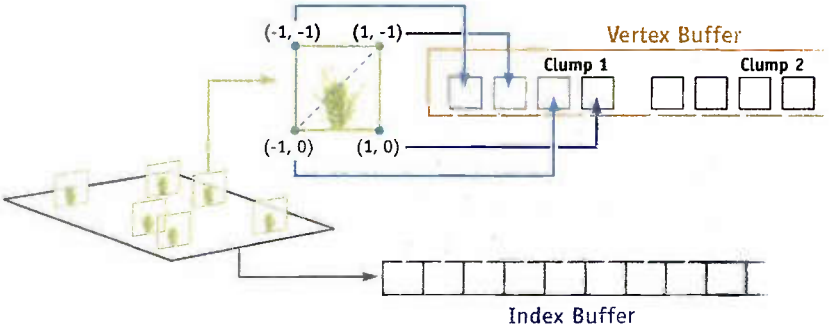


Figure 1-4. Structures for Drawing Each Grid Cell
Each grid cell contains a vertex and an index buffer into which we unroll our grass quads. The position of each of the quad's four vertices is set to the planting point (where the grass roots enter the ground) and the vertex shader moves the vertices into proper position. The entire grid cell's worth of grass can now be drawn in one call.

NVIDIA Ex1023
Page Q-61

offset each of the vertices so that they form a screen-aligned quad. Since each vertex moves in a different direction, we have to identify which vertex forms what corner of the quad. To do this, we augment our vertex data with two additional floats that contain -1, 0, or 1. The first float is for the x direction on the screen, and the second is for the y. We multiply this factor by our scale in x and y to offset as necessary. Additionally, we can randomly set all -1 and 1 values to slightly different values (such as 0.98 or -1.2) to add size variety to each grass clump.

Though we intend to move the vertex in screen space, we do all our work in world space so that we get the perspective effect for free. To do this, we provide our vertex shader with a vector that points to the right of the camera and another that points up from the camera. Simple math moves the vertex into the correct position:

```
// For each vertex, we pass a -1, 0, 1 value for x, y, which determines
// how it is moved by the right and up vectors of the camera. This
// we pass in as a texture coordinate (inTexCoord2).

Out.Pos = Input.inPos + (Input.inTexCoord2.x * RightVector) +
                        (Input.inTexCoord2.y * UpVector);
```

Our approach differs from Pelzer's because we use camera-facing billboards instead of having three quads per clump and doing no screen alignment. The screen-facing billboards create a constant depth at all view angles (even when looking down), whereas the three-quad clump approach progressively breaks down as the camera looks more directly down at the grassy field. In a third-person camera view, typical of many types of games and simulations, this is not an uncommon camera angle.

1.2.1 Simulating Alpha Transparency via Dissolve

When rendering grass, we want to use transparency to improve the visual blending and fade out at a distance near the boundary of our virtual grid. However, alpha-based transparency is far from ideal because it requires sorting, which is slow. Although there are techniques that take advantage of the chaotic nature of grass to minimize some of the sorting (Pelzer 2004), we can actually eliminate it altogether.

To do this, we adopt a dissolve effect (also called the screen-door effect) instead of alpha blending to simulate translucency. We accomplish this by modulating the alpha channel of our grass texture with a noise texture. Then we use alpha test to eliminate pixels from rendering. By sliding the alpha test value from 0 to 1, the texture appears to dissolve. The process is shown in Figure 1-5. SpeedTreeRT, a commercial package for real-time foliage creation, was the first to use this technique to blend between levels of detail

NVIDIA Ex1023
Page Q-62

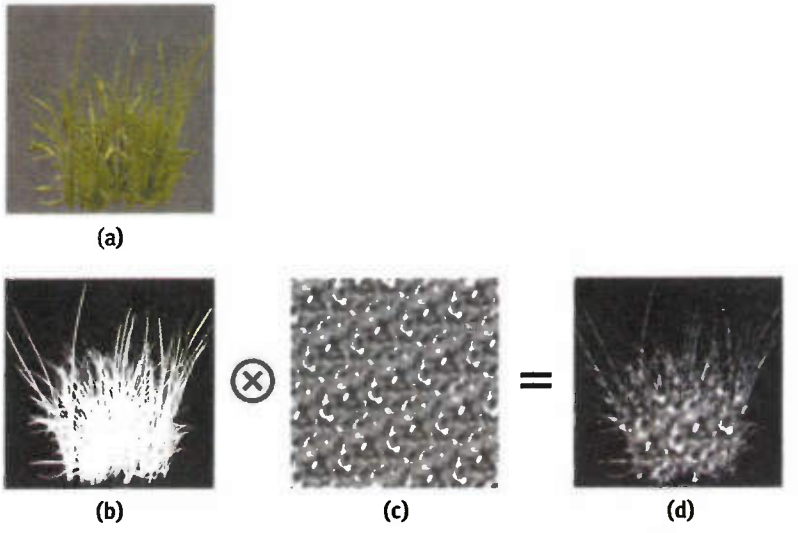


Figure 1-5. Components of the Grass Texture
Grass texture with (a) the diffuse texture, (b) the artist-created alpha channel, (c) the Perlin noise texture, and (d) the result of multiplying the artist alpha by the Perlin noise. This can be done in the pixel shader, or the fixed-function pipeline, or as part of the authoring process by generating mip levels that have a progressively more transparent pattern, to create fade with a fixed alpha-test value.

with alpha testing enabled. Simutronics has licensed IDV's SpeedTreeRT for *Hero's Journey*, and techniques drawn or adapted from this commercial foliage toolkit are noted as such in this chapter.

The benefit of this technique is that the alpha test is a fast operation and order-independent. We no longer need to sort, yet grass appears to fade out at a distance. Although a dissolve does not look nearly as good as true alpha translucency under normal circumstances, we can exploit the fractal properties of nature to completely mask any visual artifacts of the screen-door technique. Experimentally, we have found that if we use a Perlin noise texture (Perlin 2002) rather than a random noise texture, the dissolve effect matches the environment well enough to be nearly as good as alpha translucency.

One problem we have, however, is that the alpha test value we select needs to be based on the distance of each billboard from the camera. However, we can only use exactly one alpha test value for each batch (each grid cell) even though the batch is filled with grass clumps at varying distances; this is because we are rendering the entire grid cell in one draw call. Because we want each grass billboard to fade out precisely based on its distance from the camera, we select a fixed alpha test value and instead manipulate the

NVIDIA Ex1023
Page Q-63

alpha channel in the shaders, linearly interpolating them toward full alpha based on the distance from the camera and the maximum range at which we want grass to start being visible. We do this by adjusting the alpha component of the output color of the vertex shader, which then attenuates the alpha of the texel in the pixel shader (this can also be done with the fixed-function pipeline).

An additional pass can be made over grid cells nearest the camera, drawing with alpha blending to feather the edges of the grass blades into the scene, thus eliminating any harsh edges from blades of grass close to the camera. This can achieve marginally better compositing at the expense of another pass. But because alpha test is still used, relatively few pixels are written to the back buffer (just the edges of the blades, at most). Alpha blending and alpha testing work well together; it is often a good idea to experiment with both to achieve the best illusion of depth and volume.

1.2.2 Variation

To increase the realism of our grass, we want to introduce as much variety as we can without impeding frame rate. One approach is to use a variety of images for grass, but our batching approach limits us to one texture per draw call. Fortunately, we can use a larger texture with different variations of grass arranged on them. During the vertex building, we can adjust the UV coordinates to select different subregions of the texture (that is, we can build a texture atlas; see NVIDIA 2004 for more on this technique). It is easy to have, for instance, four different grass clump variations on one texture. Plus, as we unroll grass quads into the vertex buffer, we can randomly flip the U coordinate to mirror the image. Be sure to allow space between your images so that mipmapping does not introduce texel-bleed artifacts.

Each billboard can also carry along color information. This is very useful for tinting a grayscale texture or doing subtle color shifting in the vertex shader, if you also establish a color for each cluster when planting. We have found that Perlin noise works here as well. It is easy, for example, to tint grass from a healthy green to a dying brown to impart broad color variations and break up the repetitiveness of the grass. See Figure 1-6.

1.2.3 Lighting

Lighting plays an important role in how grass looks. For billboard grass, we want to make sure that our grass is lit like the ground underneath. Because the ground naturally undulates, and thus picks up different angles of sunlight, we want to simulate this by attenuating the brightness of the grass. To do so, we need to know the angle of the

NVIDIA Ex1023
Page Q-64



Figure 1-6. Using RGB Information to Increase Realism
Each planting of grass carries RGB information in the vertices. In this scene, the color values are derived from a Perlin noise function to simulate grass that is greener and patches that are a less-healthy brown.

ground on which the grass is sitting. An easy solution is to pass along this information in the vertex definition as another vector. During planting, we determine the normal of the polygon on which we are planting grass and carry this along in our grass billboard definition. With this approach, the vertex shader can do the same lighting calculation as for the polygon underneath the grass and attenuate its color to match. On hilly terrain, this causes the grass to have the subtle angle-to-the-light cues that the ground has.

Unfortunately, this approach leads to a faceted shading of the grass even though the ground polygons are likely smooth shaded (such as with Gouraud shading). To get around this discrepancy, the normal that is passed through the vertex shader must itself be smoothly interpolated during the planting process.

If the sun angle is dynamic, a simplification is to assume that the ground normal is roughly straight up and then carry out the lighting based on this normal and the light angle. In this way, we do not have to compute or carry the ground polygon normal in the vertex definition. There is a quality trade-off here, but this approach was good enough for our application.

NVIDIA Ex1023
Page Q-65

1.2.4 Wind

Grass comes alive when wind affects it. Offsetting the top two vertices of the grass quad each frame causes the quad to wave in the wind. We compute this offset by using a sum of sines approximation, similar to computing surface undulation for water (Finch 2004). The trick to this is to carry a blend weight in the vertex definition, where the top two vertices of the grass quad are set to 1 and the bottom two to 0. Then we multiply our wind scale factor by this value, so that the bottom vertices remain firmly planted in the ground. For additional variation, we can randomize the top two vertex weights somewhat during planting. This simulates some grass being more or less rigid.

Grass blades often change their orientation to the light as they wave in the wind, causing them to brighten and darken. We can use the wind term to augment the lighting to simulate this effect. This greatly improves the visual effect of the wind, even for grass clumps at a distance where the physical waving deformation becomes subpixel in size.

Note, however, that the wind factors should not be allowed to deform the grass quads too much. If they do, the resulting deformation will appear comical rather than realistic. Subtlety is the key.

1.3 The Ground Clutter Layer

Ground cover consists of more than just waving fields of grass. Twigs, small plants, rocks, and other debris complete the illusion of natural complexity. Some of these can be represented as billboards just as grass is, but the richness of the environment is enhanced when we mix in an assortment of geometric objects, as well.

Just as we did with grass billboards, we unroll our 3D mesh data into vertex and index buffers for each grid cell, which can then be drawn with a single call. We must group our ground clutter into layers that use the same textures and shaders. We can apply random transforms to vary their size and orientation as we pick our planting points, but the transforms must vary depending on the nature of the mesh: an upside-down rock is okay, but an upside-down bush is not. For additional variety, we can pass RGB information to tint the objects just as we did with the grass polygons.

The dissolve technique for handling order-independent transparency effects works exactly the same for 3D meshes as it does for billboards. We modulate the alpha channel of the texture by our Perlin noise texture and use our distance from the camera to attenuate. Then alpha test dissolves the 3D meshes the same way it did with the grass billboards.

NVIDIA Ex1023
Page Q-66

Each vertex can be given a weighting value, which allows us to apply the same wind math to the 3D as we did with the billboards. Obviously, we want objects such as rocks and twigs to be rigid, but leafy plants can have artist-driven weights to achieve the proper effect. Figure 1-7 shows an example of a scene with ground clutter.



Figure 1-7. Using Ground Clutter to Add Dense Detail
Here ground clutter is used to add dense detail to the shoreline of a river. The rocks, branches, and twigs are all 3D meshes unrolled into vertex buffers per grid cell. The reeds are rendered with the grass-rendering technique.

1.4 The Tree and Shrub Layers

The trunk and primary branches of a tree should be modeled as 3D meshes. Secondary branches can be simulated with billboards to add visual complexity. For trees that are leafy, we can use techniques similar to the one we used for grass to create clumps of leaves from camera-facing billboards.

The following approach to rendering trees and shrubs is based on SpeedTreeRT, which provides all of the real-time rendering information for tree assets and includes SpeedTreeCAD for parametric authoring of trees (IDV 2004). The actual rendering of

NVIDIA Ex1023
Page Q-67

trees is still the responsibility of the game engine, which gives developers a lot of flexibility in implementation.

Because trees need to maintain their basic volume over long distances but are expensive to render in great detail, a level-of-detail (LOD) strategy must be employed. As a tree recedes into the distance, larger but fewer leaf clump billboards can be used. For the larger billboards, we use a different texture that shows more but smaller leaves.

For efficiency, all of the textures related to a tree should be consolidated into a single texture, as shown in Figure 1-8. In fact, it is even preferable to have multiple tree textures packed into one texture, so we can draw more of them in one call.

At some suitable distance, we eventually want to represent a tree with a camera-facing billboard. This can be difficult when the tree's profile is asymmetrical. To get around



Figure 1-8. Storing Multiple Leaf Cluster Images in a Single Texture
A single texture contains many leaf cluster images, at different resolutions for level of detail, plus the billboards for a tree at a distance. Building a composite texture in this way makes it possible to draw many trees with fewer texture changes. This composite map was generated using SpeedTreeCAD.

NVIDIA Ex1023
Page Q-68

the problem, we can produce tree images for the billboard at various angles and then blend between them based on the angle of the tree and the camera.

If trees are placed manually, we have found that it is best to give the level designer fine control over LOD transition points for each instance. In areas where trees are dense, you can get away with LOD transitions fairly near the camera. But in other places, with long lines of sight but fewer trees, it's easier to maintain their high detail and avoid visual artifacts.

Shrubs and fronds can be handled as just another type of tree using many of the same techniques. For example, a leafy bush is simply a tree with a small or nonexistent trunk. Further, you can turn a tree upside down, turn off the leaves, and get an elaborate exposed root system to marry up with a normal tree.

Trees can be authored in a standard modeling package, but artists need some mechanism to specify where leaf points and branch billboards go. This can complicate exporting. The actual rendering of the trees is still the responsibility of the game engine, which gives developers a lot of flexibility in implementation.

1.5 Shadowing

Because we pass an RGB tint for grass and ground clutter, we can choose a dark tint for areas in shadow. This requires us to know whether or not each planted item is in shadow. To be effective in natural environments, this sort of shadowing needs to only grossly approximate the correct shadowing. See Figure 1-9.

One approach is to make the shadow determination when the planting occurs. Simply cast a shadow feeler ray from the planting position toward the dominant light source (the sun) and see if there is an intersection. If so, adjust the RGB values toward the scene's ambient color. Remember that a shadow feeler ray cast is concerned only about any intersection (not just the closest) so it can be much more efficient than a standard collision ray cast.

Soft shadows (technically called *antialiased* shadows in this context) can be achieved by casting more than one shadow feeler. Figure 1-10 shows how this works. By offsetting each ray start position slightly, three or five ray casts from a given spot can be performed. The fraction of hits is used to attenuate the light between the diffuse sun lighting and the scene's ambient lighting. Widening the offsets increases the softness of the shadowing.

NVIDIA Ex1023
Page Q-69



Figure 1-9. Grass Shadowed by a Tree Root
In this scene, grass is shadowed by an object (an oversized tree root). Having multiple shadow feelers per grass clump helps soften the transitions between shadowed and illuminated areas.

These sorts of shadows are not dynamic, but they can be recomputed fairly quickly at intervals for slow-moving light sources (such as the traveling sun). In general, they provide sufficient visual cues to cause the scenery to seem more lifelike.

Special approximation techniques can be used to cause shadow feeler hits when casting through the bulk of a tree. Instead of looking for intersections with individual leaves, simply collide with the spherical volume of the leafy part of the tree or the cylinder of the trunk. For the leafy part, use a random function to determine if rays intersect based on the density of the leaves. Although this shadowing technique is crude, accurate solutions are not visually distinct.

If planting is precomputed as an offline process, then shadow fidelity can be greatly enhanced. One possibility beyond shadow feelers is to look at the texel of a light map to determine shadowing. This can be difficult in real time if the light map is not in system memory.

NVIDIA Ex1023
Page Q-70

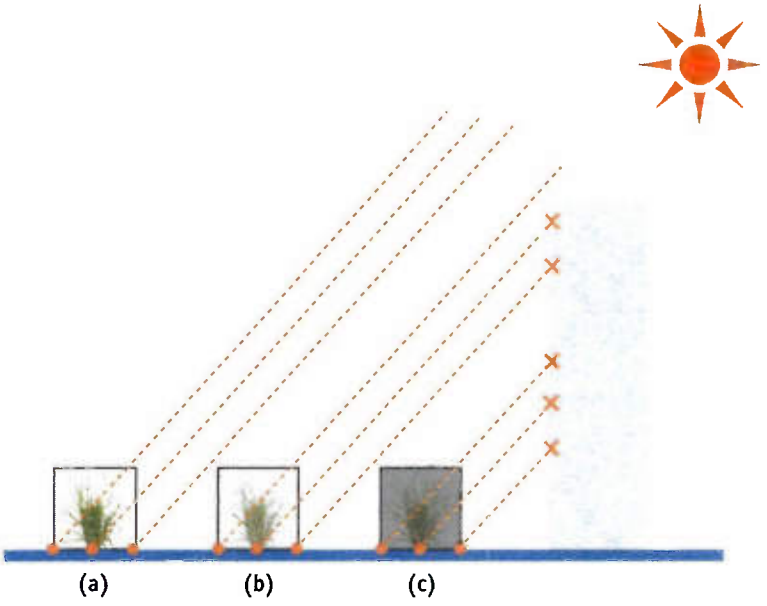


Figure 1-10. Visibility Testing Using Ray Casts
Shadow feeler ray casts are projected from the planting point toward the light source (which is directional in this case). Any collision indicates that the point is in shadow. To get a softer shadow, project additional offset shadow feelers and use the fraction of hits to determine how much to tint toward the ambient lighting. Care must be taken so that the shadow feelers do not immediately intersect the ground plane at the planting point.

1.6 Post-Processing

Natural environments react to sunlight in ways that are hard to simulate in real time. Like most interactive techniques, the goal is to find methods that achieve a reasonable, if not accurate, result. Using post-processing effects, we can tie the visuals together to achieve a superior environment.

Full-scene post-processing glow is useful for magical and high-tech effects in games, and extending this simple technique for natural effects has proven effective. Real-time glow techniques provide a way to simulate blooming—as well as create a natural softening effect (brought about by a Gaussian blur)(James and O'Rorke 2004). However, because glow is an additive effect, care must be taken to account for this when authoring textures. It is easy to get carried away and overbrighten a scene. As with all effects, just because a little is good, more isn't necessarily better.

NVIDIA Ex1023
Page Q-71

1.6.1 Sky Dome Blooming

A textured sky dome provides a rich opportunity to use glow to our advantage. We can set aside the alpha channel of the sky dome for the artist to define areas that are more luminous. This allows the artist to have a lot of control over how the sun and the clouds interact. When rendering the sky dome, simply apply the alpha layer to the glow channel. Figure 1-11 shows an example.

A sky dome with a good amount of glow blooms around the delicate structure of tree leaves and branches. This yields a particularly realistic touch that is almost immediately noticeable, as shown in Figure 1-12. Because the sky glow is controlled by the artist, this technique is particularly effective and finely tuned at no additional cost.

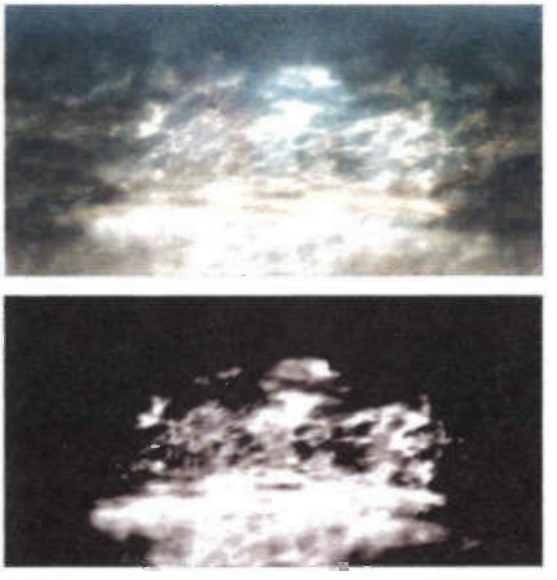


Figure 1-11. Sky Dome Diffuse Texture and Glow Component



Figure 1-12. Varying Amounts of Sky Dome Glow
From left to right, no glow, moderate glow, and a lot of glow. Notice that it glows only in the areas of the sky designated by the artist in the alpha channel of the sky dome texture.

NVIDIA Ex1023
Page Q-72

1.6.2 Full-Scene Glow

Cinematography of natural environments is often enhanced by a technique in which diffusing gauze (cheesecloth) is placed over the camera lens. If post-processing glow is being used, we can simply clear the glow channel to an adjustable, nonzero value to create a full-scene glow. Because the glow is a simulation based on a Gaussian blur, this causes the whole scene to appear to be diffused; also, bright areas bloom slightly without having to resort to more expensive HDR effects. For natural outdoor scenes, this approach can greatly mitigate the harsh computer-generated look of polygons for no extra GPU cost (because full-scene glow processing is already occurring). See Figure 1-13.



Figure 1-13. The Effect of Full-Scene Glow
The scene on the left is rendered with no glow; the scene on the right has a large amount of full-scene glow. Besides the obvious brightening effect, the glow's Gaussian blur causes the scene to appear softer and diffused.

1.7 Conclusion

Rendering nature in a convincing way always adds visual drama to a game. Figure 1-14 illustrates how these techniques can be very effective at conveying a sense of the inherent complexity and grandeur of natural scenery.

1.8 References

Finch, Mark. 2004. "Effective Water Simulation from Physical Models." In *GPU Gems*, edited by Randima Fernando, pp. 5–29. Addison-Wesley.

IDV. 2004. SpeedTreeRT API and SpeedTreeCAD Windows application. **http://www.idvinc.com/html/speedtreert.htm**

James, Greg, and John O'Rorke. 2004. "Real-Time Glow." In *GPU Gems*, edited by Randima Fernando, pp. 343–362. Addison-Wesley.

NVIDIA Ex1023
Page Q-73



Figure 1-14. Creating the Illusion of a Lush Landscape
Here all techniques work together to produce a suitable illusion of a lush landscape. As the camera travels over this expansive game level, the level of detail remains high near the camera at interactive frame rates.

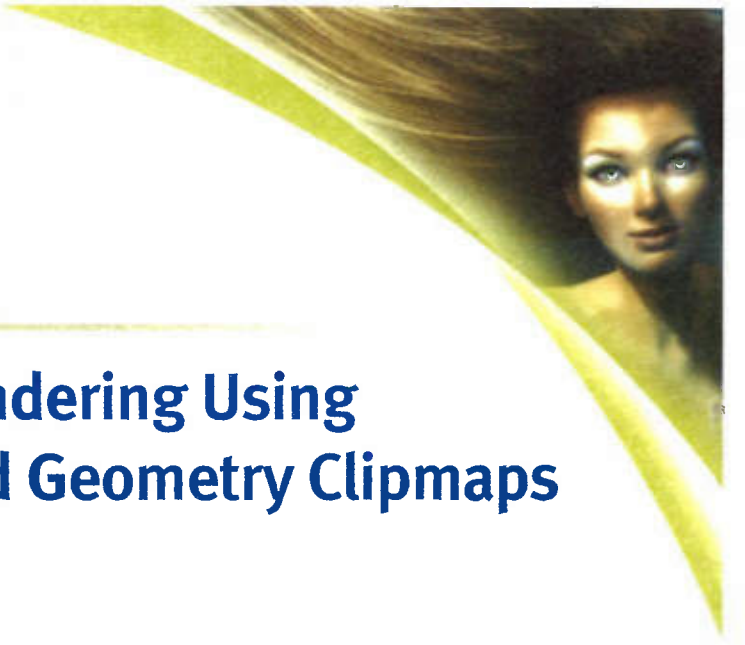
NVIDIA Corporation. 2004. "Improve Batching Using Texture Atlases." SDK white paper. **http://download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf**

Pelzer, Kurt. 2004. "Rendering Countless Blades of Waving Grass." In *GPU Gems*, edited by Randima Fernando, pp. 107–121. Addison-Wesley.

Perlin, Ken. 2002. "Improving Noise." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)* 21(3), pp. 681–682. **http://mrl.nyu.edu/~perlin/paper445.pdf**

I would like to thank the entire Simutronics Hero's Journey *art and programming team for their invaluable contributions to this effort, especially 3D artists Richard Amsinger and Kyle Knight, whose work is featured in this chapter. Additionally, I would like to thank Dave Dean and Bryan Cool for their programming wizardry, which contributed a great deal to the techniques presented in this chapter. Last, I want to thank our art director, Tracy Butler, for providing the illustrations that help clarify many of the concepts in this chapter.*

NVIDIA Ex1023
Page Q-74



Chapter 2

Terrain Rendering Using GPU-Based Geometry Clipmaps

Arul Asirvatham
Microsoft Research

Hugues Hoppe
Microsoft Research

The geometry clipmap introduced in Losasso and Hoppe 2004 is a new level-of-detail structure for rendering terrains. It caches terrain geometry in a set of nested regular grids, which are incrementally shifted as the viewer moves. The grid structure provides a number of benefits over previous irregular-mesh techniques: simplicity of data structures, smooth visual transitions, steady rendering rate, graceful degradation, efficient compression, and runtime detail synthesis. In this chapter, we describe a GPU-based implementation of geometry clipmaps, enabled by vertex textures. By processing terrain geometry as a set of images, we can perform nearly all computations on the GPU itself, thereby reducing CPU load. The technique is easy to implement, and allows interactive flight over a 20-billion-sample grid of the United States stored in just 355 MB of memory, at around 90 frames per second.

2.1 Review of Geometry Clipmaps

In large outdoor environments, the geometry of terrain landscapes can require significant storage and rendering bandwidth. Numerous level-of-detail techniques have been developed to adapt the triangulation of the terrain mesh as a function of the view. However, most such techniques involve runtime creation and modification of mesh structures (vertex and index buffers), which can prove expensive on current graphics architectures. Moreover, use of irregular meshes generally requires processing by the CPU, and many applications such as games are already CPU-limited.

NVIDIA Ex1023
Page Q-75

The geometry clipmap framework (Losasso and Hoppe 2004) treats the terrain as a 2D elevation image, prefiltering it into a mipmap pyramid of L levels as illustrated in Figure 2-1. For complex terrains, the full pyramid is too large to fit in memory. The geometry clipmap structure caches a square window of $n \times n$ samples within each level, much like the texture clipmaps of Tanner et al. 1998. These windows correspond to a set of nested regular grids centered about the viewer, as shown in Figure 2-2. Note that the finer-level windows have smaller spatial extent than the coarser ones. The aim is to maintain triangles that are uniformly sized in screen space. With a clipmap size $n = 255$, the triangles are approximately 5 pixels wide in a 1024×768 window.

Only the finest level is rendered as a complete grid square. In all other levels, we render a hollow "ring," which omits the interior region already rendered at finer resolutions.

As the viewer moves, the clipmap windows are shifted and updated with new data. To permit efficient incremental updates, the clipmap window in each level is accessed toroidally, that is, with 2D wraparound addressing (see Section 2.4).

One of the challenges with the clipmap structure is to hide the boundaries between successive resolution levels, while at the same time maintaining a watertight mesh and avoiding temporal popping artifacts. The nested grid structure of the geometry clipmap provides a simple solution. The key idea is to introduce a transition region near the outer perimeter of each level, whereby the geometry and textures are smoothly morphed to interpolate the next-coarser level (see Figure 2-3). These transitions are efficiently implemented in the vertex and pixel shaders, respectively.

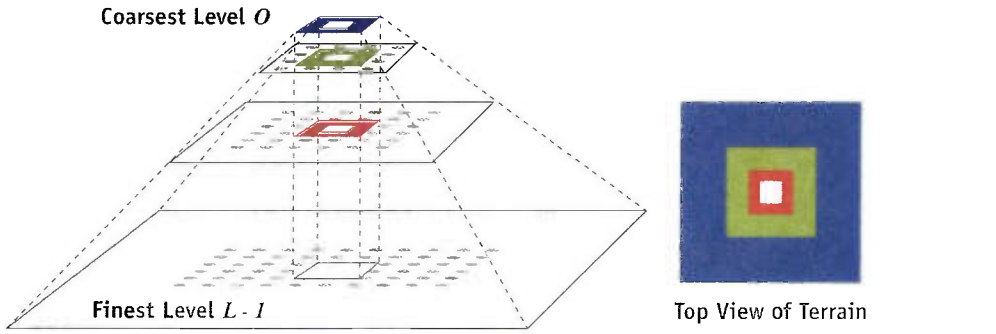


Figure 2-1. How Geometry Clipmaps Work
Given a filtered terrain pyramid of L levels, the geometry clipmap caches a square window at each resolution level. From these windows, we extract a set of L nested "rings" centered about the viewer. The finest-level ring is filled in.

NVIDIA Ex1023
Page Q-76

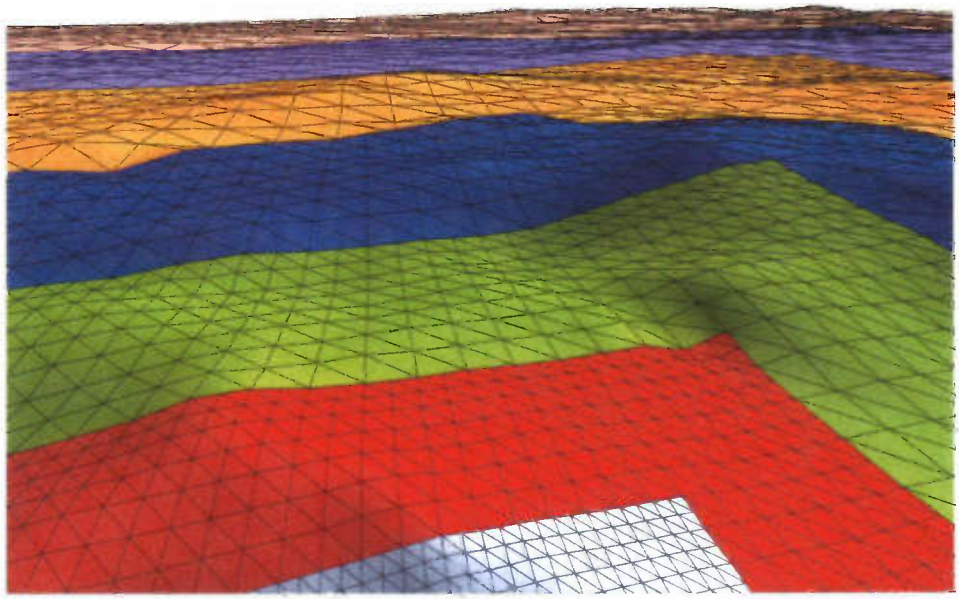


Figure 2-2. Terrain Rendering Using a Coarse Geometry Clipmap
Illustration of terrain rendering using a coarse geometry clipmap (size n = 31, L = 10). Each colored ring is formed from a different clipmap level.

The nested grid structure of the geometry clipmap also enables effective compression and synthesis. It allows the prediction of the elevation data for each level by upsampling the data from the coarser level. Thus, one need only store or synthesize the residual detail added to this predicted signal.

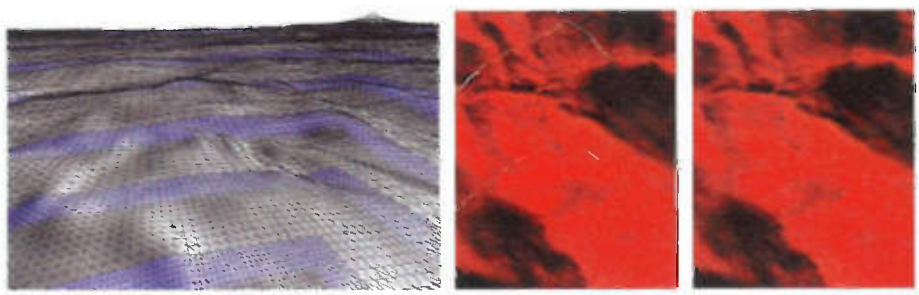


Figure 2-3. Achieving Visual Continuity by Blending Within Transition Regions
(a) Transition regions (in purple). (b) Without blending. (c) With blending.

NVIDIA Ex1023
Page Q-77

2.2 Overview of GPU Implementation

The original implementation of geometry clipmaps presented in Losasso and Hoppe 2004 represents each clipmap level as a traditional vertex buffer. Because the GPU currently lacks the ability to modify vertex buffers, that implementation required significant intervention by the CPU to both update and render the clipmap (see Table 2-1).

In this chapter, we describe an implementation of geometry clipmaps using vertex textures. This is advantageous because the 2D grid data of each clipmap window is much more naturally stored as a 2D texture, rather than being artificially linearized into a 1D vertex buffer.

Recall that the clipmap has L levels, each containing a grid of $n \times n$ geometric samples. Our approach is to split the (x, y, z) geometry of the samples into two parts:

- The (x, y) coordinates are stored as constant vertex data.
- The z coordinate is stored as a single-channel 2D texture—the *elevation map*. We define a separate $n \times n$ elevation map texture for each clipmap level. These textures are updated as the clipmap levels shift with the viewer's motion.

Because clipmap levels are uniform 2D grids, their (x, y) coordinates are regular, and thus constant up to a translation and scale. Therefore, we define a small set of *read-only*

Table 2-1. Comparison with Original CPU Implementation
Our implementation of geometry clipmaps using vertex textures moves nearly all operations to the GPU.

	Original Implementation[1]	GPU-Based Implementation
Elevation Data	In vertex buffer	In 2D vertex texture
Vertex Buffer	Incrementally updated by CPU	Constant!
Index Buffer	Generated every frame by CPU	Constant!
Upsampling	CPU	GPU
Decompression	CPU	CPU
Synthesis	CPU	GPU
Adding Residuals	CPU	GPU
Normal-Map Update	CPU	GPU
Transition Blends	GPU	GPU

1. Losasso and Hoppe 2004.

NVIDIA Ex1023
Page Q-78

vertex and index buffers, which describe 2D "footprints," and repeatedly instance these footprints both within and across resolution levels, as described in Section 2.3.2.

The vertices obtain their z elevations by sampling the elevation map as a vertex texture. Accessing a texture in the vertex shader is a new feature in DirectX 9 Shader Model 3.0, and is supported by GPUs such as the NVIDIA GeForce 6 Series.

Storing the elevation data as a set of images allows direct processing using the GPU rasterization pipeline (as noted in Table 2-1). For the case of synthesized terrains, all runtime computations (elevation-map upsampling, terrain detail synthesis, normal-map computation, and rendering) are performed entirely within the graphics card, leaving the CPU basically idle. For compressed terrains, the CPU incrementally decompresses and uploads data to the graphics card (see Section 2.4).

2.2.1 Data Structures

To summarize, the main data structures are as follows. We predefine a small set of constant vertex and index buffers to encode the (x, y) geometry of the clipmap grids. And for each level $0 \ldots L-1$, we allocate an elevation map (a 1-channel floating-point 2D texture) and a normal map (a 4-channel 8-bit 2D texture). All these data structures reside in video memory.

2.2.2 Clipmap Size

Because the outer perimeter of each level must lie on the grid of the next-coarser level (as shown in Figure 2-4), the grid size n must be odd. Hardware may be optimized for texture sizes that are powers of 2, so we choose $n = 2^k-1$ (that is, 1 less than a power of 2) leaving 1 row and 1 column of the textures unused. Most of our examples use $n = 255$.

The choice of grid size $n = 2^k-1$ has the further advantage that the finer level is never exactly centered with respect to its parent next-coarser level. In other words, it is always offset by 1 grid unit either left or right, as well as either top or bottom (see Figure 2-4), depending on the position of the viewpoint. In fact, it is necessary to allow a finer level to shift while its next-coarser level stays fixed, and therefore the finer level must sometimes be off-center with respect to the next-coarser level. An alternative choice of grid size, such as $n = 2^k-3$, would provide the possibility for exact centering, but it would still require the handling of off-center cases and thus result in greater overall complexity.

NVIDIA Ex1023
Page Q-79

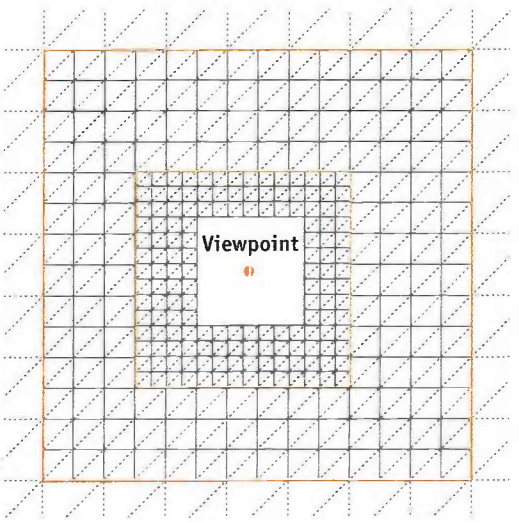


Figure 2-4. Two Successive Clipmap Rings
Note that vertices of the coarser level are coincident with those of the finer level at the interlevel boundary. This property is crucial for avoiding cracks in the rendered terrain.

2.3 Rendering

2.3.1 Active Levels

Although we have allocated L levels for the clipmap, we often render (and update) only a set of *active levels* $0 \ldots L'-1$, where the number of levels $L' \leq L$ is based on the height of the viewpoint over the underlying terrain. The motivation is that when the viewer is sufficiently high, the finest clipmap levels are unnecessarily dense and in fact result in aliasing artifacts. Specifically, we deactivate levels for which the grid extent is smaller than $2.5h$, where h is the viewer height above the terrain. Since all the terrain data resides in video memory, computing h involves reading back one point sample (immediately below the viewpoint) from the elevation map texture of the finest active level $(L'-1)$. This readback incurs a small cost, so we perform it only every few frames. Of course, we render level $L'-1$ using a complete square rather than a hollow ring.

The implementation described in Losasso and Hoppe 2004 allows "cropping" of a level if its data was not fully updated during viewer motion. To simplify our GPU implementation, we have decided to forgo this feature. We instead assume that a level is either fully updated or declared inactive.

NVIDIA Ex1023
Page Q-80

2.3.2 Vertex and Index Buffers

As explained earlier, we represent the grid (x, y) values as vertex data, while storing the z elevation values as a one-channel floating-point texture. A brute-force approach would be to define a single vertex buffer containing the (x, y) data for the entire ring within a level. To both reduce memory costs and enable view frustum culling, we instead break the ring into smaller 2D footprint pieces, as illustrated in Figure 2-5.

Most of the ring is created using 12 blocks (shown in gray) of size $m \times m$, where $m = (n + 1)/4$. Since the 2D grid is regular, the (x, y) geometries of all blocks within a level are identical up to a translation. Moreover, they are also identical across levels up to a uniform scale. Therefore, we can render the (x, y) geometries of all terrain blocks using a single read-only vertex buffer and index buffer, by letting the vertex shader scale and translate the block (x, y) geometry using a few parameters.

For our default clipmap size $n = 255$, this canonical block has 64×64 vertices. The index buffer encodes a set of indexed triangle strips whose lengths are chosen for optimal vertex caching. We use 16-bit indices for the index buffer, which allows a maximum block size of $m = 256$ and therefore a maximum clipmap size of $n = 1023$.

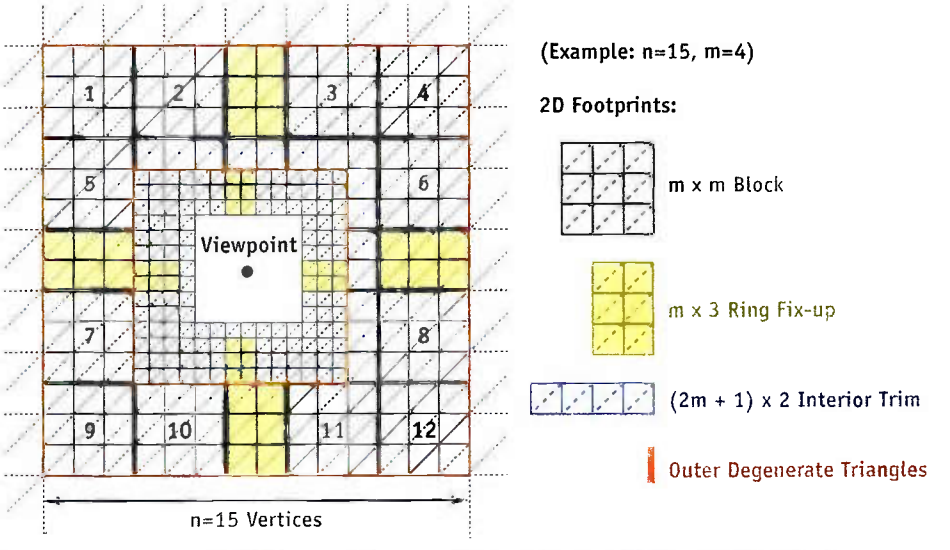


Figure 2-5. Partitioning a Clipmap Ring
Each clipmap ring is partitioned into a set of 2D footprints: 12 block instances, ring fix-up, and interior trim.

NVIDIA Ex1023
Page Q-81

However, the union of the 12 blocks does not completely cover the ring. We fill the small remaining gaps using a few additional 2D footprints, as explained next. Note that in practice, these additional regions have a very small area compared to the regular $m \times m$ blocks, as revealed in Figure 2-6. First, there is a gap of $(n - 1) - ((m - 1) \times 4) = 2$ quads at the middle of each ring side. We patch these gaps using four $m \times 3$ fix-up regions (shown in green in Figures 2-5 and 2-6). We encode these regions using one vertex and index buffer, and we reuse these buffers across all levels. Second, there is a gap of one quad on two sides of the interior ring perimeter, to accommodate the off-center finer level. This L-shaped strip (shown in blue) can lie at any of four possible locations (top-left, top-right, bottom-left, bottom-right), depending on the relative position of the fine level inside the coarse level. We define four vertex and one index buffer for this interior trim, and we reuse these across all levels.

Also, we render a string of degenerate triangles (shown in orange) on the outer perimeter. These zero-area triangles are necessary to avoid mesh T-junctions. Finally, for the finest level, we fill the ring interior with four additional blocks and one more L-shaped region.

Because the footprint (x, y) coordinates are local, these do not require 32-bit precision, so we pack them into a `D3DDECLTYPE_SHORT2`, requiring only 4 bytes per vertex. In the future, it may be possible to compute the (x, y) coordinates within the $m \times m$ block from the vertex index i itself as `(fmod(i, m), floor(i/m))`.

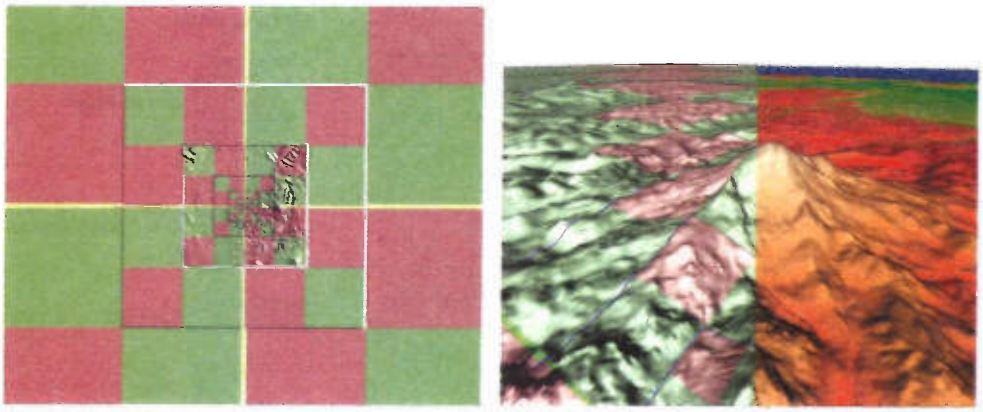


Figure 2-6. Visualizing the Nested Grid Structure

For practical clipmap sizes, most of the terrain is rendered using instances of the m×m block. Left: top view of terrain, showing each nested grid composed of 12 blocks. Right: forward view, showing nested grid structure and final shaded terrain.

NVIDIA Ex1023
Page Q-82

2.3.3 View Frustum Culling

View frustum culling is done at the block level on the CPU. Each block is extruded by $[z_{min}, z_{max}]$ and intersected with the view frustum in 3D. It is rendered only if this intersection is nonempty. Depending on the view direction, the rendering load is reduced by a factor of 2 to 3 for a 90-degree field of view, as shown in Figure 2-7.

2.3.4 `DrawPrimitive` Calls

For each level, we make up to 14 `DrawPrimitive` (DP) calls: 12 for the blocks, 1 for the interior trim, and 1 for the remaining triangles. With view frustum culling, on average only 4 of the 12 blocks are rendered each frame. The finest level requires 5 more calls to fill the interior hole. Thus, overall we make $6L + 5$ (71 for $L = 11$) DP calls per frame on average. This total number could be further reduced to $3L + 2$ (35) by instancing all blocks rendered in each level using a single DP call.

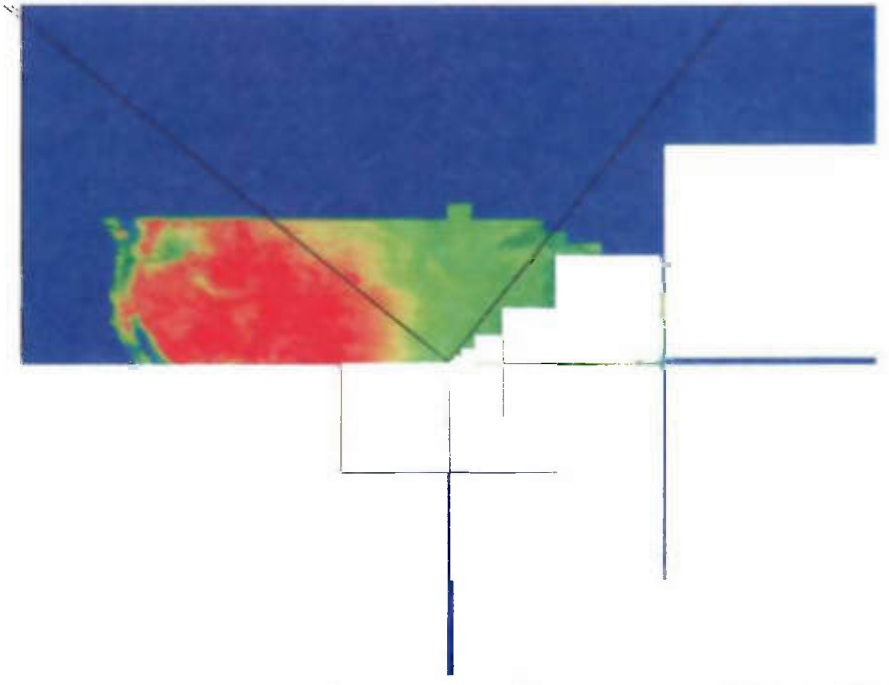


Figure 2-7. View Frustum Culling

NVIDIA Ex1023
Page Q-83

2.3.5 The Vertex Shader

The same vertex shader is applied to the rendering of all 2D footprints described previously. First, given the footprint (x, y) coordinates, the shader computes the world (x, y) coordinates using a simple scaling and translation. Next, it reads the z value from the elevation map stored in the vertex texture. No texture filtering is necessary because the vertices correspond one-to-one with the texture samples.

For smooth transitions, the vertex shader blends the geometry near the outer boundary of each level with that of the next-coarser level. It computes the blending parameter α based on the (x, y) location relative to the continuous viewer position (v_x, v_y). Specifically, $\alpha = \max(\alpha_x, \alpha_y)$, with

$$\alpha_x = \text{clamp}\left(\left(\left|x - v_x\right| - \left(\frac{n-1}{2} - w - 1\right)\right)\Big/ w,\ 0,\ 1\right),$$

and similarly for α_y.

Here, all coordinates are expressed in the $[0 \ldots n-1]$ range of the clipmap grid, and w is the transition width (chosen to be $n/10$). The desired property is that α evaluates to 0 except in the transition region, where it ramps up linearly to reach 1 at the outer perimeter. Figure 2-3a (on page 29) shows the evaluation of the parameter α within the transition regions in purple.

For geometric blending, we linearly interpolate between the current (fine-level) elevation z_f and the elevation z_c at the same (x, y) location in the next-coarser level:

$$z' = (1 - \alpha) z_f + \alpha z_c$$

In the general case, the sample location lies on an edge of the coarser grid, and z_c is obtained by averaging the two coarse samples on the edge endpoints. We could perform this computation at runtime, but it would require a total of three vertex-texture lookups (one for z_f and two for $z_c = (z_{c1} + z_{c2})/2$) and vertex-texture reads are currently expensive (Gerasimov et al. 2004).

Instead, we compute z_c as part of the clipmap update, and pack both z_f and z_c into the same 1-channel floating-point texture. We achieve this by storing z_f into the integer part of the float, and the difference $z_d = z_c - z_f$ (scaled) into the fractional part. The packing is done in the pixel shader of the upsampling stage (see Section 2.4.1).

NVIDIA Ex1023
Page Q-84

Listing 2-1 is the High-Level Shader Language (HLSL) vertex program for clipmap rendering.

Listing 2-1. Vertex Shader Code for Rendering a Clipmap Level

```
struct OUTPUT {
  vector pos   : POSITION;
  float2 uv    : TEXCOORD0; // coordinates for normal-map lookup
  float  z     : TEXCOORD1; // coordinates for elevation-map lookup
  float  alpha : TEXCOORD2; // transition blend on normal map
};

uniform float4 ScaleFactor, FineBlockOrig;
uniform float2 ViewerPos, AlphaOffset, OneOverWidth;
uniform float  ZScaleFactor, ZTexScaleFactor;
uniform matrix WorldViewProjMatrix;

// Vertex shader for rendering the geometry clipmap
OUTPUT RenderVS(float2 gridPos: TEXCOORD0)
{
  OUTPUT output;
  // convert from grid xy to world xy coordinates
  //  ScaleFactor.xy: grid spacing of current level
  //  ScaleFactor.zw: origin of current block within world
  float2 worldPos = gridPos * ScaleFactor.xy + ScaleFactor.zw;

  // compute coordinates for vertex texture
  //  FineBlockOrig.xy: 1/(w, h) of texture
  //  FineBlockOrig.zw: origin of block in texture
  float2 uv = gridPos * FineBlockOrig.xy + FineBlockOrig.zw;

  // sample the vertex texture
  float zf_zd = tex2Dlod(ElevationSampler, float4(uv, 0, 1));

  // unpack to obtain zf and zd = (zc - zf)
  //  zf is elevation value in current (fine) level
  //  zc is elevation value in coarser level
  float zf = floor(zf_zd);
  float zd = frac(zf_zd) * 512 - 256; // (zd = zc - zf)
```


NVIDIA Ex1023
Page Q-85

Listing 2-1 *(continued)*. Vertex Shader Code for Rendering a Clipmap Level

```
    // compute alpha (transition parameter) and blend elevation
    float2 alpha = clamp((abs(worldPos - ViewerPos) -
                          AlphaOffset) * OneOverWidth, 0, 1);
    alpha.x = max(alpha.x, alpha.y);
    float z = zf + alpha.x * zd;
    z = z * ZScaleFactor;

    output.pos = mul(float4(worldPos.x, worldPos.y, z, 1),
                     WorldViewProjMatrix);
    output.uv = uv;
    output.z = z * ZTexScaleFactor;
    output.alpha = alpha.x;
    return output;
}
```

2.3.6 The Pixel Shader

The pixel shader accesses the normal map and shades the surface. We let the normal map have twice the resolution of the geometry, because one normal per vertex is too blurry. The normal map is incrementally computed from the geometry whenever the clipmap is updated (see Section 2.4.3).

For smooth shading transitions, the shader looks up the normals for the current level and the next-coarser level and blends them using the α value computed in the vertex shader. Normally this would require two texture lookups. Instead, we pack the two normals as $(N_x, N_y, N_{cx}, N_{cy})$ in a single four-channel, 8-bit-per-channel texture, implicitly assuming $N_z = 1$ and $N_{cz} = 1$. We must therefore renormalize after unpacking.

Shading color is obtained using a lookup into a z-based 1D texture map.

Listing 2-2 shows the HLSL pixel program for clipmap rendering.

Listing 2-2. Pixel Shader Code for Rendering a Clipmap Level

```
// Parameters uv, alpha, and z are interpolated from vertex shader.
// Two texture samplers have min and mag filters set to linear:
//   NormalMapSampler:   2D texture containing normal map,
//   ZBasedColorSampler: 1D texture containing elevation-based color

uniform float3 LightDirection;
```


NVIDIA Ex1023
Page Q-86

Listing 2-2 *(continued)*. Pixel Shader Code for Rendering a Clipmap Level

```
// Pixel shader for rendering the geometry clipmap
float4 RenderPS(float2 uv    : TEXCOORD0,
                float  z     : TEXCOORD1,
                float  alpha : TEXCOORD2) : COLOR
{
  float4 normal_fc = tex2D(NormalMapSampler, uv);
  // normal_fc.xy contains normal at current (fine) level
  // normal_fc.zw contains normal at coarser level
  // blend normals using alpha computed in vertex shader
  float3 normal = float3((1 - alpha) * normal_fc.xy +
                         alpha * (normal_fc.zw), 1);

  // unpack coordinates from [0, 1] to [-1, +1] range, and renormalize
  normal = normalize(normal * 2 - 1);
  // compute simple diffuse lighting
  float s = clamp(dot(normal, LightDirection), 0, 1);
  // assign terrain color based on its elevation
  return s * tex1D(ZBasedColorSampler, z);
}
```

2.4 Update

As the viewer moves through the environment, each clipmap window translates within its pyramid level so as to remain centered about the viewer; thus, the window must be updated accordingly. Because viewer motion is coherent, generally only a small L-shaped region of the window needs to be incrementally processed each frame. Also, the relative motion of the viewer within the windows decreases exponentially at coarser levels, so coarse levels seldom require updating.

We update the active clipmap levels in coarse-to-fine order. Recall that each clipmap level stores two textures: a single-channel floating-point elevation map, and a four-channel, 8-bit-per-channel normal map. During the update step, we modify regions of these textures by rendering to them using a pixel shader.

To avoid having to translate the existing data, we perform all accesses toroidally as illustrated in Figure 2-8. That is, we use wraparound addressing to position the clipmap window within the texture image. Under this toroidal access, the L-shaped update region

NVIDIA Ex1023
Page Q-87



Figure 2-8. Processing a Toroidal Update
(a) Elevation map before update. (b) Viewer motion in green and update region in red. (c) Elevation map after update.

usually becomes a + shape, shown in red in Figure 2-8. We write to this region by rendering two quads. Note that three or four quads may be needed if the update region wraps across the texture boundary.

To facilitate terrain compression and synthesis, elevation data in the update region is first predicted by upsampling the coarser level, and a residual signal is then added to provide the detail. The next sections describe the three basic steps of the update process: upsampling the coarser-level elevation data, adding the residuals, and computing the normal map.

2.4.1 Upsampling

The finer-level geometry is predicted from the coarser one using an interpolatory subdivision scheme. We use the tensor-product version of the well-known four-point subdivision curve interpolant, which has mask weights $(-1/16, 9/16, 9/16, -1/16)$(Kobbelt 1996). This upsampling filter has the desirable property of being C^1 smooth.

Depending on the position of the sample in the grid (even-even, even-odd, odd-even, odd-odd), a different mask is used. For an even-even pixel, only 1 texture lookup is needed because the sample is simply interpolated; for an odd-odd pixel, 4×4 texture lookups are needed; the remaining two cases require 4 texture lookups. On the CPU, the mask to be used can be easily deduced using an "if" statement. However, branch

NVIDIA Ex1023
Page Q-88

statements are expensive in the pixel shader. Hence, we always do 16 texture lookups but apply a different mask based on a lookup in a 2×2 texture.

The HLSL code for the upsampling shader appears in Listing 2-3. Code for the missing functions `Upsample` and `ZCoarser` is included on the accompanying CD.

Listing 2-3. Pixel Shader Code for Creating/Updating a Clipmap Elevation Map

```
// residualSampler: 2D texture containing residuals, which can be
//  either decompressed data or synthesized noise
// p_uv: coordinates of the grid sample

uniform float2 Scale;

// Pixel shader for updating the elevation map
float4 UpsamplePS(float2 p_uv : TEXCOORD0) : COLOR
{
  float2 uv = floor(p_uv);

  // the Upsample() function samples the coarser elevation map
  //  using a linear interpolatory filter with 4x4 taps
  // (depending on the even/odd configuration of location uv,
  //  it applies 1 of 4 possible masks)
  float z_predicted = Upsample(uv);  // details omitted here

  // add the residual to get the actual elevation
  float residual = tex2D(residualSampler, p_uv * Scale);
  float zf = z_predicted + residual;

  // zf should always be an integer, since it gets packed
  //  into the integer component of the floating-point texture
  zf = floor(zf);

  // compute zc by linearly interpolating the vertices of the
  //  coarse-grid edge on which the sample p_uv lies
  float zc = ZCoarser(uv);   // details omitted here
  float zd = zc - zf;
  // pack the signed difference zd into the fractional component
  float zf_zd = zf + (zd + 256)/512;

  return float4(zf_zd, 0, 0, 0);
}
```


NVIDIA Ex1023
Page Q-89

2.4.2 Residuals

The residuals added to the upsampled coarser data can come from either decompression or synthesis.

In the compressed representation, the residual data is encoded using image compression. We use a particular lossy image coder called PTC because it supports efficient region-of-interest decoding (Malvar 2000). The CPU performs the decompression and stores the result into a residual texture image.

Alternatively, we synthesize fractal detail by letting the residuals be uncorrelated Gaussian noise (Fournier et al. 1982). This synthesis is made extremely fast on the GPU by reading a small precomputed 2D texture containing Gaussian noise. We enable texture wrapping to extend the Gaussian texture infinitely, and apply a small magnification to the texture coordinates to break the regular periodicity. The superposition of noise across the many resolution levels gives rise to a terrain without any visible repetitive patterns, as Figure 2-9 shows.

2.4.3 Normal Map

The shader that updates the normal map for a level takes as input the elevation map at the same level. It computes the current normal as the cross product of two grid-aligned tangent vectors. Additionally, it does a texture lookup to gather the normal from the coarser level. Finally, it packs $(N_x, N_y, N_{cx}, N_{cy})$ into the four-channel texture. The code is included in the accompanying CD.

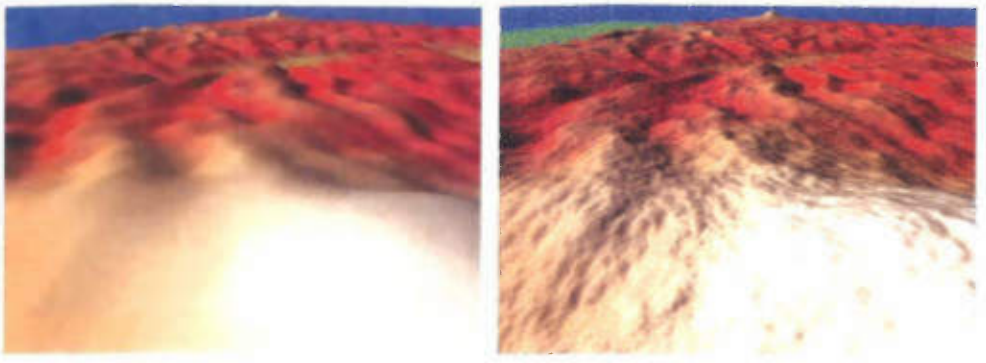


Figure 2-9. On-the-Fly Terrain Synthesis
(a) Coarse geometry plus zero residuals. (b) Coarse geometry plus synthesized noise.

NVIDIA Ex1023
Page Q-90

2.5 Results and Discussion

Our main data set is a 216,000×93,600 height map of the conterminous United States at 30-meter spacing and 1.0-meter vertical resolution. It compresses by a factor of more than 100 and therefore fits in only 355 MB of system memory. We render this terrain into a 1024×768 window on a PC with a 2.4 GHz Pentium 4, 1 GB system memory, and an NVIDIA GeForce 6800 GT.

Rendering rate: For $L = 11$ levels of size $n = 255$, we obtain 130 frames/sec with view frustum culling, at a rendering rate of 60 million triangles/sec. The use of a vertex-texture lookup is a bottleneck, since removing the lookup increases rendering rate to 185 frames/sec. (For $n = 127$, the rate with vertex textures is 298 frames/sec.)

Update rate: Table 2-2 shows the processing times for the various steps of the update process, for the extreme case of updating a whole level at once. Decompression on the CPU is clearly the bottleneck of the update process.

Final frame rate: For decompressed terrains, the system maintains a rate of around 87 frames/sec during viewer motion, and for synthesized terrains, the frame rate is about 120 frames/sec.

Table 2-2. Update Times for Processing an Entire 255×255 Level
These are worst-case numbers because, in general, during smooth viewer motions, only small fractions of a full level need be updated per frame.

	Previous Implementation*	GPU-Based Implementation
Upsampling	3 ms	1.0 ms
Decompression	8 ms	8 ms**
Synthesis	3 ms	~0 ms
Normal-Map Computation	11 ms	0.6 ms

* Losasso and Hoppe 2004.
**Still on the CPU.

2.6 Summary and Improvements

We have presented a GPU implementation of the geometry clipmap framework. The representation of geometry using regular grids allows nearly all computation to proceed on the graphics card, thereby offloading work from the CPU. The system supports interactive flight over a 20-billion-sample grid of the U.S. data set stored in just 355 MB of memory at around 90 frames/sec.

NVIDIA Ex1023
Page Q-91

2.6.1 Vertex Textures

Geometry clipmaps use vertex textures in a very special, limited way: the texels are accessed essentially in raster-scan order, and in one-to-one correspondence with vertices. Thus, we can hope for future mechanisms that would allow increased rendering efficiency.

2.6.2 Eliminating Normal Maps

It should be possible to directly compute normals in the pixel shader by accessing four filtered samples of the elevation map. At present, vertex texture lookups require that the elevation maps be stored in 32-bit floating-point images, which do not support efficient bilinear filtering.

2.6.3 Memory-Free Terrain Synthesis

Fractal terrain synthesis is so fast on the GPU that we can consider regenerating the clipmap levels during every frame, thereby saving video memory. We allocate two textures, T_1 and T_2, and toggle between them as follows. Let T_1 initially contain the coarse geometry used to seed the synthesis process. First, we use T_1 as a source texture in the update pixel shader to upsample and synthesize the next-finer level into the destination texture T_2. Second, we use T_1 as a vertex texture to render its clipmap ring. Then, we swap the roles of the two textures and iterate the process until all levels are synthesized and rendered. Initial experiments are extremely promising, with frame rates of about 59 frames/sec using $L = 9$ levels.

2.7 References

Fournier, Alain, Don Fussell, and Loren Carpenter. 1982. "Computer Rendering of Stochastic Models." *Communications of the ACM* 25(6), June 1982, pp. 371–384.

Gerasimov, Philip, Randima Fernando, and Simon Green. 2004. "Shader Model 3.0: Using Vertex Textures." NVIDIA white paper DA-01373-001_v00, June 2004. Available online at **http://developer.nvidia.com/object/using_vertex_textures.html**.

Kobbelt, Leif. 1996. "Interpolatory Subdivision on Open Quadrilateral Nets with Arbitrary Topology." In *Eurographics 1996*, pp. 409–420.

NVIDIA Ex1023
Page Q-92

Losasso, Frank, and Hugues Hoppe. 2004. "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)* 23(3), pp. 769–776.

Malvar, Henrique. 2000. "Fast Progressive Image Coding without Wavelets." In *Data Compression Conference (DCC '00)*, pp. 243–252.

Tanner, Christopher, Christopher Migdal, and Michael Jones. 1998. "The Clipmap: A Virtual Mipmap." In *Computer Graphics (Proceedings of SIGGRAPH 98)*, pp. 151–158.

NVIDIA Ex1023
Page Q-93



Chapter 3

Inside Geometry Instancing

Francesco Carucci
Lionhead Studios

A great way to enrich the user experience in an interactive application is to present a credible world, full of small, interesting features and objects. From countless blades of grass, to trees, to generic clutter: it all improves the final perception and helps maintain the user's "suspension of disbelief." If the user can believe the world he is immersed in, he is more likely to be emotionally connected to that world—the Holy Grail of game development.

From a rendering point of view, achieving this richness translates into rendering many small objects, often each one similar to the next, with only small differences such as color, position, and orientation. For example, every tree in a big forest might be geometrically very similar, but all may be different in terms of their color or height. The user perceives a forest made of many unique trees and believes the world to be credible, enriching his or her game experience.

However, rendering a large number of small objects, each made from a few polygons, imposes a big strain on today's GPUs and rendering libraries. Graphics APIs such as Direct3D and OpenGL are not designed to efficiently render a small number of polygons thousands of times per frame (Wloka 2003).

This chapter addresses the problem of rendering many unique instances of the same geometry in Direct3D. Figure 3-1 shows an example from *Black & White 2*, a game in development at Lionhead Studios.

NVIDIA Ex1023
Page Q-94



Figure 3-1. Geometry Instancing in *Black & White 2*

3.1 Why Geometry Instancing?

Submitting triangles to the GPU for rendering in Direct3D is a relatively slow operation. Wloka 2003 shows that a 1 GHz CPU can render only around 10,000 to 40,000 batches per second in Direct3D. On a more modern CPU, we can expect this number to be between 30,000 and 120,000 batches per second (around 1,000 to 4,000 batches per frame at 30 frames/sec). That's not much! It means that if we want to render a forest of trees, and we submit one tree per batch, we cannot render more than 4,000 trees, regardless of how many polygons are in a tree—with no CPU time left for the rest of the game. This situation is clearly not ideal. We would like to be able to minimize the number of state and texture changes in our application and render the same triangles multiple times within the same batch in a single call to Direct3D. Thus, we would minimize CPU time spent in submitting batches and free up that time for other systems, such as physics, artificial intelligence, and game logic.

NVIDIA Ex1023
Page Q-95

3.2 Definitions

We first define concepts behind geometry instancing, to make clear the kinds of objects that are part of the problem.

3.2.1 Geometry Packet

A geometry packet is a description of a packet of geometry to be instanced, a collection of vertices and indices. A geometry packet can be described in terms of vertices—with position, texture coordinates, normal, possibly tangent space and bones information for skinning, and per-vertex colors—and in terms of indices into the vertex stream. This kind of description directly maps to the most efficient way to submit geometry to the GPU.

A geometry packet is an abstract description of a piece of geometry, where the geometric entities are expressed in model space without any explicit reference to the context in which they will be rendered.

Here's a possible description in code of a geometry packet; it includes both the geometric information for the object as well as its bounding sphere:

```
struct GeometryPacket
{
  Primitive mPrimType
  void* mVertices;
  unsigned int mVertexStride;

  unsigned short* mIndices;

  unsigned int mVertexCount;
  unsigned int mIndexCount;

  D3DXVECTOR3 mSphereCentre;
  float mSphereRadius;
};
```

3.2.2 Instance Attributes

Typical attributes per instance are the model-to-world transformation matrix, the instance color, and an animation player providing the bones used to skin the geometry packet.

NVIDIA Ex1023
Page Q-96

```
struct InstanceAttributes
{
  D3DXMATRIX mModelMatrix;
  D3DCOLOR mInstanceColor;
  AnimationPlayer* mAnimationPlayer;
  unsigned int mLOD;
};
```

3.2.3 Geometry Instance

A geometry instance is a geometry packet with the attributes specific to the instance. It directly connects a geometry packet and the instance attributes packet with which it's going to be rendered, giving an almost complete description of the entity ready to be submitted to the GPU.

Here's how a geometry instance structure might look in code:

```
struct GeometryInstance
{
  GeometryPacket* mGeometryPacket;
  InstanceAttributes mInstanceAttributes;
};
```

3.2.4 Render and Texture Context

The render context is the current state of the GPU in terms of render states (such as alpha blending and testing states, active render target, and more). The texture context tracks the currently active textures. Render context and texture context are usually modeled with classes, as shown in Listing 3-1.

3.2.5 Geometry Batch

A batch is a collection of geometry instances and the render states and texture context to be used to render the collection. It always directly maps to a single `DrawIndexed-Primitive()` call, to simplify the design of the class.

Listing 3-2 is an abstract interface for a geometry batch class.

NVIDIA Ex1023
Page Q-97

Listing 3-1. Render and Texture Context Classes

```
class RenderContext
{
public:

  // Begin the render context and make its render state active
  //
  void Begin(void);

  // End the render context and restore previous render state,
  // if necessary
  //
  void End(void);

private:

  // Any description of the current render state and pixel
  // and vertex shaders.
  // D3DX Effect framework is particularly useful.
  //
  ID3DXEffect* mEffect;

  // Application-specific render states
  // . . .
};

class TextureContext
{
public:

  // Set current textures to the appropriate texture stages
  //
  void Apply(void) const;

private:
  Texture mDiffuseMap;
  Texture mLightMap;

  // . . .
};
```


NVIDIA Ex1023
Page Q-98

Listing 3-2. GeometryBatch Class

```
class GeometryBatch
{
public:

  // Remove all instances from the geometry batch
  //
  virtual void ClearInstances(void);

  // Add an instance to the collection and return its ID.
  // Return -1 if it can't accept more instances.
  //
  virtual int AddInstance(GeometryInstance* instance);

  // Commit all instances, to be called once before the
  // render loop begins and after every change to the
  // instances collection
  //
  virtual unsigned int Commit(void) = 0;

  // Update the geometry batch, eventually prepare GPU-specific
  // data ready to be submitted to the driver, fill vertex and
  // index buffers as necessary, to be called once per frame
  //
  virtual void Update(void) = 0;

  // Submit the batch to the driver, typically implemented
  // with a call to DrawIndexedPrimitive
  //
  virtual void Render(void) const = 0;

private:
  GeometryInstancesCollection  mInstances;

  // . . .
};
```


NVIDIA Ex1023
Page Q-99

3.3 Implementation

The engine's renderer can see geometry instancing only in terms of the abstract interface of a GeometryBatch, which hides the specifics of the instancing implementation and provides services for managing instances, updating, and rendering a batch. This way, the engine can concentrate on sorting batches to minimize render state and texture changes, while the GeometryBatch takes care of the actual implementation and communication protocol with Direct3D.

Listing 3-3 shows a simple implementation of a render loop in pseudocode, sorting GeometryBatches first by render context and then by texture context, thus minimizing the number of render-state changes.

Listing 3-3. Render Loop in Pseudocode

```
// Update phase
Foreach GeometryBatch in ActiveBatchesList
  GeometryBatch.Update();

// Render phase
Foreach RenderContext
Begin
  RenderContext.BeginRendering();
  RenderContext.CommitStates();

  Foreach TextureContext
  Begin
    TextureContext.Apply();
    Foreach GeometryBatch in the texture context
      GeometryBatch.Render();
  End
End
```

The update and render phases can be kept separate in case we want to update all batches once and render several times: this idea is extremely useful when implementing shadow mapping or water reflection and refraction, for example.

In this chapter, we discuss four implementations of GeometryBatch and analyze their performance characteristics, with a particular interest in the memory footprint and flexibility of each technique.

NVIDIA Ex1023
Page Q-100

Here's a brief summary:

- **Static batching.** The fastest way to instance geometry. Each instance is transformed once to world space, its attributes are applied, and then it's sent already transformed to the GPU with every frame. Although simple, static batching is the least flexible technique.
- **Dynamic batching.** The slowest way to instance geometry. Each instance is streamed to GPU memory every frame, already transformed and with attributes applied. Dynamic batching seamlessly supports skinning and provides the most flexible implementation.
- **Vertex constants instancing.** A hybrid implementation in which multiple copies of the geometry for each instance are copied once into GPU memory. Instance attributes are then set every frame through vertex constants, and a vertex shader completes geometry instancing.
- **Batching with Geometry Instancing API.** Using the Geometry Instancing API provided by DirectX 9 and fully supported in hardware by GeForce 6 Series GPUs, this implementation offers a flexible and fast solution to geometry instancing. Unlike all other methods, this does not require geometry packet replication in the Direct3D vertex stream.

3.3.1 Static Batching

In static batching, we want to transform all instances once and copy them to a static vertex buffer. The most noticeable advantages of static batching are its rendering efficiency and its compatibility with any GPU on the market, regardless of age.

To implement static batching, we first create a vertex buffer object (with its associated index buffer) to fill with the transformed geometry. This object must be big enough to accept the maximum number of instances we decide to handle. Because we fill the buffer once and never touch it again, we can create it with the `D3USAGE_WRITEONLY` flag, giving a useful hint to the driver to put the buffer in the fastest memory available:

```
HRESULT res;

res = lpDevice->CreateVertexBuffer(
  MAX_STATIC_BUFFER_SIZE,
  D3DUSAGE_WRITEONLY,
  0,
  D3DPOOL_MANAGED,
  &mStaticVertexStream,
  0);

ENGINE_ASSERT(SUCCEEDED(res));
```


NVIDIA Ex1023
Page Q-101

Whether to choose `D3DPOOL_MANAGED` or `D3DPOOL_DEFAULT` to create the buffer depends on the particular application and especially on the memory management strategy of the engine.

The next step is to implement the `Commit()` method to actually fill the vertex and index buffers with the transformed geometry that needs to be rendered.

Here is the `Commit()` method in pseudocode:

```
Foreach GeometryInstance in Instances
Begin
  Transform geometry in mGeometryPacket to world space
    with instance mModelMatrix
  Apply other instance attributes (like instance color)
  Copy transformed geometry to the Vertex Buffer
  Copy indices (with the right offset) to the Index Buffer
  Advance current pointer to the Vertex Buffer
  Advance current pointer to the Index Buffer
End
```

From now on, it's just a matter of submitting the instances we have already prepared with a simple call to `DrawIndexedPrimitive()`. The actual implementation of the `Update()` and `Render()` methods is trivial.

Static batching is the fastest way to render many instances, and it supports instances of different geometry packets inside the same geometry batch, but it has a few serious limitations:

- **Large memory footprint.** Depending on the size of each geometry packet and the number of instances being rendered, the memory footprint might become too big; for large worlds, it's easy to spend the video memory budget reserved for geometry. Falling back to AGP memory is possible, but that makes this technique slower and thus less appealing.
- **No support for different levels of detail.** Because all instances are copied once into the vertex buffer at commit time, choosing a level of detail valid for all situations is not easy and leads to suboptimal usage of the polygon budget. A semi-static solution is possible, in which all the levels of detail of a particular instance are instanced in the vertex buffer, while indices are streamed every frame, picking up the right level of detail for each instance. But this makes the implementation clumsier, defeating the purpose of this solution: simplicity and efficiency.

NVIDIA Ex1023
Page Q-102

- **No support for skinning.**
- **No direct support for moving instances.** Implementing moving instances requires vertex shader logic with dynamic branching in order to be efficient. This can be prohibitive on older graphics cards. The end result would essentially be vertex constants instancing.

These limitations are addressed with the next technique, trading off rendering speed for additional flexibility.

3.3.2 Dynamic Batching

Dynamic batching overcomes the limitations of static batching at the cost of reducing rendering efficiency. A major advantage of dynamic batching is, like static batching, that it can be implemented on any GPU without support for advanced programmable pipelines.

The first step is to create a vertex buffer (and its associated index buffer) as dynamic by using the `D3DUSAGE_DYNAMIC` and `D3DPOOL_DEFAULT` flags. These flags ensure that the buffer is placed in the best possible memory location, given that we intend to dynamically update its contents.

```
HRESULT res;

res = lpDevice->CreateVertexBuffer(
  MAX_DYNAMIC_BUFFER_SIZE,
  D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY,
  0,
  D3DPOOL_DEFAULT,
  &mDynamicVertexStream,
  0);
```

Choosing the right value for `MAX_DYNAMIC_BUFFER_SIZE` is vital in this implementation. Two choices dictating the actual value are possible:

- Choosing a value large enough to contain geometry for all instances that might be needed in a frame
- Choosing a value large enough to contain geometry for a certain number of instances

The first approach ensures a proper separation between updating and rendering a batch. Updating a batch means streaming all instances into the dynamic buffer; rendering just submits the geometry with a call to `DrawIndexedPrimitive()`. But this

NVIDIA Ex1023
Page Q-103

approach typically requires a large amount of graphics memory (that is, either video or AGP memory) and is less reliable in the worst case if we can't ensure that the buffer is always big enough during the life of the application.

The second solution requires interleaving between streaming geometry and rendering it: when the dynamic buffer is full, its geometry is submitted for rendering and the content of the buffer is discarded, ready for more instances to be streamed. For optimal performance, it is important to use the proper protocol, that is, locking the dynamic buffer with `D3DLOCK_DISCARD` at the beginning of a new batch of instances and with `D3DLOCK_WRITEONLY` for each new instance to be streamed. The drawback of this solution is that it requires locking the buffer and streaming geometry again each time the batch needs to be rendered, for example, when implementing shadow mapping.

The choice depends on the particular application and its requirements. In this chapter, we choose the first solution, for its simplicity and clarity, but we add a slight complication: dynamic batching naturally supports skinning, and we address this in our implementation.

The `Update()` method is very similar to the `Commit()` method we have already described in Section 3.3.1, but it is executed every frame. Here it is in pseudocode:

```
Foreach GeometryInstance in Instances
Begin
  Transform geometry in mGeometryPacket to world space with
    instance mModelMatrix
  If instance needs skinning, request a set of bones from
    mAnimationPlayer and skin geometry
  Apply other instance attributes (like instance color)
  Copy transformed geometry to the Vertex Buffer
  Copy indices (with the right offset) to the Index Buffer
  Advance current pointer to the Vertex Buffer
  Advance current pointer to the Index Buffer
End
```

In this case, the `Render()` method is a trivial call to `DrawIndexedPrimitive()`.

3.3.3 Vertex Constants Instancing

In vertex constants instancing, we take advantage of vertex constants to store instancing attributes. Vertex constants batching is very fast in terms of rendering performance, and it supports instances moving from frame to frame, but it pays in terms of flexibility.

NVIDIA Ex1023
Page Q-104

The main limitations are these:

- Limited number of instances per batch, depending on the number of constants available; usually no more than 50 to 100 instances can be batched in a single call. However, this can be enough to significantly reduce the CPU load from draw calls.
- No skinning; vertex constants are already used to store instance attributes.
- Need graphics hardware support for vertex shaders.

The first step is to prepare a static vertex buffer (with its associated index buffer) to store multiple copies of the same geometry packets, stored in model space, one for each instance in the batch.

The vertex buffer layout is illustrated in Figure 3-2.

The source vertex format must be updated to add an integer index to each vertex, which is constant per instance, indicating which instance the geometry packet belongs

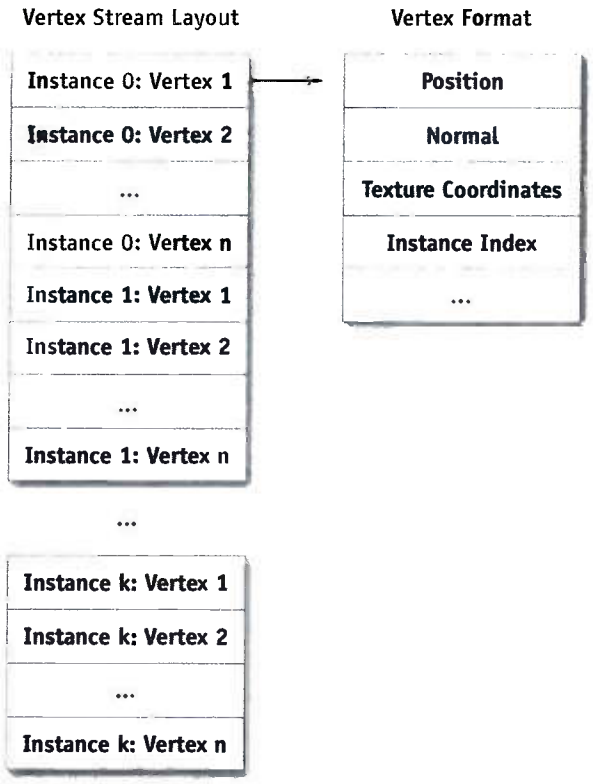


Figure 3-2. Vertex Buffer Layout

NVIDIA Ex1023
Page Q-105

to. This is similar to palette skinning, in which each vertex contains an index to the bone (or multiple bones) affecting it.

The updated vertex format is this:

```
struct InstanceVertex
{
  D3DXVECTOR3  mPosition;

  // Other vertex properties, such as normal and texture
  // coordinates

  WORD  mInstanceIndex[4]; // Direct3D requires SHORT4

};
```

The `Commit()` method prepares the vertex buffer with the proper layout after all instances have been added to the geometry batch.

The next step is to upload attributes for each instance being rendered. Suppose we have only the model matrix, describing instance position and orientation, and the instance color.

When using a DirectX 9-class GPU, we can use a maximum of 256 vertex constants; we can reserve 200 for instancing attributes. In our example, each instance requires 4 constants for the model matrix and 1 constant for instance color, for a total of 5 constants per instance, leading to a maximum of 40 instances per batch.

Listing 3-4 shows the `Update()` method. The actual instancing is done in the vertex shader, shown in Listing 3-5.

Listing 3-4. The `Update()` Method

```
D3DXVECTOR4 instancesData[MAX_NUMBER_OF_CONSTANTS];

unsigned int count = 0;

for (unsigned int i = 0; i < GetInstancesCount(); ++i)
{
  // write model matrix
  instancesData[count++] = *(D3DXVECTOR4*)
      &mInstances[i].mModelMatrix.m11;
  instancesData[count++] = *(D3DXVECTOR4*)
      &mInstances[i].mModelMatrix.m21;
```


NVIDIA Ex1023
Page Q-106

Listing 3-4 *(continued)*. The `Update()` Method

```
    instancesData[count++] = *(D3DXVECTOR4*)
        &mInstances[i].mModelMatrix.m31;
    instancesData[count++] = *(D3DXVECTOR4*)
        &mInstances[i].mModelMatrix.m41;

    // write instance color
    instanceData[count++] = ConvertColorToVec4(mInstances[i].mColor));
}

lpDevice->SetVertexConstants(
    INSTANCES_DATA_FIRST_CONSTANT,
    instancesData,
    count);
```

Listing 3-5. Vertex Shader for Vertex Constants Instancing

```
// vertex input declaration
struct vsInput
{
    float4 position : POSITION;
    float3 normal   : NORMAL;

    // other vertex data
    int4 instance_index : BLENDINDICES;
};

vsOutput VertexConstantsInstancingVS(
    in vsInput input)
{
    // get the instance index; the index is premultiplied
    // by 5 to take account of the number of constants
    // used by each instance
    int instanceIndex = ((int[4]) (input.instance_index))[0];

    // access each row of the instance model matrix
    float4 m0 = InstanceData[instanceIndex + 0];
    float4 m1 = InstanceData[instanceIndex + 1];
    float4 m2 = InstanceData[instanceIndex + 2];
    float4 m3 = InstanceData[instanceIndex + 3];

    // construct the model matrix
    float4x4 modelMatrix = { m0, m1, m2, m3 };
```


NVIDIA Ex1023
Page Q-107

Listing 3-5 *(continued)*. Vertex Shader for Vertex Constants Instancing

```
    // get the instance color
    float4 instanceColor = InstanceData[instanceIndex + 4];

    // transform input position and normal to world space with
    // the instance model matrix
    float4 worldPosition = mul(input.position, modelMatrix);
    float3 worldNormal = mul(input.normal, modelMatrix);

    // output position, normal, and color
    output.position = mul(worldPosition, ViewProjectionMatrix);
    output.normal = mul(worldNormal, ViewProjectionMatrix);
    output.color = instanceColor;

    // output other vertex data
}
```

The `Render()` sets the view projection matrix and submits all instances with a `DrawIndexedPrimitive()` call.

A possible optimization in production code is to store the rotational part of the model matrix as a quaternion, saving two vertex constants and increasing the maximum number of instances to around 70. A uniform scaling value can be stored in the w component of the translation vector. The model matrix is then reconstructed in the vertex shader, increasing its complexity and execution time.

3.3.4 Batching with the Geometry Instancing API

The last technique we present in this chapter is batching with the Geometry Instancing API exposed by DirectX 9 and fully implemented in hardware on GeForce 6 Series GPUs. As more hardware starts supporting the Geometry Instancing API, this technique will become more interesting, because it elegantly solves limitations imposed by vertex constants instancing, has a very limited memory footprint, and requires little intervention by the CPU. The only drawback of this technique is that it can only handle instances of the same geometry packet.

DirectX 9 provides the following call to access the Geometry Instancing API:

```
HRESULT SetStreamSourceFreq(
  UINT StreamNumber,
  UINT FrequencyParameter);
```


NVIDIA Ex1023
Page Q-108

StreamNumber is the index of the source stream and *FrequencyParameter* indicates the number of instances each vertex contains.

We first create two vertex buffers: a static one to store geometry for the single geometry packet we want to instance multiple times, and a dynamic one to store instance data. The two vertex streams are shown in Figure 3-3.

The `Commit()` method ensures that all instances are using only one geometry packet and copies its geometry to the static buffer.

The `Update()` method simply copies all instance attributes into the second stream. Even though this method looks very similar to the `Update()` method of a dynamic batch, CPU intervention and graphics bus (AGP or PCI Express) bandwidth are minimal. Moreover, we can choose to allocate a vertex buffer big enough to accept attributes for all instances without worrying about graphics memory consumption, because each instance attribute uses only a fraction of the memory consumed by a whole geometry packet.

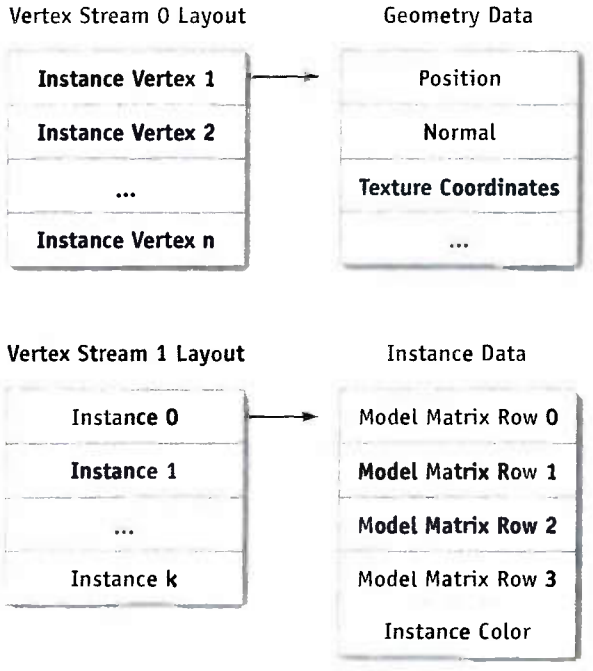


Figure 3-3. Vertex Streams for Instancing with the Geometry Instancing API

NVIDIA Ex1023
Page Q-109

The `Render()` method sets up two streams with the correct stream frequency and issues a `DrawIndexedPrimitive()` call to render all instances in one batch, as shown in Listing 3-6.

The GPU takes care of virtually duplicating vertices of the first stream and packing them together with the second stream. The vertex shader sees as input a vertex with its position in model space and an additional instance attribute giving the model matrix to transform it to world space.

Listing 3-6. Setting Up Two Streams for Batching with the Geometry Instancing API

```
unsigned int instancesCount = GetInstancesCount();

// set up stream source frequency for the first stream
// to render instancesCount instances
// D3DSTREAMSOURCE_INDEXEDDATA tells Direct3D we'll use
// indexed geometry for instancing
lpDevice->SetStreamSourceFreq(
  0,
  D3DSTREAMSOURCE_INDEXEDDATA | instancesCount);

// set up first stream source with the vertex buffer
// containing geometry for the geometry packet
lpDevice->SetStreamSource(
  0,
  mGeometryInstancingVB[0],
  0,
  mGeometryPacketDecl);

// set up stream source frequency for the second stream;
// each set of instance attributes describes one instance
// to be rendered
lpDevice->SetStreamSourceFreq(
  1,
  D3DSTREAMSOURCE_INSTANCEDATA | 1);

// set up second stream source with the vertex buffer
// containing all instances' attributes
pd3dDevice->SetStreamSource(
  1,
  mGeometryInstancingVB[0],
  0,
  mInstancesDataVertexDecl);
```


NVIDIA Ex1023
Page Q-110

The vertex shader code appears in Listing 3-7.

Listing 3-7. Vertex Shader for Batching with the Geometry Instancing API

```
// vertex input declaration
struct vsInput
{
  // stream 0
  float4 position : POSITION;
  float3 normal   : NORMAL;

  // stream 1
  float4 model_matrix0 : TEXCOORD0;
  float4 model_matrix1 : TEXCOORD1;
  float4 model_matrix2 : TEXCOORD2;
  float4 model_matrix3 : TEXCOORD3;

  float4 instance_color : D3DCOLOR;
};

vsOutput GeometryInstancingVS(
  in vsInput input)
{
  // construct the model matrix
  float4x4 modelMatrix =
  {
    input.model_matrix0,
    input.model_matrix1,
    input.model_matrix2,
    input.model_matrix3
  };

  // transform input position and normal to world space
  // with the instance model matrix
  float4 worldPosition = mul(input.position, modelMatrix);
  float3 worldNormal = mul(input.normal, modelMatrix);

  // output position, normal, and color
  output.position = mul(worldPosition, ViewProjectionMatrix);
  output.normal = mul(worldNormal, ViewProjectionMatrix);
  output.color = input.instance_color;

  // output other vertex data
}
```


NVIDIA Ex1023
Page Q-111

With minimal CPU overhead and memory footprint, this technique can efficiently render many copies of the same geometry and is therefore the ideal solution in many scenarios. The only drawback is that it requires special hardware capabilities and doesn't easily support skinning.

To implement skinning, it would be possible to store all bones for all instances into a texture and fetch the right bones of the right instance with the texture fetch available in Shader Model 3.0 (which is required by the Geometry Instancing API). If this solution seems attractive, the performance impact of using texture fetches with this kind of access pattern is uncertain and should be tested.

3.4 Conclusion

This chapter defined the concepts behind geometry instancing and described four different techniques to achieve the goal of efficiently rendering the same geometry multiple times. Each technique has pros and cons, and no single technique clearly represents the ideal choice for every scenario. The best choice depends mainly on the application and on the type of objects being rendered.

Some scenarios and proposed solutions:

- An indoor scene with many static instances of the same geometry, which rarely or never move (such as walls or furniture). This is an ideal case for static batching.
- An outdoor scene with many instances of animated objects, as in a strategy game with big battles involving hundreds of soldiers. In this scenario, dynamic batching is probably the best solution.
- An outdoor scene with lots of vegetation and trees, many of which tend to modify their attributes (for example, trees and grass swaying in the wind), and lots of particle systems. This is probably the best scenario for the Geometry Instancing API.

Often the same application needs to rely on two or more techniques. Figures 3-4 and 3-5 show two examples. In such cases, hiding the implementation behind an abstract geometry batch concept makes the engine more modular and easier to maintain. As a result, the costs of implementing all the geometry-instancing techniques are greatly reduced for the entire application.

NVIDIA Ex1023
Page Q-112



Figure 3-4. Combining Static Batching and Geometry Instancing in a Real Scene
Buildings in this screenshot are rendered with static batching, while trees use the Geometry Instancing API.



Figure 3-5. Combining GPU and CPU Animation in a Real Scene
Close soldiers are animated on the GPU, while distant soldiers are batched in few dynamic batches and animated on the CPU.

NVIDIA Ex1023
Page Q-113

3.5 References

Cebenoyan, Cem. 2004. "Graphics Pipeline Performance." In *GPU Gems*, edited by Randima Fernando, pp. 473–486. Addison-Wesley. Available online at **http://developer.nvidia.com/object/GPU_Gems_Samples.html**

O'Rorke, John. 2004. "Managing Visibility for Per-Pixel Lighting." In *GPU Gems*, edited by Randima Fernando, pp. 245–257. Addison-Wesley.

Wloka, Matthias. 2003. "Batch, Batch, Batch: What Does It Really Mean?" Presentation at Game Developers Conference 2003. **http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf**

NVIDIA Ex1023
Page Q-114



Chapter 4

Segment Buffering

Jon Olick
2015

As graphical requirements evolve to include more polygons, more clutter, and better lighting to create more realistic-looking scenes, we are forced to produce more art to meet those requirements. One technique that helps to conserve development time is an instancing approach. For example, a single chair is modeled and placed all around the game world (see Figure 4-1). Instancing helps immensely to shrink development time. It is, however, all too common for modern games to become limited by the number of draw calls (Wloka 2003). If each instance used in a scene leads to one more batch, it may not be possible to realize the benefits of instancing: batches become the bottleneck. Segment buffering is a technique that collects multiple instances that are close to each other in the scene and merges them into "über-instances," thus reducing the number of batches and providing a simple and elegant solution to the batch bottleneck.

4.1 The Problem Space

Figure 4-1 shows a scene containing many static objects of the same material. Most GPUs cannot efficiently render scenes like this for more than a thousand instances; each instance potentially requires a render-state change, such as transform changes, light map texture changes, or vertex stream changes. These render-state changes cause driver and thus CPU overhead, ultimately resulting in poor performance. Although we

NVIDIA Ex1023
Page Q-115



Figure 4-1. A Scene with Many Instances of the Same Object
From Men of Valor: Vietnam

could work around this problem on the content side by merging multiple models to create new models to instance around the world (thereby reducing the total number of instances), such an approach would require making a fair amount of custom art and could cause the maps to look repetitious.

4.2 The Solution

Segment buffering automates the merging of similar instances while maintaining most of the benefits of rendering separate instances. The primary benefits of segment buffering are thus a nonrepetitious look and the ability to not draw some of the original instances, as if they were removed from the visibility set.

NVIDIA Ex1023
Page Q-116

4.3 The Method

Let's assume that we have a list of instances of a specific model. The next step is to spatially organize the instances such that in a 1D array, objects that are near each other spatially are also near each other in the array. With this new, organized list, the final step is to stitch all of the individual instances together as if they were one big instance, recording which parts in the new vertex/index buffers belong to the individual instances. We call each record in this list a segment. To render objects inside the segment buffer, simply generate a list of segments to be rendered while merging sequential segments in the list. This procedure thus generates a new list of optimally merged sections of the vertex/index buffer to render. Segments are rendered as static batches (see Chapter 3 of this book, "Inside Geometry Instancing").

4.3.1 Segment Buffering, Step 1

The key to making segment buffering work and work well is in the spatial organization of the segments inside the segment buffer. For this, we can use any spatial organization structure. K-d trees (Samet 1989, 1990), where k is 3, perform well for 2015's game *Men of Valor: Vietnam*. Octrees (Suter 1999) or any other spatial data structures would probably work just as well.

The tree is generated from a list of points that represent the locations of the instances for which we are creating a segment buffer. Once we have this tree, we do a depth-first traversal, always branching left first at every interior tree node. When we hit a leaf, we add all the points contained in the leaf to the new ordered list. When the traversal is complete, we have a spatially ordered list of instances.

The segment buffers and the spatially ordered list of instances need to be generated only when a new list of instances is introduced. In the majority of cases, then, the segment buffers and ordered list of instances need to be created only at load time.

4.3.2 Segment Buffering, Step 2

Given a spatially ordered list of instances, we iterate through the list and construct a single vertex/index buffer containing all of the instances. We make sure to transform each instance's vertex buffer components into world space before the instance is written into the big vertex buffer. Also, we need to record what parts of the big vertex/index

NVIDIA Ex1023
Page Q-117

buffer belong to each instance, for later reference during rendering. Again, this big vertex/index buffer needs to be created only in the event a new list of instances is introduced, typically at load time.

4.3.3 Segment Buffering, Step 3

When rendering, instead of immediately drawing an instance that is segmented, we generate a list of all these instances that need to be rendered. Instances outside the view frustum, for example, need not be rendered and should be omitted. Then we translate this list of instances into the parts of the big vertex/index buffer we previously generated for segment buffering. While doing this, we can merge segments adjacent to one another in the big vertex/index buffer to achieve an optimal list of the pieces of those buffers to be rendered. The result is the exact same output as if the instance list was rendered individually without segment buffering.

4.4 Improving the Technique

To make segment buffering a viable technique for a game that uses light mapping extensively, we need to implement automatic texture-atlas generation (NVIDIA 2004). This is because we cannot segment instances together that require render-state changes in between the rendering of each instance. In scenes where every instance has its own light map, a texture change is required between the rendering of each instance. Automatic texture-atlas generation is fairly easy to implement in the context of segment buffering because every instance inside the big vertex buffer has a unique section. Thus, the texture coordinates can be directly modified without affecting any other instance. This is in contrast to the traditional method of modifying the texture coordinates inside a vertex or pixel shader.

4.5 Conclusion

In this chapter, we have described a technique to significantly reduce the number of batches rendered in a single displayed frame. In doing so, the technique enables us to create a much richer and more realistic-looking environment. In addition to segment buffering, we described enhancements that make it applicable in situations where it normally would not be appropriate.

NVIDIA Ex1023
Page Q-118

4.6 References

NVIDIA Corporation. 2004. "Improve Batching Using Texture Atlases." SDK white paper. **http://download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf**

Samet, Hanan. 1989. *Applications of Spatial Data Structures*. Addison-Wesley.

Samet, Hanan. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.

Suter, Jaap. 1999. "Introduction to Octrees." Flipcode Web site. **http://www.flipcode.com/tutorials/tut_octrees.shtml**

Wloka, Matthias. 2003. "Batch, Batch, Batch: What Does It Really Mean?" Presentation at Game Developers Conference 2003. **http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf**

NVIDIA Ex1023
Page Q-119



Chapter 5

Optimizing Resource Management with Multistreaming

Oliver Hoeller
Piranha Bytes

Kurt Pelzer
Piranha Bytes

One of the most difficult problems in modern real-time graphics applications is the massive amount of data that must be managed. Complex scenes with complex meshes combined with multiple render passes—as are needed for high-quality shadows and additional reflection and refraction effects, for example—are expensive to render. Finding ways to feed the GPU with optimally formatted data is crucial for successfully rendering complex scenes.

This chapter addresses this issue and focuses on a flexible model to resolve the problem of handling the gamut of graphics hardware found in current PCs. A PC game must be able to run and run well on the latest high-end GPU all the way down to last year's "value" GPU—and on everything in between. We introduce a solution that handles the massive amount of data and is careful to transmit only the currently needed vertex components for each pass. This system, which has been implemented in the *Gothic III* engine, combines two powerful techniques: several vertex buffers are combined via multistreaming (a feature introduced with Microsoft DirectX 8.0), and each vertex buffer is controlled by an optimized resource manager. We present both the abstract concept as well as its implementation based on DirectX 9.0c.

Introduction

NVIDIA Ex1023
Page Q-120

5.1 Overview

Each vertex in a mesh has multiple components associated with it (position, normal, tangent, texture coordinates, and so on), but all of these entities aren't always necessary when rendering the object. We want an automated system that uses only the currently needed vertex components. To handle the task, we use vertex buffer streams specialized for certain subtasks. Depending on the current rendering pass, subsets of these streams are combined to assemble the vertices.

We need four basic types of streams, as shown in Figure 5-1. Here are the streams and their subtasks:

- G—**Vertex stream for geometry data.** Contains vertex position, normal, and vertex color(s).
- T—**Vertex stream for texture-mapping data.** Holds texture coordinate sets and additional information such as tangent vectors for tangent-space normal maps.
- A—**Vertex stream for animation data.** Holds animation data such as bone weights and influences.
- I—**Vertex stream for instancing data.** Contains data for vertex stream frequency instancing.

Subsets of these four streams combine to handle different tasks, as shown in Figure 5-2:

- **Render meshes without animation**
  Possible stream combinations: G *or* G + T
- **Render meshes with animation**
  Possible stream combinations: G + A *or* G + T + A
- **Render instanced meshes (optionally with animation)**
  Possible stream combinations: G + I *or* G + T + I
  (Optional: G + A + I *or* G + T + A + I)
- **Render pure z-pass (optionally with or without instancing, with or without animation)**
  Possible stream combinations: G
  (Optional: G + A *or* G + I *or* G + A + I)

The next section presents an implementation of this model.

NVIDIA Ex1023
Page Q-121

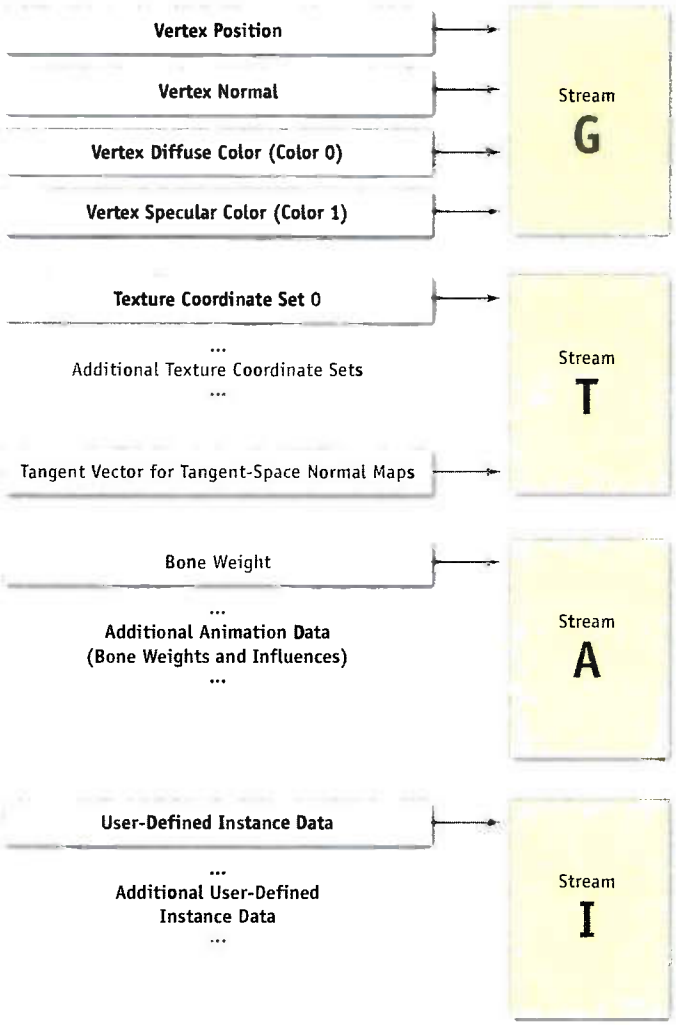


Figure 5-1. Four Types of Vertex Streams

5.2 Implementation

Now we show how to implement the abstract concept discussed in Section 5.1. First we examine some multistreaming example code based on DirectX 9.0c. Next we present the resource manager that handles the mesh resources. Finally, we show how to translate generic mesh data into the appropriate hardware structure.

NVIDIA Ex1023
Page Q-122

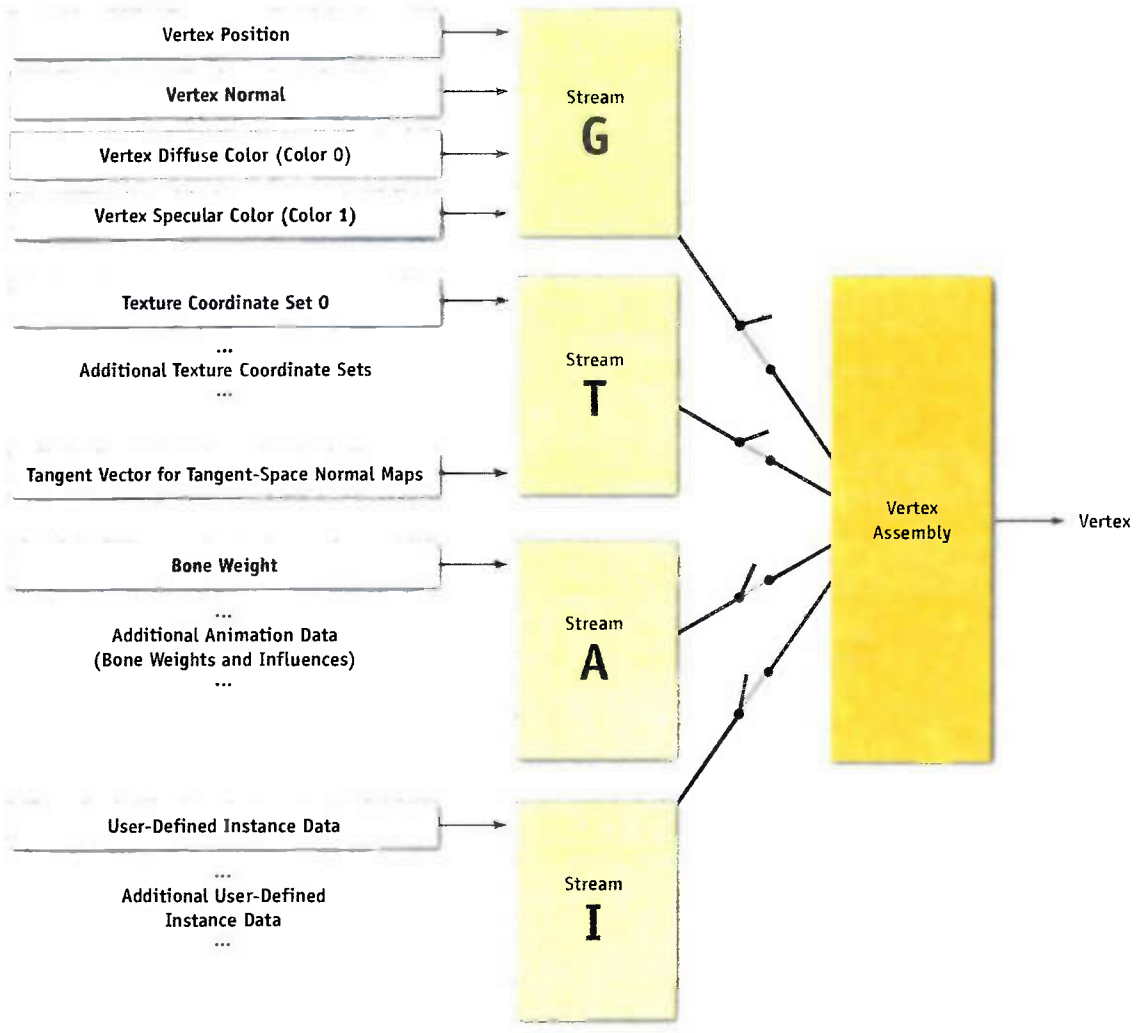


Figure 5-2. Combining Currently Needed Streams

5.2.1 Multistreaming with DirectX 9.0

Microsoft DirectX 8.0 introduced the notion of a stream to bind data to input registers. A stream is a uniform array of component data. Each component consists of one or more elements that represent a single entity, such as position, normal, color, texture coordinate set, and so on. With streams, GPUs are able to perform a direct memory access (DMA) from multiple vertex buffers in parallel. The Direct3D caps bit

NVIDIA Ex1023
Page Q-123

`D3DCAPS9.MaxStreams` defines how many streams a GPU supports; modern hardware supports up to 16 streams, while older, DirectX 8.0-compliant GPUs are limited to 8 streams. Because multistreaming has some minor performance implications, it's advisable to minimize the number of active streams. Because our implementation uses only 1 to 4 streams, all GPUs with multistreaming support can be targeted, with minimal loss of performance.

Listing 5-1 shows our multistreaming example code. Using DirectX 9.0 for this task, we need only three simple components:

- A correct vertex declaration (an array of `D3DVERTEXELEMENT9`) and the `IDirect3DDevice9::CreateVertexDeclaration()` method to create a vertex declaration object.
- The `IDirect3DDevice9::SetVertexDeclaration()` method to set the vertex declaration object.
- The `IDirect3DDevice9::SetStreamSource()` method to bind a vertex buffer to a device data stream, creating an association between the vertex data and one of several data stream ports that feed the primitive processing functions.

Listing 5-1. A Set of Vertex Definitions and a Vertex Declaration

```
//
// Initializations:
//
// Here is a set of vertex definitions to support two streams.
// Vertex format:
//     stream 0 -> position + normal + color 0 + color 1
//     stream 1 -> 4 texture coordinate pairs
struct VTXSTREAM_0
{
  float fPosX, fPosY, fPosZ;
  float fNormX, fNormY, fNormZ;
  DWORD dwColor0, dwColor1;
};

struct VTXSTREAM_1
{
  float fTexU0, fTexV0;
  float fTexU1, fTexV1;
  float fTexU2, fTexV2;
  float fTexU3, fTexV3;
};
```


NVIDIA Ex1023
Page Q-124

Listing 5-1 *(continued)*. A Set of Vertex Definitions and a Vertex Declaration

```
// Vertex declaration
D3DVERTEXELEMENT9 m_VtxDcl[] =
{
  {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_POSITION, 0}, // stream 0, position
  {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_NORMAL, 0},   // stream 0, normal
  {0, 24, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_COLOR, 0},    // stream 0, color 0
  {0, 28, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_COLOR, 1},    // stream 0, color 1
  {1, 0, D3DDECLUSAGE_FLOAT2, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_TEXCOORD, 0}, // stream 1, tex coord set 0
  {1, 8, D3DDECLUSAGE_FLOAT2, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_TEXCOORD, 1}, // stream 1, tex coord set 1
  {1, 16, D3DDECLUSAGE_FLOAT2, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_TEXCOORD, 2}, // stream 1, tex coord set 2
  {1, 24, D3DDECLUSAGE_FLOAT2, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_TEXCOORD, 3}  // stream 1, tex coord set 3
}

// Create a vertex declaration object
LPDIRECT3DVERTEXDECLARATION9 m_pVtxDeclObject;
m_pd3dDevice->CreateVertexDeclaration(m_VtxDcl,&m_pVtxDclObj);
```

In Listing 5-1, we present a set of vertex definitions and create a vertex declaration object `m_pVtxDeclObject` that describes the following vertex data layout:

- Vertex position data are found in stream 0 with a 0-byte offset. The data type is three-component float that expands to (float, float, float, 1).
- Vertex normal data are found in stream 0 with a 12-byte offset. The data type is three-component float that expands to (float, float, float, 1).
- Diffuse and specular color are found in stream 0 with offsets of 24 bytes (diffuse) and 28 bytes (specular). The data type is four-component, packed, unsigned bytes mapped to a range of 0 to 1.
- Texture coordinate data (sets 0 to 3) are found in stream 1 with offsets of 0 bytes (set 0), 8 bytes (set 1), 16 bytes (set 2), and 24 bytes (set 3). The data type is two-component float that expands to (float, float, 0, 1).

NVIDIA Ex1023
Page Q-125

Before calling a draw method, we must set this vertex declaration object together with the vertex buffers bound to the correct device data streams, as shown in Listing 5-2.

Listing 5-2. Binding Vertex Buffers to Streams

```
//
// Each time we want to render primitives:
//

// Set the vertex declaration object
m_pd3dDevice->SetVertexDeclaration(m_pVtxDclObj);

// Bind vertex buffers to device data streams
m_pd3dDevice->SetStreamSource(0, m_pVtxBuf0, 0, sizeof(VTXSTREAM_0));
m_pd3dDevice->SetStreamSource(1, m_pVtxBuf1, 0, sizeof(VTXSTREAM_1));

// Render ..
m_pd3dDevice->DrawIndexedPrimitive( .. );
```

The actual references to the stream data do not occur until a drawing method, such as `IDirect3DDevice9::DrawIndexedPrimitive()`, is called.

5.2.2 Resource Management

In the *Gothic III* engine, we employ the concept of *resource management* to handle the administration of all data-intensive elements, such as vertex data (mesh information), textures, animation data (key frames and skinning data), sounds, and music. All these data structures use a uniform framework. It provides a simple interface so that other systems of the engine can access it.

All resources are packed into abstract data objects. We address these data objects by file name or by a globally unique identifier (GUID). The access of data containers is similarly organized like a database access. If you call a query, you get a referenceable data container with all necessary structures of the specified data.

The resource management system ensures the immediate and simultaneous availability of the data. It is also responsible for converting this proprietary data into the graphics hardware's preferred format.

As we mentioned, the resource management system makes it possible to store mesh data in specified file locations and to load these mesh objects only as needed. These

NVIDIA Ex1023
Page Q-126

structures must be stored in an efficient and flexible way, but vertex data layout can vary substantially from mesh to mesh.

The loaded resources should provide a reference counter, so that the same mesh can be used repeatedly in a scene. Geometry instancing is used to speed up rendering this type of mesh. (See Chapter 3 of this book, "Inside Geometry Instancing.")

The mesh information is generated by tools such as 3ds max or Maya. For this operation, a suitable importer has to be developed, as shown in Figure 5-3.

The Mesh Resource

One of the database containers provided by the resource management system is a mesh resource, which has all relevant structures, all vertex and index data arrays, and a link to an abstract material object (which is also a referenceable resource).

Note that the vertex structure wrapped into a resource class (as shown in Figure 5-3) isn't adapted to any specific graphics API (such as DirectX or OpenGL). We can therefore implement the whole resource system independent of the graphics API used.

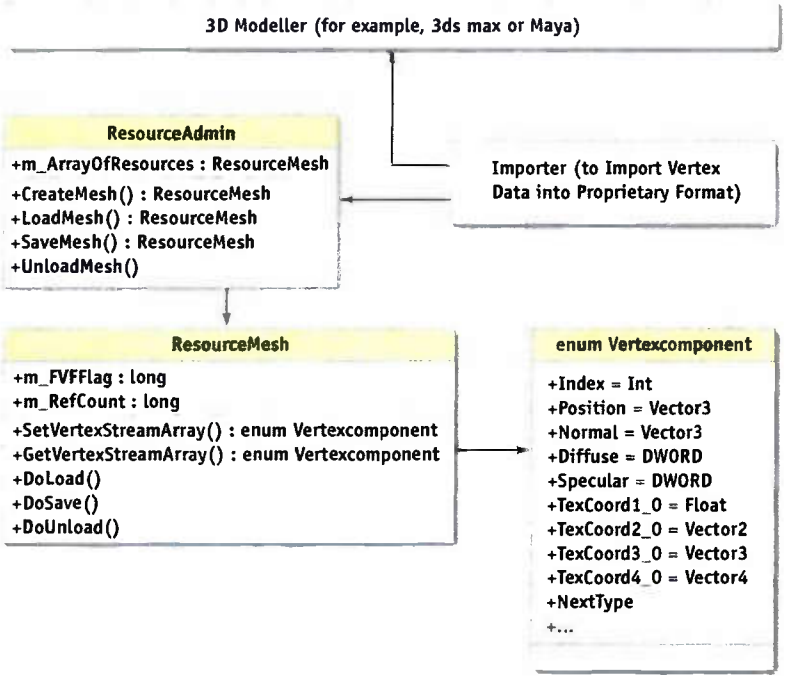


Figure 5-3. Structure of the Resource Framework

NVIDIA Ex1023
Page Q-127

These structures consist of simple vertex streams (arrays) containing a single vertex component, such as position (an array of 3D vectors), normals (an array of 3D vectors), color (an array of `DWORD` values), or up to 16 texture coordinates (an array of 1D, 2D, 3D, or 4D vectors) and other customized data arrays such as weights for hardware skinning.

To complete this groundwork, we have developed a hardware-independent resource framework with a mesh class and its administrative interface. Furthermore, we have written a specified importer for the corresponding modeling tool.

5.2.3 Processing Vertices

The renderer takes over the task of changing the hardware-independent data into a format that is optimal for the graphics hardware.

First, the mesh resources are loaded into main memory at the start of the application. Alternatively, we use an on-demand approach to load the data into memory when needed, using a separate thread.

For the renderer, the mesh object is only a temporary container to build up the correct format. The generated hardware-dependent format is used by the application to draw the scene. The reason for this strict separation is the ability to tailor the data optimally for the GPU found in the PC on which the application is currently running. We use only the vertex data that are actually needed.

This separation becomes useful in the adaptation of needed texture coordinates: On less powerful hardware, many visual effects are turned off because the hardware is incapable of running them, or because the overall rendering load needs to be reduced. Because these effects are not rendered, the texture coordinates used by these effects need not be passed to the graphics hardware. They can simply be removed. So we reduce memory consumption and save processor power and bandwidth.

For this task, we provide a submodule, which we have named Vertexprocessor, that translates the generic mesh data into the specified hardware structure. The following steps are necessary for the data to be prepared for multistreaming:

- When a mesh resource is requested, Vertexprocessor loads this data into memory if the object isn't already available, or increments the reference counter of the mesh object if it is already loaded.
- The stored vertex format of the mesh object is determined. This can happen via the embedded FVF flag or with the implementation used in the resource class (hardware independent).

NVIDIA Ex1023
Page Q-128

- The Vertexprocessor divides the single generic vertex arrays into the four hardware-specific vertex streams (that is, streams **G**, **T**, **A**, and **I**, discussed in Section 5.1). The vertex arrays are accessed by the `MeshResource::GetVertexStreamArray()` method, as shown in Figure 5-4, which translates or copies the arrays into the specific component of one of the four vertex buffers.

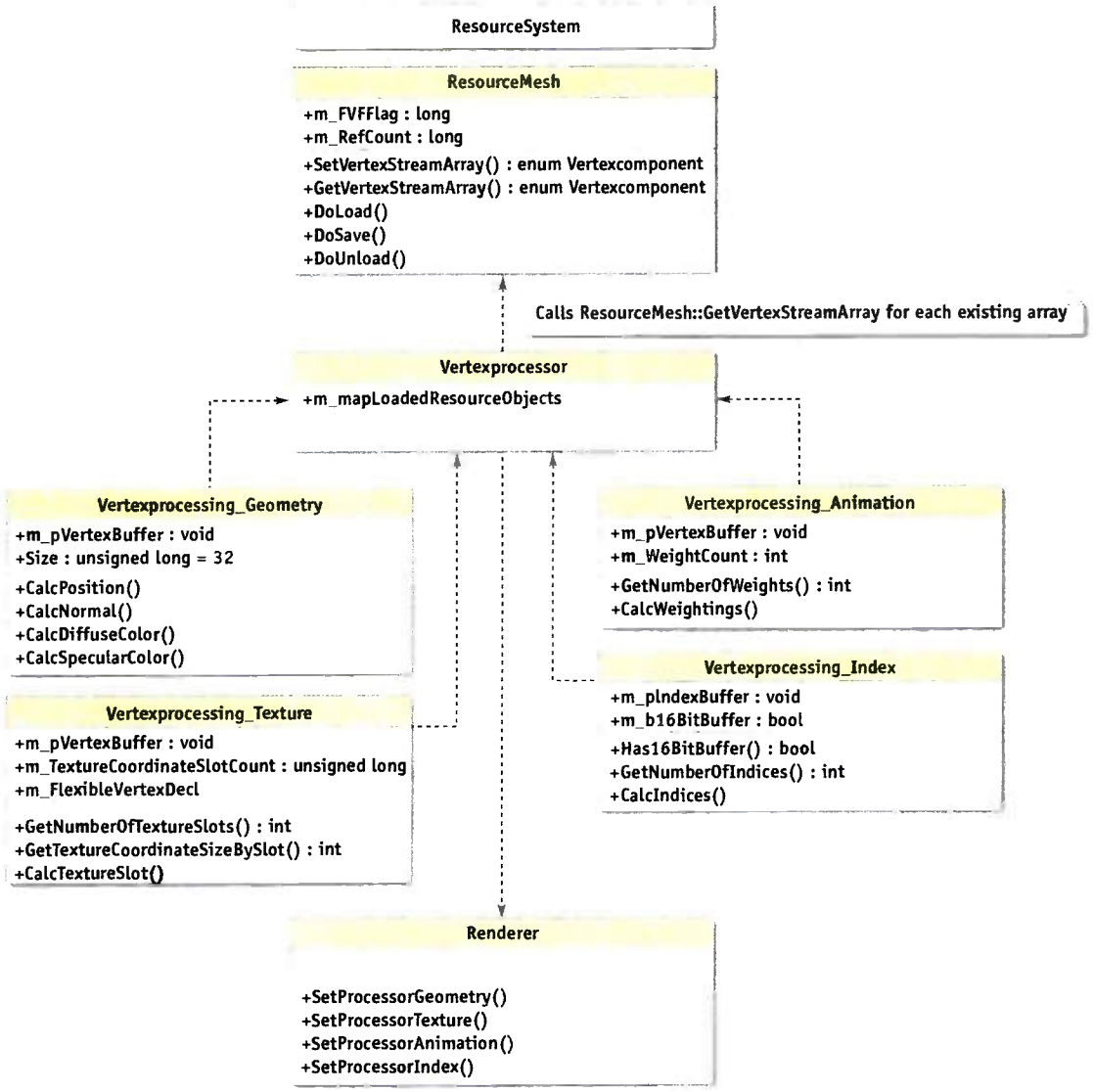


Figure 5-4. Query of Single Vertex Streams Used by Vertexprocessor

NVIDIA Ex1023
Page Q-129

Geometry Data

The Vertexprocessor unit for geometry data seeks position, normals, and the two colors (if available) from a mesh object.

For illustration purposes, Listing 5-3 creates a vertex buffer that contains all this information and uses this specific vertex layout. The buffer should have a cache-friendly size of 32 bytes per entry, so the layout is fixed for this buffer type.

Listing 5-3. Creating a Vertex Buffer

```
D3DVERTEXELEMENT9 m_VtxDcl[] =
{
  {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_POSITION, 0}, // stream 0, position
  {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_NORMAL, 0},   // stream 0, normal
  {0, 24, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_COLOR, 0},    // stream 0, color 0
  {0, 28, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
   D3DDECLUSAGE_COLOR, 0},    // stream 0, color 1
};

// Here comes the vertex definition.
// Vertex format:
//     stream 0 -> position + normal + color 0 + color 1
struct sVTXStreamGeometry
{
  float vecPos[ 3 ];
  float vecNorm[ 3 ];
  DWORD dwColor0, dwColor1;
};

// Data container Resourcemesh supports GetVertexStreamArray here.
// (See API methods above.)
cResourceMesh* m_pResourceMesh;

// Create the vertex declaration and create a specific vertex buffer.
// For clarity, we calculate the size of the buffer in the number of
// elements and the corresponding FVF flags. In our engine implementation,
// the geometry stream has a size of 32 bytes (cache-friendly), so we
// can easily create the sufficient memory size of the vertex buffer.
CreateVertexDeclaration( m_VtxDcl );
m_pVtxBuf0 = CreateVertexBuffer(NumVertices,
             FVF_XYZ | FVF_NORMAL | FVF_DIFFUSE | FVF_SPECULAR);
```


NVIDIA Ex1023
Page Q-130

Listing 5-3 *(continued)*. Creating a Vertex Buffer

```
sVTXStreamGeometry* pBuffer = LockBuffer(m_pVtxBuf0);
for each vertexIndex in m_pResourceMesh
{
  pBuffer->m_vecPos =
   m_pResourceMesh->GetVertexStreamArray(enum_Position, vertexIndex);
  pBuffer->m_vecNorm =
   m_pResourceMesh->GetVertexStreamArray(enum_Normal, vertexIndex);
  pBuffer->m_dwColor0 =
   m_pResourceMesh->GetVertexStreamArray(enum_Diffuse, vertexIndex);
  pBuffer->m_dwColor1 =
   m_pResourceMesh->GetVertexStreamArray(enum_Specular, vertexIndex);
  pBuffer++;
}
```

Texture Data

The Vertexprocessor unit for texture data fetches the available texture coordinate slots from the mesh object and builds up a flexible vertex buffer with all necessary texture coordinates. At most, eight texture coordinate groups (with a 1D, 2D, 3D, or 4D vector per texture coordinate) are possible.

In addition, one of the slots can be encoded as tangent vectors for tangent-space normal maps; otherwise, it would be necessary to use a separate vertex buffer stream for this information. We omit this in Listing 5-4 for clarity. It is, however, straightforward to add this feature.

If light maps are available, we should use a separate Vertexprocessor unit (that is, a separate vertex buffer per mesh instance), because every instance must have unique UV pairs for light map textures. Light maps must be individually mapped, and their mapping coordinates depend on their world-space position, and so they can't be instantiated. This can be a big problem for the memory footprint and performance of your application. Consequently, light maps are useful for big meshes, such as terrain in outdoor scenarios or irregular wall meshes in dungeons, but not for small meshes or multi-instantiated objects.

Listing 5-4. A Vertex Buffer Stream for Texture Data

```
// Data container Resourcemesh supports GetVertexStreamArray here.
// (See API methods above.)
cResourceMesh* m_pResourceMesh;
```


NVIDIA Ex1023
Page Q-131

Listing 5-4 *(continued)*. A Vertex Buffer Stream for Texture Data

```
// This is a template array structure to allocate a flexible amount
// of vtxDcl structures in one block
Array<D3DVERTEXELEMENT9> arrDeclVtx;

// We create a specific vertex buffer for textures. We use the FVF flag
// from Resourcemesh and filter the specific texture flags to create
// the specific buffer.
// For clarity, we use here a function that calculates the size of the
// buffer and the corresponding FVF flags. You can use an alternative
// implementation to calculate the correct size of the vertex buffer.
sVTXStreamTexture* pVtxBuf1 =
 CreateVertexBuffer(NumTextureEntries,
                    m_pResourceMesh->m_FVFFlag & TextureFlags);
// Lock buffer with unspecific byte pointer
char* pBuffer = LockBuffer( pVtxBuf1 );

// Current buffer position (in bytes) in locked data of vertex buffer
DWORD dwGlobalBufferPosition = 0;
for each textureIndex in m_pResourceMesh
{
  DWORD dwCurrentOffset = 0; // Current offset position for vtxDecl

  for (int i = 0; i<m_pResourceMesh->GetNumberOfTextureSlots(); i++)
  {
    // Array template allocates us a new structure at the end of the
    // structure block and returns a nonconstant reference of the block.
    // All structures allocated before remain unchanged (by realloc).
    D3DVERTEXELEMENT9& flexibleElement = arrDeclVtx.InsertNewEnd();

    flexibleElement.Stream = 1; // Is at 2nd position after Geometry
    flexibleElement.Usage  = D3DDECLUSAGE_TEXCOORD;

    // Current size in bytes of texture slots (u,v = 8 bytes)
    // r,s,t = 12 bytes, r,s,t,w = 16 bytes
    DWORD dwCurrentTextureSlotSize =
          m_pResourceMesh->GetTextureCoordinateSizeBySlot( i );

    // For specified slot, we look for what size (in bytes) we need
    // and add it to our global offset counter dwCurrentOffset
    flexibleElement.Offset = dwCurrentOffset;
```


NVIDIA Ex1023
Page Q-132

Listing 5-4 *(continued)*. A Vertex Buffer Stream for Texture Data

```
        // Calculate stream position in Resourcemesh for specified
        // texture slot
        int enumStreamArraySlot =
           m_pResourceMesh->GetTextureSlotEnum( i );

        // Now copy out the vertex information from the Resourcemesh
        // stream into locked buffer
        memcpy(&pBuffer[ dwGlobalBufferPosition + dwCurrentOffset ],
         m_pResourceMesh->GetVertexStreamArray(enumStreamArraySlot,
          textureindex), dwCurrentTextureSlotSize);

        dwCurrentOffset += dwCurrentTextureSlotSize;

        // Calculate types of vtxDecl (D3DDECLTYPE_FLOAT2 = 8 bytes,
        // D3DDECLTYPE_FLOAT3 = 12 bytes, D3DDECLTYPE_FLOAT4 = 16 bytes)
        flexibleElement.Type =
          CalculateDataSizeType(dwCurrentTextureSlotSize);

    }
    // Add new position offset to global position in buffer
    dwGlobalBufferPosition += dwCurrentOffset;
}
```

Animation Data

The Vertexprocessor unit for animation data builds up weights for animation skinning with bones in hardware. A varying number of weights per bone can be used, so it is not necessary to waste more memory than is really needed. In *Gothic III*, we use up to six weights per bone for our animation system.

Index Data

An index buffer corresponding to the above vertex buffers is also built up. Indices are stored into a mesh object as a separate stream of information. The bit size of index entries (whether 16 or 32 bit) depends on the number of vertices. Having more than 65,535 entries necessitates using `DWORD`s as index entries. Otherwise, `WORD` (16-bit) indices suffice. Choosing the smallest possible data type helps reduce an application's memory footprint.

NVIDIA Ex1023
Page Q-133

The Vertexprocessor has the responsibility to create all the previously described vertex buffers in an optimal manner for hardware (typically they are created as `Pool_Managed` and `WriteOnly`)(Ashida 2004).

Rendering the Streams

Once all the streams are built up properly, they are available for rendering and drawing.

As an example of using streams separately, we execute the G-stream (that is, the geometry stream) as a separate z-pass, to achieve fast z-rejects in hardware and to use this buffer in conjunction with an occlusion query system implemented in the render system. (See Chapter 6 of this book, "Hardware Occlusion Queries Made Useful," for an example of such a system.)

It is possible to add the A-stream (the animation stream) in the calculation, but the effort isn't worthwhile in most cases. The number of pixels that differ because of the animation is typically small and thus out of proportion to the additional rendering cost of adding animation.

Individual streams are activated depending on the view (whether solid rendering or z-pass) by the renderer.

5.3 Conclusion

In this chapter, we have shown how current applications can overcome problems caused by the growing amount of geometry data in scenes. We've discussed a flexible model that gives the application more control over the data and drives the detected hardware optimally by combining two powerful techniques:

- Several vertex buffers are combined via multistreaming.
- Each vertex buffer is controlled by an optimized resource manager.

A nice side benefit: we efficiently handle the bandwidth for data, which sometimes can be limited because of data duplication/redundant data transmission across the bus from system memory to the GPU.

NVIDIA Ex1023
Page Q-134

5.4 References

Ashida, Koji. 2004. "Optimising the Graphics Pipeline." NVIDIA presentation. **http://developer.nvidia.com/docs/IO/10878/ChinaJoy2004_OptimizationAndTools.pdf**

Cebenoyan, Cem. 2004. "Graphics Pipeline Performance." In *GPU Gems*, edited by Randima Fernando, pp. 473–486. Addison-Wesley. Available online at **http://developer.nvidia.com/object/GPU_Gems_Samples.html**

For additional information about programming one or more streams in DirectX, see the Microsoft Web site:
http://msdn.microsoft.com/directx

NVIDIA Ex1023
Page Q-135



Chapter 6

Hardware Occlusion Queries Made Useful

Michael Wimmer
Vienna University of Technology

Jiří Bittner
Vienna University of Technology

6.1 Introduction

Hardware occlusion queries were one of the most eagerly awaited graphics hardware features in a long time. This feature makes it possible for an application to ask the 3D API (OpenGL or Direct3D) whether or not any pixels would be drawn if a particular object were rendered. With this feature, applications can check whether or not the bounding boxes of complex objects are visible; if the bounds are occluded, the application can skip drawing those objects. Hardware occlusion queries are appealing to today's games because they work in completely dynamic scenes.

However, although this feature has been available for more than two years (via the `NV/ARB_occlusion_query` OpenGL extension and by the `IDirect3DQuery9` interface in Direct3D), there is not yet widespread adoption of this feature for solving visibility culling in rendering engines. This is due to two main problems related to naive usage of the occlusion query feature: the overhead caused by issuing the occlusion queries themselves (since each query adds an additional draw call), and the latency caused by waiting for the query results.

In this chapter, we present a simple but powerful algorithm that solves these problems (Bittner et al. 2004): it minimizes the number of issued queries and reduces the impact of delays due to the latency of query results.

NVIDIA Ex1023
Page Q-136

To achieve this, the algorithm exploits the spatial and temporal coherence of visibility by reusing the results of occlusion queries from the previous frame in order to initiate and schedule the queries in the current frame. This is done by storing the scene in a hierarchical data structure (such as a k-d tree or octree [Cohen-Or et al. 2003]), processing nodes of the hierarchy in a front-to-back order, and interleaving occlusion queries with the rendering of certain previously visible nodes.

The algorithm is smart about the queries it actually needs to issue: most visible interior nodes of the spatial hierarchy and many small invisible nodes are not tested at all. Furthermore, previously visible nodes are rendered immediately without waiting for their query result, which allows filling the time needed to wait for query results with useful work.

6.2 For Which Scenes Are Occlusion Queries Effective?

Let's recap briefly for which scenes and situations the algorithm presented in this chapter is useful.

Occlusion queries work best for large scenes where only a small portion of the scene is visible each frame—for example, a walkthrough in a city. More generally, the algorithm presented here works if in each frame there are large occluded interior nodes in the hierarchy. These nodes should contain many objects, so that skipping them if they are occluded saves geometry, overdraw, and draw calls (which are a typical bottleneck in today's rendering systems [Wloka 2003]). An important advantage the algorithm has over most other techniques used nowadays in games (such as portal culling) is that it allows completely dynamic environments.

Some games—for example, shader-heavy ones—use a separate initial rendering pass that writes only to the depth buffer. Subsequent passes test against this depth buffer, and thus expensive shading is done only for visible pixels. If the scenes are complex, our occlusion-culling algorithm can be used to accelerate establishing the initial depth buffer and at the same time to obtain a complete visibility classification of the hierarchy. For the subsequent passes, we skip over objects or whole groups of objects that are completely invisible at virtually no cost.

Finally, if there is little or no occlusion in a scene—for example, a flyover of a city—there is no benefit to occlusion culling. Indeed, occlusion queries potentially introduce an overhead in these scenes and make rendering slower, no matter how the occlusion queries are used. If an application uses several modes of navigation, it will pay to switch off occlusion culling for modes in which there is no or only very little occlusion.

NVIDIA Ex1023
Page Q-137

6.3 What Is Occlusion Culling?

The term *occlusion culling* refers to a method that tries to reduce the rendering load on the graphics system by eliminating (that is, culling) objects from the rendering pipeline if they are hidden (that is, occluded) by other objects. There are several methods for doing this.

One way to do occlusion culling is as a preprocess: for any viewpoint (or regions of viewpoints), compute ahead of time what is and is not visible. This technique relies on most of the scene to be static (so visibility relationships do not change) and is thus not applicable for many interactive and dynamic applications (Cohen-Or et al. 2003).

Portal culling is a special case of occlusion culling; a scene is divided into cells (typically rooms) connected via portals (typically door and window openings). This structure is used to find which cells (and objects) are visible from which viewpoint—either in a preprocess or on the fly (Cohen-Or et al. 2003). Like general preprocess occlusion culling, it relies on a largely static scene description, and the room metaphor restricts it to mostly indoor environments.

Online occlusion culling is more general in that it works for fully dynamic scenes. Typical online occlusion-culling techniques work in image space to reduce computation overhead. Even then, however, CPUs are less efficient than, say, GPUs for performing rasterization, and thus CPU-based online occlusion techniques are typically expensive (Cohen-Or et al. 2003).

Fortunately, graphics hardware is very good at rasterization. Recently, an OpenGL extension called `NV_occlusion_query`, or now `ARB_occlusion_query`, and Direct3D's occlusion query (`D3DQUERYTYPE_OCCLUSION`) allow us to query the number of rasterized fragments for any sequence of rendering commands. Testing a single complex object for occlusion works like this (see also Sekulic 2004):

1. Initiate an occlusion query.
2. Turn off writing to the frame and depth buffer, and disable any superfluous state. Modern graphics hardware is thus able to rasterize at a much higher speed (NVIDIA 2004).
3. Render a simple but conservative approximation of the complex object—usually a bounding box: the GPU counts the number of fragments that would actually have passed the depth test.
4. Terminate the occlusion query.

NVIDIA Ex1023
Page Q-138

5. Ask for the result of the query (that is, the number of visible pixels of the approximate geometry).
6. If the number of pixels drawn is greater than some threshold (typically zero), render the complex object.

The approximation used in step 3 should be simple so as to speed up the rendering process, but it must cover at least as much screen-space area as the original object, so that the occlusion query does not erroneously classify an object as invisible when it actually is visible. The approximation should thus be much faster to render, and does not modify the frame buffer in any way.

This method works well if the tested object is really complex, but step 5 involves waiting until the result of the query actually becomes available. Since, for example, Direct3D allows a graphics driver to buffer up to three frames of rendering commands, waiting for a query results in potentially large delays. In the rest of this chapter, we refer to steps 1 through 4 as "issuing an occlusion query for an object."

If correctness of the rendered images is not important, a simple way to avoid the delays is to check for the results of the queries only in the following frame (Sekulic 2004). This obviously leads to visible objects being omitted from rendering, which we avoid in this chapter.

6.4 Hierarchical Stop-and-Wait Method

As a first attempt to use occlusion queries, we show a naive occlusion-culling algorithm and extend it to use a hierarchy. In the following section, we show our coherent hierarchical culling algorithm, which makes use of coherence and other tricks to be much more effective than the naive approach.

6.4.1 The Naive Algorithm, or Why Use Hierarchies at All?

To understand why we want to use a hierarchical algorithm, let's take a look at the naive occlusion query algorithm first:

1. Sort objects front to back.
2. For each object
   a. Issue occlusion query for the object (steps 1–4 from previous section).
   b. Stop and wait for result of query.
   c. If number of visible pixels is greater than 0, render the object.

NVIDIA Ex1023
Page Q-139

Although this algorithm calculates correct occlusion information for all of our objects, it is most likely slower than just rendering all the objects directly. The reason is that we have to issue a query and wait for its result for every object in the scene!

Now imagine, for example, a walkthrough of a city scene: We typically see a few hundred objects on screen (buildings, streetlights, cars, and more). But there might be tens of thousands of objects in the scene, most of which are hidden by nearby buildings. If each of these objects is not very complex, then issuing and waiting for queries for all of them is more expensive than just rendering them.

6.4.2 Hierarchies to the Rescue!

We need a mechanism to group objects close to one another so we can treat them as a single object for the occlusion query. That's just what a spatial hierarchy does. Examples of spatial hierarchies are k-d trees, BSP trees, and even standard bounding-volume hierarchies. They all have in common that they partition the scene recursively until the cells of the partition are "small" enough according to some criterion. The result is a tree structure with interior nodes that group other nodes, and leaf nodes that contain actual geometry.

The big advantage of using hierarchies for occlusion queries is that we can now test interior nodes, which contain much more geometry than the individual objects. In our city example, hundreds or even thousands of objects might be grouped into one individual node. If we issue an occlusion query for this node, we save the tests of all of these objects—a potentially huge gain!

If geometry is not the main rendering bottleneck, but rather the number of draw calls issued (Wloka 2003), then making additional draw calls to issue the occlusion queries is a performance loss. With hierarchies, though, interior nodes group a larger number of draw calls, which are all saved if the node is occluded using a single query. So we see that in some cases, a hierarchy is what makes it possible to gain anything at all by using occlusion queries.

6.4.3 Hierarchical Algorithm

The naive hierarchical occlusion-culling algorithm works like this—it is specified for a node within the hierarchy and initially called for the root node:

1. Issue occlusion query for the node.
2. *Stop and wait* for the result of the query.

NVIDIA Ex1023
Page Q-140

3. If the node is visible:
   a. If it is an interior node:
      i. Sort the children in front-to-back order.
      ii. Call the algorithm recursively for all children.
   b. If it is a leaf node, render the objects contained in the node.

This algorithm can potentially be much faster than the basic naive algorithm, but it has two significant drawbacks.

6.4.4 Problem 1: Stalls

This approach shares the first drawback with the naive algorithm. Whenever we issue an occlusion query for a node, we cannot continue our algorithm until we know the result. But waiting until the query result becomes available may be prohibitive. The query has to be sent to the GPU. There it sits in a command queue until all previous rendering commands have been issued (and modern GPUs can queue up more than one frame's worth of rendering commands!). Then the bounding box associated with the query must be rasterized, and finally the result of the query has to travel back over the bus to the driver.

During all this time, the CPU sits idle, and we have caused a CPU stall. But that's not all. While the CPU sits idle, it doesn't feed the GPU any new rendering commands. Now when the result of the occlusion query arrives, and the CPU has finally figured out what to draw and which commands to feed the GPU next, the command buffer of the GPU has emptied and it has become idle for some time as well. We call this GPU starvation. Obviously, we are not making good use of our resources. Figure 6-1 sums up these problems.

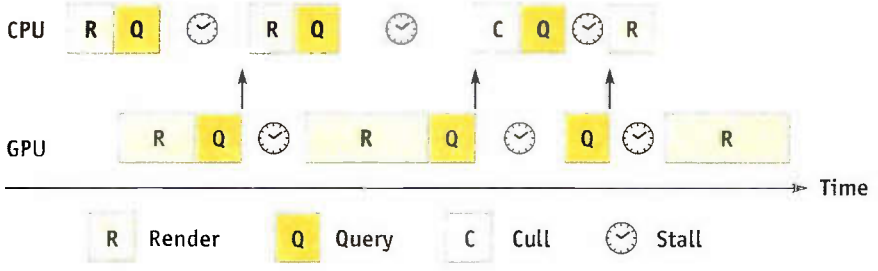


Figure 6-1. CPU Stalls and GPU Starvation Caused by Occlusion Queries

NVIDIA Ex1023
Page Q-141

6.4.5 Problem 2: Query Overhead

The second drawback of this algorithm runs contrary to the original intent of the hierarchical method. We wanted to reduce the overhead for the occlusion queries by grouping objects together. Unfortunately, this approach increases the number of queries (especially if many objects are visible): in addition to the queries for the original objects, we have to add queries for their groupings. So we have improved the good case (many objects are occluded), but the bad case (many objects are visible) is even slower than before.

The number of queries is not the only problem. The new queries are for interior nodes of the hierarchy, many of which, especially the root node and its children, will have bounding boxes covering almost the entire screen. In all likelihood, they are also visible. In the worst case, we might end up filling the entire screen tens of times just for rasterizing the bounding geometry of interior nodes.

6.5 Coherent Hierarchical Culling

As we have seen, the hierarchical stop-and-wait method does not make good use of occlusion queries. Let us try to improve on this algorithm now.

6.5.1 Idea 1: Being Smart and Guessing

To solve problem 1, we have to find a way to avoid the latency of the occlusion queries. Let's assume that we could "guess" the outcome of a query. We could then react to our guess instead of the actual result, meaning we don't have to wait for the result, eliminating CPU stalls and GPU starvations.

Now where does our guess come from? The answer lies, as it so often does in computer graphics, in temporal coherence. This is just another way of expressing the fact that in real-time graphics, things don't move around too much from one frame to the next. For our occlusion-culling algorithm, this means that if we know what's visible and what's occluded in one frame, it is very likely that the same classification will be correct for most objects in the following frame as well. So our "guess" is simply the classification from the previous frame.

But wait—there are two ways in which our guess can go wrong: If we assume the node to be visible and it's actually occluded, we will have processed the node needlessly. This

NVIDIA Ex1023
Page Q-142

can cost some time, but the image will be correct. However, if we guess that a node is occluded and in reality it isn't, we won't process it and some objects will be missing from the image—something we need to avoid!

So if we want to have correct images, we need to verify our guess and rectify our choice in case it was wrong. In the first case (the node was actually occluded), we update the classification for the next frame. In the second case (the node was actually visible), we just process (that is, traverse or render) the node normally. The good news is that we can do all of this later, whenever the query result arrives. Note also that the accuracy of our guess is not critical, because we are going to verify it anyway. Figure 6-2 shows the different cases.

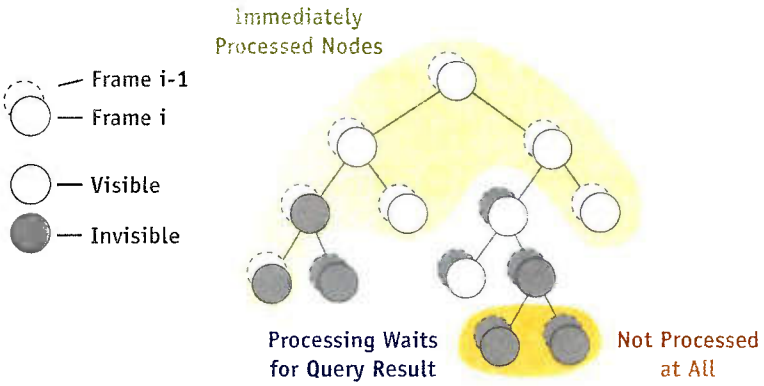


Figure 6-2. Processing Requirements for Various Nodes

6.5.2 Idea 2: Pull Up, Pull Up

To address problem 2, we need a way to reduce overhead caused by the occlusion queries for interior nodes.

Luckily, this is easy to solve. Using idea 1, we are already processing previously visible interior nodes without waiting for their query results anyway. It turns out that we don't even need to issue a query for these nodes, because at the end of the frame, this information can be easily deduced from the classification of its children, effectively "pulling up" visibility information in the tree. Therefore, we save a number of occlusion queries and a huge amount of fill rate.

On the other hand, occlusion queries for interior nodes that were occluded in the previous frame are essential. They are required to verify our choice not to process the node,

NVIDIA Ex1023
Page Q-143

and in case the choice was correct, we have saved rendering all the draw calls, geometry, and pixels that are below that node (that is, inside the bounding box of the node).

What this boils down to is that we issue occlusion queries only for previously visible leaf nodes and for the largest possible occluded nodes in the hierarchy (in other words, an occluded node is not tested if its parent is occluded as well). The number of queries issued is therefore linear in the number of visible objects. Figure 6-3 illustrates this approach.

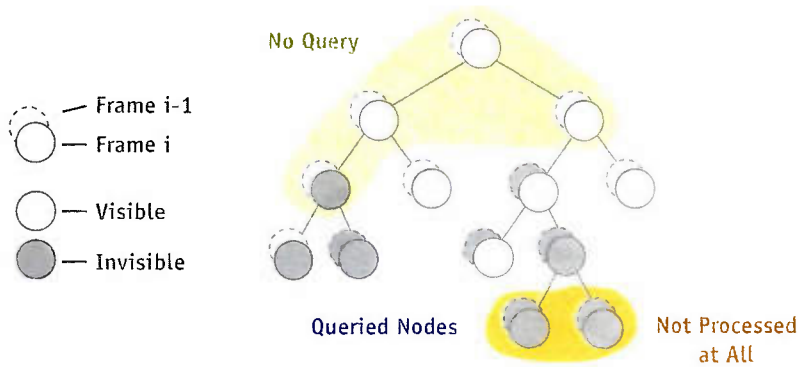


Figure 6-3. Occlusion Query Requirements for Various Nodes

6.5.3 Algorithm

We apply these two ideas in an algorithm we call "coherent hierarchical culling." In addition to the hierarchical data structure for the front-to-back traversal we already know, we need a "query queue" that remembers the queries we issued previously. We can then come back later to this queue to check whether the result for a query is already available.

The algorithm is easy to incorporate into any engine as it consists of a simple main traversal loop. A queue data structure is part of the C++ standard template library (STL), and a hierarchical data structure is part of most rendering libraries anyway; for example, for collision detection.

The algorithm visits the hierarchy in a front-to-back order and immediately recurses through any previously visible interior node (idea 1). For all other nodes, it issues an occlusion query and stores it in the query queue. If the node was a previously visible leaf node, it also renders it immediately, without waiting for the query result.

NVIDIA Ex1023
Page Q-144

During the traversal of the hierarchy, we want to make sure that we incorporate the query results as soon as possible, so that we can issue further queries if we encounter a change in visibility.

Therefore, after each visited node, we check the query queue to see if a result has already arrived. Any available query result is processed immediately. Queries that verify our guess to be correct are simply discarded and do not generate additional work. Queries that contradict our guess are handled as follows: Nodes that were previously visible and became occluded are marked as such. Nodes that were previously occluded and became visible are traversed recursively (interior nodes) or rendered (leaf nodes). Both of these cases cause visibility information to be pulled up through the tree. See Figure 6-4, which also depicts a "pull-down" situation: a previously occluded node has become visible and its children need to be traversed.

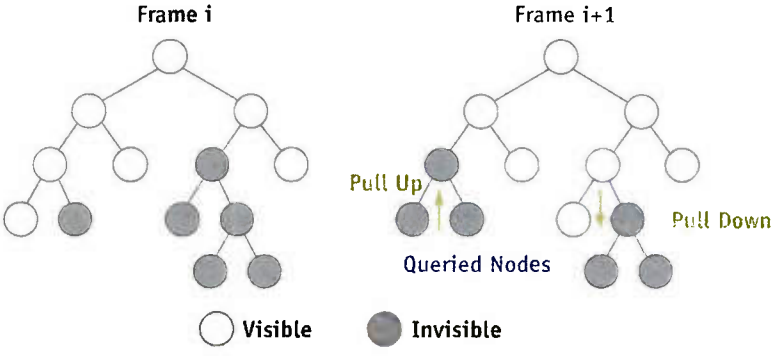


Figure 6-4. Visibility of Hierarchy Nodes in Two Consecutive Frames

6.5.4 Implementation Details

The algorithm is easy to follow using the pseudocode, which we show in Listing 6-1. Let's discuss some of the details in the code.

To maintain the visibility classification for all nodes over consecutive frames, we use a `visible` flag and a `frameID`, which increments every frame. Nodes visible in the previous frame are easily identified through `lastVisited = frameID - 1` and `visible = true`; all other nodes are assumed invisible. This way, we don't have to reset the visibility classification for each frame.

NVIDIA Ex1023
Page Q-145

Listing 6-1. Algorithm Pseudocode

```
TraversalStack.Push(hierarchy.Root);
while ( not TraversalStack.Empty() or
        not QueryQueue.Empty() )
{
  //- PART 1: process finished occlusion queries
  while ( not QueryQueue.Empty() and
          (ResultAvailable(QueryQueue.Front()) or
           TraversalStack.Empty()) )
  {
    node = QueryQueue.Dequeue();
    // wait if result not available
    visiblePixels = GetOcclusionQueryResult(node);
    if ( visiblePixels > VisibilityThreshold )
    {
      PullUpVisibility(node);
      TraverseNode(node);
    }
  }

  //- PART 2: hierarchical traversal
  if ( not TraversalStack.Empty() )
  {
    node = TraversalStack.Pop();
    if ( InsideViewFrustum(node) )
    {
      // identify previously visible nodes
      wasVisible = node.visible and
                   (node.lastVisited == frameID - 1);
      // identify nodes that we cannot skip queries for
      leafOrWasInvisible = not wasVisible or IsLeaf(node);
      // reset node's visibility classification
      node.visible = false;
      // update node's visited flag
      node.lastVisited = frameID;
      // skip testing previously visible interior nodes
      if ( leafOrWasInvisible )
      {
        IssueOcclusionQuery(node);
        QueryQueue.Enqueue(node);
      }
```


NVIDIA Ex1023
Page Q-146

Listing 6-1 *(continued)*. Algorithm Pseudocode

```
        // always traverse a node if it was visible
        if ( wasVisible )
          TraverseNode(node);
      }
    }
  }

  TraverseNode(node)
  {
    if ( IsLeaf(node) )
      Render(node);
    else
      TraversalStack.PushChildren(node);
  }

  PullUpVisibility(node)
  {
    while (not node.visible)
    {
      node.visible = true;
      node = node.parent;
    }
  }
```

To establish the new visibility classification for the current frame, we set all nodes we encounter during the traversal to invisible by default. Later, when a query identifies a node as visible, we "pull up" visibility; that is, we set all its ancestors to visible as well.

Interior nodes that were visible in the previous frame (that is, not `leafOrWasInvisible`) are nodes for which we skip occlusion queries altogether.

Traversing a node means rendering the node for leaf nodes or pushing its children onto the traversal stack so that they get rendered front to back.

We also assume that the `Render()` call renders an object only if it hasn't been rendered before (this can be checked using the `frameID`). This facilitates the code for previously visible objects: when we finally get the result of their visibility query, we can just process them again without introducing a special case or rendering them twice (since they have already been rendered immediately after their query was issued).

NVIDIA Ex1023
Page Q-147

Another advantage is that we can reference objects in several nodes of the hierarchy, and they still get rendered only once if several of these nodes are visible.

Furthermore, we have to be careful with occlusion tests for nodes near the viewpoint. If the front faces of a bounding box get clipped by the near plane, no fragments are generated for the bounding box. Thus the node would be classified as occluded by the test, even though the node is most likely visible. To avoid this error for each bounding box that passes the view-frustum test, we should check whether any of its vertices is closer than the near plane. In such a case, the associated node should be set visible without issuing an occlusion query. Finally, note that for the occlusion queries, the actual node bounding boxes can be used instead of the (usually less tight) nodes of the spatial hierarchy.

6.5.5 Why Are There Fewer Stalls?

Let's take a step back and see what we have achieved with our algorithm and why.

The coherent hierarchical culling algorithm does away with most of the inefficient waiting found in the hierarchical stop-and-wait method. It does so by interleaving the occlusion queries with normal rendering, and avoiding the need to wait for a query to finish in most cases.

If the viewpoint does not move, then the only point where we actually might have to wait for a query result is at the end of the frame, when all the visible geometry is already rendered. Previously visible nodes are rendered right away without waiting for the results.

If the viewpoint does move, the only dependency occurs if a previously invisible node becomes visible. We obviously need to wait for this to be known in order to decide whether to traverse the children of the node. However, this does not bother us too much: most likely, we have some other work to do during the traversal. The query queue allows us to check regularly whether the result is available while we are doing useful work in between.

Note that in this situation, we might also not detect some occlusion situations and unnecessarily draw some objects. For example, if child A occludes child B of a previously occluded interior node, but the query for B is issued before A is rendered, then B is still classified as visible for the current frame. This happens when there is not enough work to do between issuing the queries for A and B. (See also Section 6.5.7, where we show how a priority queue can be used to increase the chance that there is work to do between the queries for A and B.)

NVIDIA Ex1023
Page Q-148

6.5.6 Why Are There Fewer Queries?

We have already seen that a hierarchical occlusion-culling algorithm can save a lot of occlusion queries if large parts of the scene are occluded. However, this is paid for by large costs for testing interior nodes.

The coherent hierarchical culling algorithm goes a step further by obviating the need for testing most interior nodes. The number of queries that need to be issued is only proportional to the number of visible objects, not to the total number of objects in the scene. In essence, the algorithm always tests the largest possible nodes for occluded regions.

Neither does the number of queries depend on the depth of the hierarchy, as in the hierarchical stop-and-wait method. This is a huge win, because the rasterization of large interior nodes can cost more than occlusion culling buys.

6.5.7 How to Traverse the Hierarchy

We haven't talked a lot about traversing the hierarchy up to now. In principle, the traversal depends on which hierarchy we use, and is usually implemented with a traversal stack, where we push child nodes in front-to-back order when a node is traversed. This basically boils down to a depth-first traversal of the hierarchy.

However, we can gain something by not adhering to a strict depth-first approach. When we find nodes that have become visible, we can insert these nodes into the traversal, which should be compatible with the already established front-to-back order in the hierarchy.

The solution is not to use a traversal stack, but a priority queue based on the distance to the observer. A priority queue makes the front-to-back traversal very simple. Whenever the children of a node should be traversed, they are simply inserted into the priority queue. The main loop of the algorithm now just checks the priority queue, instead of the stack, for the next node to be processed.

This approach makes it simple to work with arbitrary hierarchies. The hierarchy only needs to provide a method to extract its children from a node, and to compute the bounding box for any node. This way, it is easy to use a sophisticated hierarchy such as a k-d tree (Cohen-Or et al. 2003), or just the bounding volume hierarchy of the scene graph for the traversal, and it is easy to handle dynamic scenes.

Note also that an occlusion-culling algorithm can be only as accurate as the objects on which it operates. We cannot expect to get accurate results for individual triangles if we

NVIDIA Ex1023
Page Q-149

only test nodes of a coarse hierarchy for occlusion. Therefore, the choice of a hierarchy plays a critical role in occlusion culling.

6.6 Optimizations

We briefly cover a few optimizations that are beneficial in some (but not all) scenes.

6.6.1 Querying with Actual Geometry

First of all, a very simple optimization that is always useful concerns previously visible leaf nodes (Sekulic 2004). Because these will be rendered regardless of the outcome of the query, it doesn't make sense to use an approximation (that is, a bounding box) for the occlusion query. Instead, when issuing the query as described in Section 6.3, we omit step 2 and replace step 3 by rendering the actual geometry of the object. This saves rasterization costs and draw calls as well as state changes for previously visible leaf nodes.

6.6.2 Z-Only Rendering Pass

For some scenes, using a z-only rendering pass can be advantageous for our algorithm. Although this entails twice the transformation cost and (up to) twice the draw calls for visible geometry, it provides a good separation between occlusion culling and rendering passes as far as rendering states are concerned. For the occlusion pass, the only state that needs to be changed between an occlusion query and the rendering of an actual object is depth writing. For the full-color pass, visibility information is already available, which means that the rendering engine can use any existing mechanism for optimizing state change overhead (such as ordering objects by rendering state).

6.6.3 Approximate Visibility

We might be willing to accept certain image errors in exchange for better performance. This can be achieved by setting the `VisibilityThreshold` in the algorithm to a value greater than zero. This means that nodes where no more than, say, 10 or 20 pixels are visible are still considered occluded. Don't set this too high, though; otherwise the algorithm culls potential occluders and obtains the reverse effect.

This optimization works best for scenes with complex visible geometry, where each additional culled object means a big savings.

NVIDIA Ex1023
Page Q-150

6.6.4 Conservative Visibility Testing

Another optimization makes even more use of temporal coherence. When an object is visible, the current algorithm assumes it will also be visible for the next frame. We can even go a step further and assume that it will be visible for a number of frames. If we do that, we save the occlusion queries for these frames (we assume it's visible anyway). For example, if we assume an object remains visible for three frames, we can cut the number of required occlusion queries by almost a factor of three! Note, however, that temporal coherence does not always hold, and we almost certainly render more objects than necessary.

This optimization works best for deep hierarchies with simple leaf geometry, where the number of occlusion queries is significant and represents significant overhead that can be reduced using this optimization.

6.7 Conclusion

Occlusion culling is an essential part of any rendering system that strives to display complex scenes. Although hardware occlusion queries seem to be the long-sought-after solution to occlusion culling, they need to be used carefully, because they can introduce stalls into the rendering pipeline.

We have shown an algorithm that practically eliminates any waiting time for occlusion query results on both the CPU and the GPU. This is achieved by exploiting temporal coherence, assuming that objects that have been visible in the previous frame remain visible in the current frame. The algorithm also reduces the number of costly occlusion queries by using a hierarchy to cull large occluded regions using a single test. At the same time, occlusion tests for most other interior nodes are avoided.

This algorithm should make hardware occlusion queries useful for any application that features a good amount of occlusion, and where accurate images are important. For example, Figure 6-5 shows the application of the algorithm to the walkthrough of a city model, with the visibility classification of hierarchy nodes below. The orange nodes were found visible; all the other depicted nodes are invisible. Note the increasing size of the occluded nodes with increasing distance from the visible set. For the viewpoint shown, the algorithm presented in this chapter provided a speedup of about 4 compared to rendering with view-frustum culling alone, and a speedup of 2.6 compared to the hierarchical stop-and-wait method.

NVIDIA Ex1023
Page Q-151

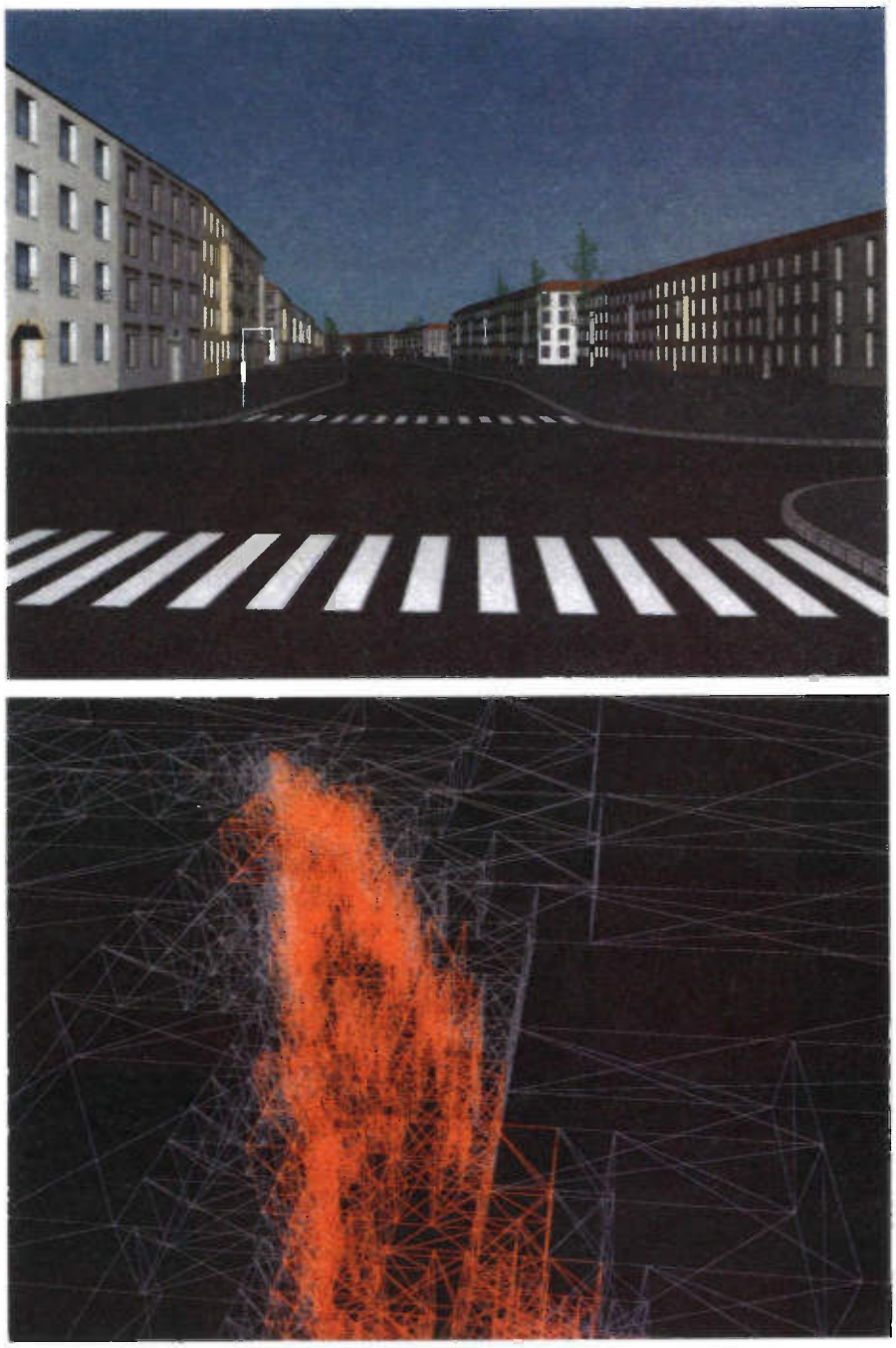


Figure 6-5. Visualizing the Benefits of Occlusion Queries for a City Walkthrough

NVIDIA Ex1023
Page Q-152

6.8 References

Bittner, J., M. Wimmer, H. Piringer, and W. Purgathofer. 2004. "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful." *Computer Graphics Forum (Proceedings of Eurographics 2004)* 23(3), pp. 615–624.

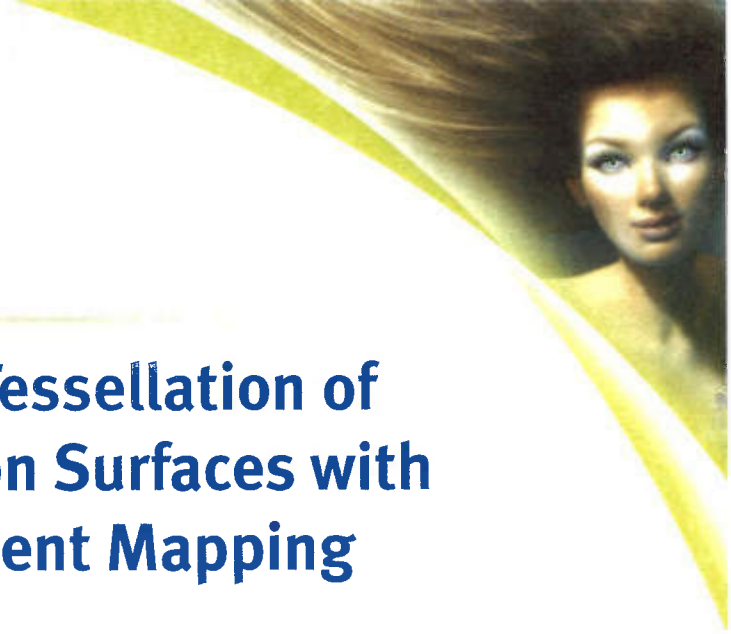
Cohen-Or, D., Y. Chrysanthou, C. Silva, and F. Durand. 2003. "A Survey of Visibility for Walkthrough Applications." *IEEE Transactions on Visualization and Computer Graphics* 9(3), pp. 412–431.

NVIDIA Corporation. 2004. *NVIDIA GPU Programming Guide.* **http://developer.nvidia.com/object/gpu_programming_guide.html**

Sekulic, D. 2004. "Efficient Occlusion Culling." In *GPU Gems*, edited by Randima Fernando, pp. 487–503. Addison-Wesley.

Wloka, M. 2003. "Batch, Batch, Batch: What Does It Really Mean?" Presentation at Game Developers Conference 2003. **http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf**

NVIDIA Ex1023
Page Q-153



Chapter 7

Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping

Michael Bunnell
NVIDIA Corporation

In this chapter we describe how to perform view-dependent, adaptive tessellation of Catmull-Clark subdivision surfaces with optional displacement mapping. We use the GPU to do the tessellation calculations, which saves graphics bus bandwidth and is many times faster than using the CPU. The first part of this chapter explains how to tessellate subdivision surfaces to polygons for rendering high-quality curved surfaces without visible polygon artifacts. The second part of this chapter describes how to add displacement-mapping support for rendering highly detailed models that can be animated in real time.

This chapter takes a repeated subdivision approach to tessellation, implemented by rendering into 2D textures. The subdivision, flatness test, and final vertex attribute calculations are done using fragment programs (pixel shaders). This method assumes that the vertex data of the subdivision surface control mesh are stored in a texture map. Intermediate results are also rendered to and read from texture maps, and the final tessellation results (position, normal, and so on) are rendered to a vertex array ready to be used by a render-primitives call such as `glDrawElements()`.

7.1 Subdivision Surfaces

Subdivision surfaces are arguably the most popular curved-surface representation used in computer graphics today. Specifically, Catmull-Clark subdivision surfaces are supported in practically every 3D modeling and animation application. Introduced to the

NVIDIA Ex1023
Page Q-154

big screen by Pixar with *A Bug's Life*, they have been used in all of Pixar's movies since, including *Finding Nemo* and *The Incredibles*. Subdivision surfaces are often combined with displacement mapping to add extra detail for computer-generated characters in live-action movies. The most striking example of a displacement-mapped subdivision surface model is probably the creature Gollum from the recent *Lord of the Rings* movies.

Subdivision surfaces are curved surfaces (or higher-order surfaces) described by a simple polygon control mesh and some subdivision rules. The subdivision rules define how to perform one subdivision step on the surface. Each subdivision step creates a new mesh with more geometry and a smoother appearance than the previous mesh. Figure 7-1 illustrates such rules applied to a cube.

7.1.1 Some Definitions

- **Limit surface**—the hypothetical surface created after an infinite number of subdivision steps
- **Valence (of a vertex)**—the number of edges connected to a vertex
- **Extraordinary point**—a vertex with a valence other than 4 (for a quad-based subdivision scheme, such as Catmull-Clark)

7.1.2 Catmull-Clark Subdivision

We use Catmull-Clark subdivision surfaces in this chapter, and so we use the Catmull-Clark subdivision rules (Catmull and Clark 1978). Catmull-Clark subdivision works on meshes of arbitrary manifold topology. However, it is considered a quad-based subdivision scheme because it works best on four-sided polygons (quads) and because all

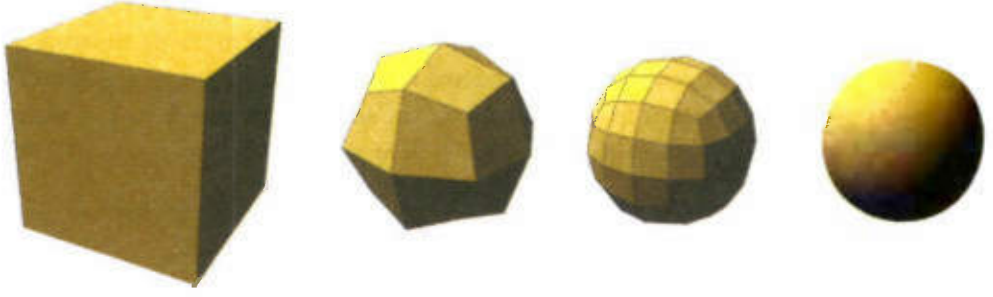


Figure 7-1. Catmull-Clark Subdivision of a Cube
From left to right: Original control mesh, mesh after one subdivision step, mesh after two steps, and the limit surface.

NVIDIA Ex1023
Page Q-155

the polygons in the mesh after the first subdivision step are quads. Our implementation is limited to control meshes consisting only of quads. This implies that the source models have to be created as all-quad models, a task familiar to artists with a NURBS modeling background or experience using Lightwave SubPatches.

The Catmull-Clark rules to subdivide an all-quad mesh are simple. Split each face of the control mesh into four faces by adding a vertex in the middle of the face and a vertex at each edge. The positions of these new vertices and the new positions of the original vertices can be computed from the pre-subdivided mesh vertex positions using the weight masks in Figure 7-2. We show only the weight masks for calculating the new positions of original vertices of valence 3, 4, and 5. The weight masks for higher valences follow the same progression.

If we examine a couple of subdivision steps of a single face, we notice that subdivision is very similar to scaling a 2D image. The only differences are the filtering values used and that the vertex position data is generally stored as floating point, while most images are stored as integer or fixed point. Figure 7-3 shows a single face of a mesh being subdivided two times. One thing to note about this figure is that we need a ring of vertices around the faces that are being subdivided to perform the subdivision. Also, the extraordinary point (the vertex of valence 5) at the upper-left corner of the face in the original mesh is present in the subdivided meshes too.

7.1.3 Using Subdivision for Tessellation

One obvious way to tessellate subdivision surfaces is to apply a fixed number of subdivision steps on the original control mesh to create a list of quads to render. In fact, that is how pre-tessellated surfaces are created with most 3D-modeling packages. The disadvantage of this method is that parts of a model are over-tessellated and other parts under-tessellated. Also, it leaves the burden of selecting the number of subdivisions on the application program. Under-tessellation leads to visible polygon artifacts. Over-tessellation creates too

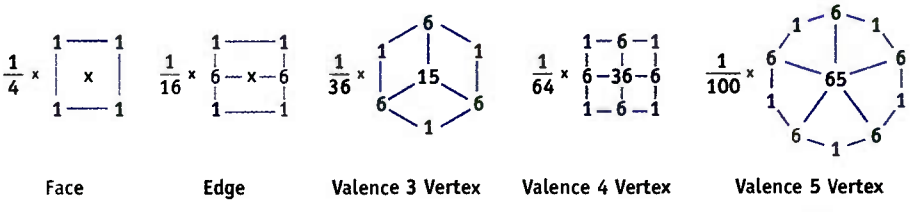


Figure 7-2. Catmull-Clark Subdivision Masks
Used for calculating new vertex positions from pre-subdivided mesh vertex positions.

NVIDIA Ex1023
Page Q-156

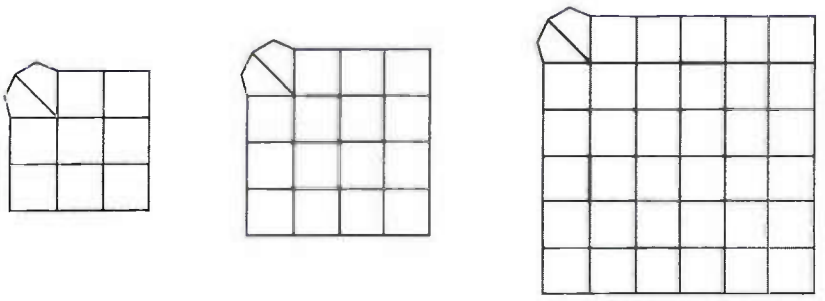


Figure 7-3. Subdividing a Face of a Subdivision Surface

many small triangles that can severely affect rendering performance. Too much geometry causes the GPU to become vertex-transform or triangle-setup limited. Worse, having many tiny triangles that are just a few pixels in size can significantly reduce rasterizer efficiency.

Adaptive Subdivision

We use a flatness-test-based adaptive subdivision algorithm to avoid the problems with the fixed subdivision method. Instead of blindly subdividing a fixed number of steps, we test the surface for flatness first. In this way, the subdivision adapts to the surface curvature and the viewpoint. The closer the surface is or the more curved it is, the more it gets subdivided. See Figure 7-4. We measure flatness as the distance from the polygonal mesh to the limit surface measured in screen pixels. For example, a flatness value of 0.5 means the maximum distance from the polygon approximation of the surface to the limit surface is half a pixel.

7.1.4 Patching the Surface

We need to store the subdivision control mesh vertex data in a format that allows for efficient processing. We do this by breaking up the original control mesh into smaller meshes that can be represented as regular 2D grids and therefore stored in 2D arrays. See Figure 7-5. Now we can infer the mesh connectivity by the location of vertices in the array. Vertices next to each other are assumed to be connected by edges. Control points arranged this way, in a 2D array or matrix, are generally referred to as *patches*, so that is what we call them. Our patches are different in that we allow extraordinary points at the patches' corners.

The obvious way to divide the control mesh into patches is to create a patch for each face of the mesh. This method requires a lot of duplicated vertices because we need to include the vertices from neighboring faces so we can subdivide the patch. We can improve on this situation by creating patches with more faces when possible.

NVIDIA Ex1023
Page Q-157

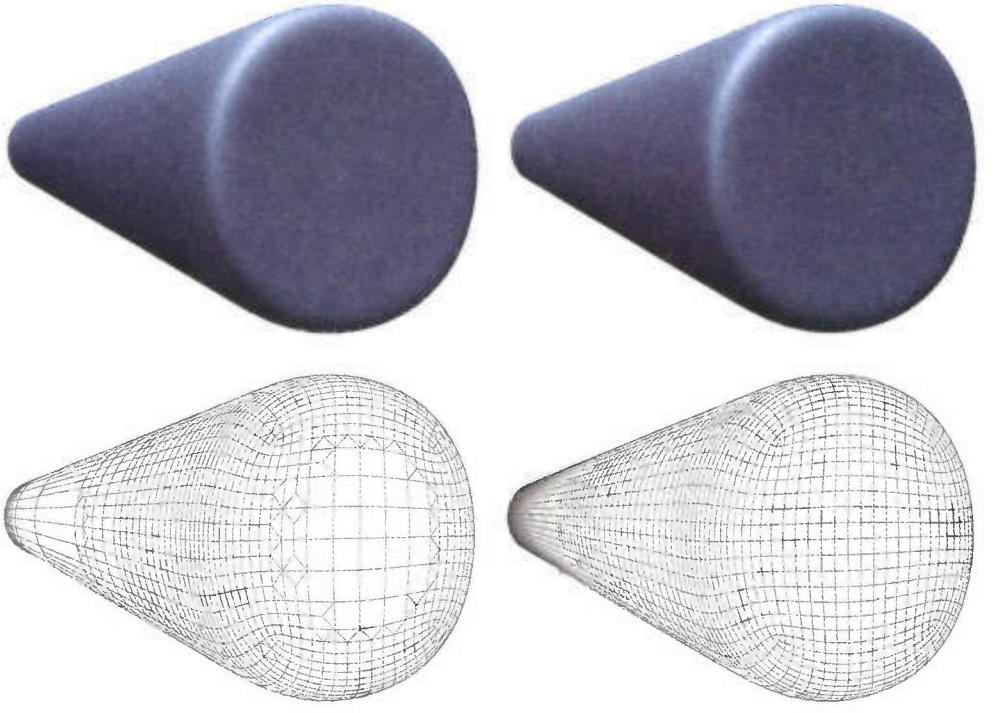


Figure 7-4. Adaptive vs. Uniform Tessellation
The long, cylindrical object on the left is adaptively tessellated and consists of about one-third as many polygons as the uniformly tessellated one on the right.

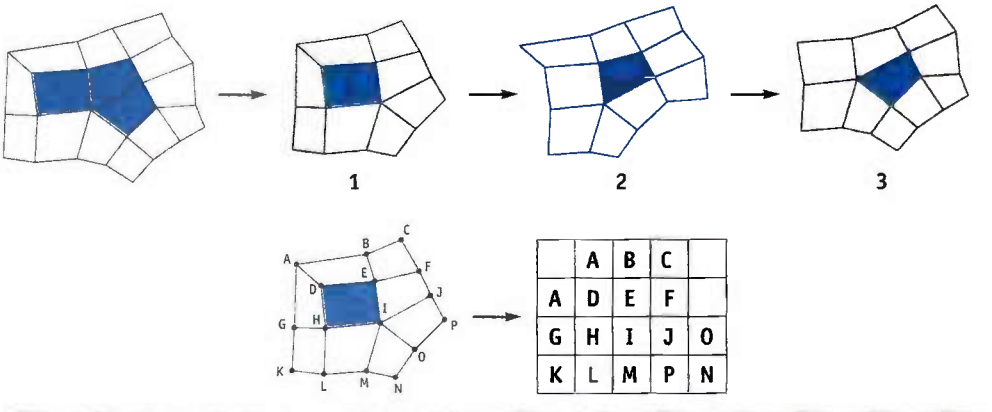


Figure 7-5. Patching a Surface
Top: A mesh is broken up into meshes 1, 2, and 3. Bottom: Mesh 1 is converted to a patch.

NVIDIA Ex1023
Page Q-158

Patching a surface is not a trivial process, but fortunately it can be done in a preprocessing step. Even animated meshes generally do not change topology, so the vertex locations in each patch can be precalculated and stored in a 2D texture. The texture can be passed to the tessellation code along with the list of patch information, which includes the size of the patch and the valences of any extraordinary vertices. The demo program available on the book's Web site shows one possible implementation of patching an arbitrary quad model.

7.1.5 The GPU Tessellation Algorithm

The tessellation algorithm works as follows. First, we copy the vertex data of the control mesh (positions) into the patch buffer texture. The flatness test assumes the vertex coordinates are in eye space. If they are not in eye space, they can be transformed before or during this copy. Next, we check each patch for flatness (see "The Flatness Test," later in this section). Then we compute vertex attribute data, such as positions and normals, for the patches that are flat and write them to the vertex array, writing the index information to a primitive index buffer. Patches that are not flat are subdivided into another patch buffer. Then we swap patch-buffer pointers and go back to the flatness test step. This section describes each step in the tessellation process in more detail.

CPU Processing

We use the CPU to do buffer management and to create the list of vertex indices that form part of the tessellation result. The CPU does not have to touch the vertex data, but it does read the flatness test results (using `glReadPixels`) and decides which patches need further tessellation. It assigns patch locations in the buffers, and it uses a simple recursive procedure to build the primitive index list based on the edge flatness results computed by the shaders.

Creating the Patches

Patches are created by rendering a quad per patch at the appropriate location in the patch buffer, to copy the control mesh vertex position data from a texture. The program executed at the quad's fragments uses a precomputed texture to index the control mesh vertices discussed in Section 7.1.4. If there are more vertex attributes to subdivide, such as texture coordinates, they are copied into a different location in the patch buffer. Vertex attributes other than position are bilinearly interpolated for performance.

The Flatness Test

The flatness of each edge in the patch is calculated by rendering a quad of appropriate size into the flatness buffer using the FlatTest1.cg or the FlatTest2.cg shaders. (All of

NVIDIA Ex1023
Page Q-159

the shaders mentioned here can be found with the SubdViewer demo program.) Unlike the other buffers used for tessellation, this buffer is not floating point. It is an 8-bit-per-component buffer.

The two flat-test shaders measure the flatness at each edge of the control mesh. Both shaders treat the four control points along an edge as a b-spline curve and measure the flatness of the curve. One shader is used before the first subdivision step because it can handle s-type curves, which are curves whose first derivatives change sign. The second, faster test is used thereafter, because there are only c-type curves after the first subdivision. (C-type curves have first derivatives that do not change sign and thus are c-shaped.)

As shown in Figure 7-6, the first shader's flatness test calculates the distance from the control point 2 to the midpoint of control points 1 and 3, as well as the distance of control point 3 to the midpoint of control points 2 and 4. The second flatness test calculates the distance from the midpoint of control points 1 and 4 to the midpoint of control points 2 and 3. Both tests compare the calculated distance to the flatness value given in pixels, scaled appropriately. The flatness value is prescaled to account for the distance of the b-spline control points from the limit curve, and it is doubled so the midpoint calculations can be done with only an add. Because the control points are assumed to be in eye space, the flatness distance is scaled by the z value of the midpoint of control points 2 and 3 to handle perspective projection.

The flatness test also needs to handle z-scale values that are less than the near render plane to avoid over-tessellating meshes that cross the near plane. It does so by reflecting z values less than the near plane to the other side of the plane. This technique results in good tessellation for patches that cross the near plane without creating too much geometry that would be clipped later.

Results of the flatness test are copied to host memory, so that the CPU can decide which patches are flat. The results are also saved for each level of subdivision except the last (we know all edges are flat after the last subdivision), so they can be used to build

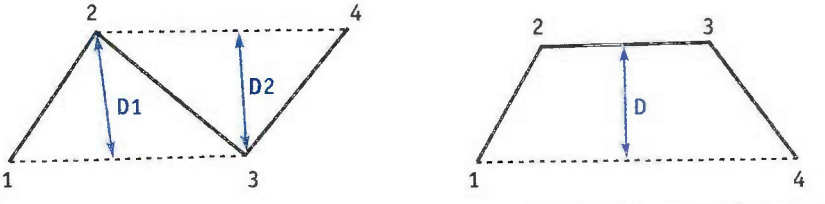


Figure 7-6. Comparing Flatness Tests
Left: Flatness test 1 measures two distances. Right: Flatness test 2 measures only one.

NVIDIA Ex1023
Page Q-160

the array of primitive indices. This data is packed to save bandwidth and averages about 200 times fewer bytes of data than the description of the patch itself.

Subdivision

Subdivision is performed by rendering a quad for each patch into the destination patch buffer using the Subdiv.cg shader shown in Listing 7-1. The render works similar to a 2× scaled image blit, but it uses the weights for Catmull-Clark subdivision, as discussed in Section 7.1.2. The source coordinate for rendering is offset from the patch's origin by (0.75, 0.75), to account for the ring of extra vertices around the faces that are being subdivided and to create a similar ring in the destination. See Figure 7-7. The size of the result patch is twice that of the source patch, minus 3. The Subdiv.cg shader uses the fractional part of the source coordinates to determine which subdivision mask to use: the face mask, the edge mask, or the valence-4 vertex mask. Attributes other than position are interpolated bilinearly in Normal.cg.

Listing 7-1. Subdiv.cg—Subdividing a Regular Mesh Using Catmull-Clark Rules

```
float4 main(float4 srcCoord,
            uniform samplerRECT srcTexMap : TEXUNIT0) : COLOR
{
  float3 wh, wv;      // weight values
  float3 s1, s2, s3;
  float2 f;

  // calculate weights based on fractional part of
  // source texture coordinate
  //
  // x == 0.0, y == 0.0 use face weights (1/4, 1/4, 0)
  //                                     (1/4, 1.4, 0)
  //                                     (  0,   0, 0)
  // x == 0.5, y == 0.0 use edge weights (horizontal)
  //                                     (1/16, 6/16, 1/16)
  //                                     (1/16, 6/16, 1/16)
  //                                     (   0,    0,    0)
  // x == 0.0, y == 0.5 use edge weights (vertical)
  //                                     (1/16, 1/16, 0)
  //                                     (6/16, 6/16, 0)
  //                                     (1/16, 1/16, 0)
  // x == 0.5, y == 0.5 use valence 4 vertex weights
  //                                     (1/64,  6/64, 1/64)
  //                                     (6/64, 36/64, 6/64)
  //                                     (1/64,  6/64, 1/64)
```


NVIDIA Ex1023
Page Q-161

Listing 7-1 *(continued)*. Subdiv.cg—Subdividing a Regular Mesh Using Catmull-Clark Rules

```
    wh = float3(1.0/8.0, 6.0/8.0, 1.0/8.0);
    f = frac(srcCoord.xy + 0.001) < 0.25; // account for finite precision
    if (f.x != 0.0) {
      wh = float3(0.5, 0.5, 0.0);
      srcCoord.x += 0.5; // fraction was zero -- move to texel center
    }
    wv = float3(1.0/8.0, 6.0/8.0, 1.0/8.0);
    if (f.y != 0) {
      wv = float3(0.5, 0.5, 0.0);
      srcCoord.y += 0.5;  // fraction was zero -- need to move to texel center
    }

    // calculate the destination vertex position by using the weighted
    // sum of the 9 vertex positions centered at srcCoord

    s1 = texRECT(srcTexMap, srcCoord.xy + float2(-1, -1)).xyz * wh.x +
         texRECT(srcTexMap, srcCoord.xy + float2(0, -1)).xyz * wh.y +
         texRECT(srcTexMap, srcCoord.xy + float2(1, -1)).xyz * wh.z;

    s2 = texRECT(srcTexMap, srcCoord.xy + float2(-1, 0)).xyz * wh.x +
         texRECT(srcTexMap, srcCoord.xy + float2(0, 0)).xyz * wh.y +
         texRECT(srcTexMap, srcCoord.xy + float2(1, 0)).xyz * wh.z;

    s3 = texRECT(srcTexMap, srcCoord.xy + float2(-1, 1)).xyz * wh.x +
         texRECT(srcTexMap, srcCoord.xy + float2(0, 1)).xyz * wh.y +
         texRECT(srcTexMap, srcCoord.xy + float2(1, 1)).xyz * wh.z;

    return float4(s1 * wv.x + s2 * wv.y + s3 * wv.z, 0);
}
```

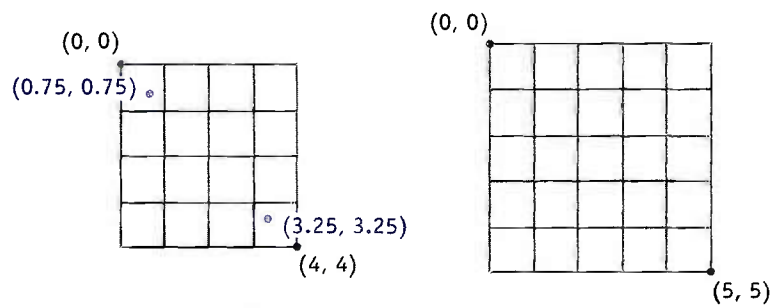


Figure 7-7. Resampling a Patch
The blue 2.5×2.5-pixel square portion of the patch (left) is sampled to create a 5×5-pixel patch (right).

NVIDIA Ex1023
Page Q-162

If there are any extraordinary points at the corners of the patch, they are copied and adjusted using the extraordinary-point subdivision shader, EPSubdiv.cg. This shader does not really subdivide per se, because no more data is created. It only adjusts the positions based on the subdivision rules, using a table lookup to get the appropriate filter weights. It calculates the coordinates of the weights in the lookup table based on the valence of the extraordinary point and the relative position of the control point that it is calculating. This shader is much slower than the regular subdivision shader, but it runs at many fewer pixels. Each corner of the patch (that has an extraordinary point) is handled by rendering a separate, small quad. The extraordinary-point shader is run after the regular subdivision shader and overwrites the control points in the destination that were calculated incorrectly.

Limit Positions and Normals

We use the limit shaders and normal shaders to create the vertex attributes that are written to the vertex array for rendering, including the attributes that are bilinearly interpolated, which are copied in the normal shader. The limit shader calculates the position on the limit surface corresponding to the vertices of the patch. It uses a weighted average of each vertex and its eight neighbors (Halstead et al. 1993). The control points themselves are often not close enough to the limit surface to be substituted for the limit surface points, especially for neighboring patches that are subdivided to different depths. The EPLimit.cg shader is used to calculate the limit position for extraordinary points.

The normal shader calculates two tangents for each control point and then uses their cross product to calculate the surface normal of the limit surface at that point. The EPNormal.cg shader does the same for extraordinary points. Both shaders calculate the tangents using a weighted average of the control vertex's neighbors. Because the normal shaders use the same control points as the limit shaders, it is a good idea to combine them in a single shader and use multiple render targets to write out the position and normal at the same time.

7.1.6 Watertight Tessellation

Watertight tessellation means that there are no gaps between polygons in the tessellation output. Even tiny gaps can lead to missing pixels when rendering the surface. One source of these gaps is the fact that floating-point operations performed in different orders do not always give the exact same result. Ordering shader calculations so that they are identical for neighboring patches costs a lot of performance. Therefore, we do

NVIDIA Ex1023
Page Q-163

not rely on positions or flatness data at the edge of a patch to match its neighbor's. The index list-building code ensures watertightness by using only one patch's data for shared corners and edges.

Another watertight tessellation problem that must be avoided is T-junctions. T-junctions often crop up with adaptive tessellation schemes. They occur if a patch is split even though one or more of its edges is flat. If the patch sharing the flat edges is not also split, then a T-junction is created. There are different methods for avoiding T-junctions during tessellation, some of which are quite complicated. Our tessellator has the advantage that index values of all the vertices of a patch are known when building the index list. The tessellator simply replaces the index of the vertex at the junction with one that is farther from the edge and fills the gap with a triangle, as illustrated in Figure 7-8. Moving the vertex at the T-junction avoids zero-area triangles, which can cause artifacts when rendering alpha-blended surfaces.

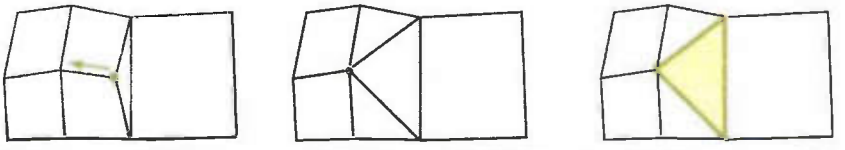


Figure 7-8. Handling T-Junctions
Left: A T-junction causes a crack in the mesh. Middle: The vertex at the T-junction is replaced with one away from the edge. Right: The crack is filled with a triangle.

7.2 Displacement Mapping

Displacement mapping is a method for adding geometric complexity to a simple, lightweight model. A displacement map is a texture map containing a scalar value per texel that describes how much the surface to which it is applied should be displaced in the direction of its normal. Unlike bump or normal maps, displacement maps change the actual geometry of the surface, not just the surface normal.

There are several reasons to add displacement mapping to subdivision surfaces. First, subdivision surfaces provide a nice, smooth surface from which to displace, even when animated. Second, they are very compact. The control mesh does not need any normal and tangent data, like polygon models, because they are derived during tessellation. Third, it is easy to modify our subdivision surface shaders to support adaptive displacement mapping.

NVIDIA Ex1023
Page Q-164

7.2.1 Changing the Flatness Test

The trick to efficient displacement mapping is adding geometry where it is really needed. We can modify our flatness test so that more geometry is added if the maximum displacement is greater than the flatness threshold. Because the flatness is tested per edge, we create a table using the displacement map containing the maximum displacement on either side of each edge. Figure 7-9 shows a 2D representation of how the maximum distance is calculated for each level of subdivision. Note that all the low-resolution mesh points are located on the displaced surface, because that is where the tessellated vertices are positioned. In the flatness-test shaders, we add the edge's maximum displacement value to the error distance before comparing it to the flatness threshold. All that is left to do geometry-wise is to add the displacement from the displacement map to the vertex position in the limit shaders.

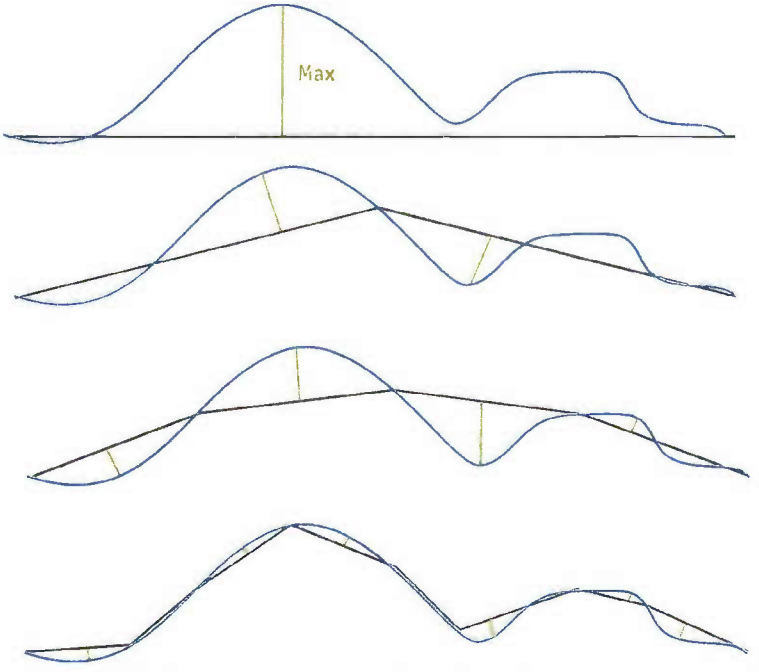


Figure 7-9. Four Steps of the Maximum Displacement Calculation

7.2.2 Shading Using Normal Mapping

We use normal mapping to take the displacement into account when shading the surface, instead of adjusting the vertex normals. See Figure 7-10 for an example. This technique has a couple of advantages. First, normal mapping is done per pixel, so it

NVIDIA Ex1023
Page Q-165

generally allows more detailed shading than is possible with vertex normals. More important, the surface does not appear to pop when animated, as the amount of geometry changes due to adaptive tessellation. Normal mapping requires some extra attributes from the tessellator: a tangent, and normal map texture coordinates.

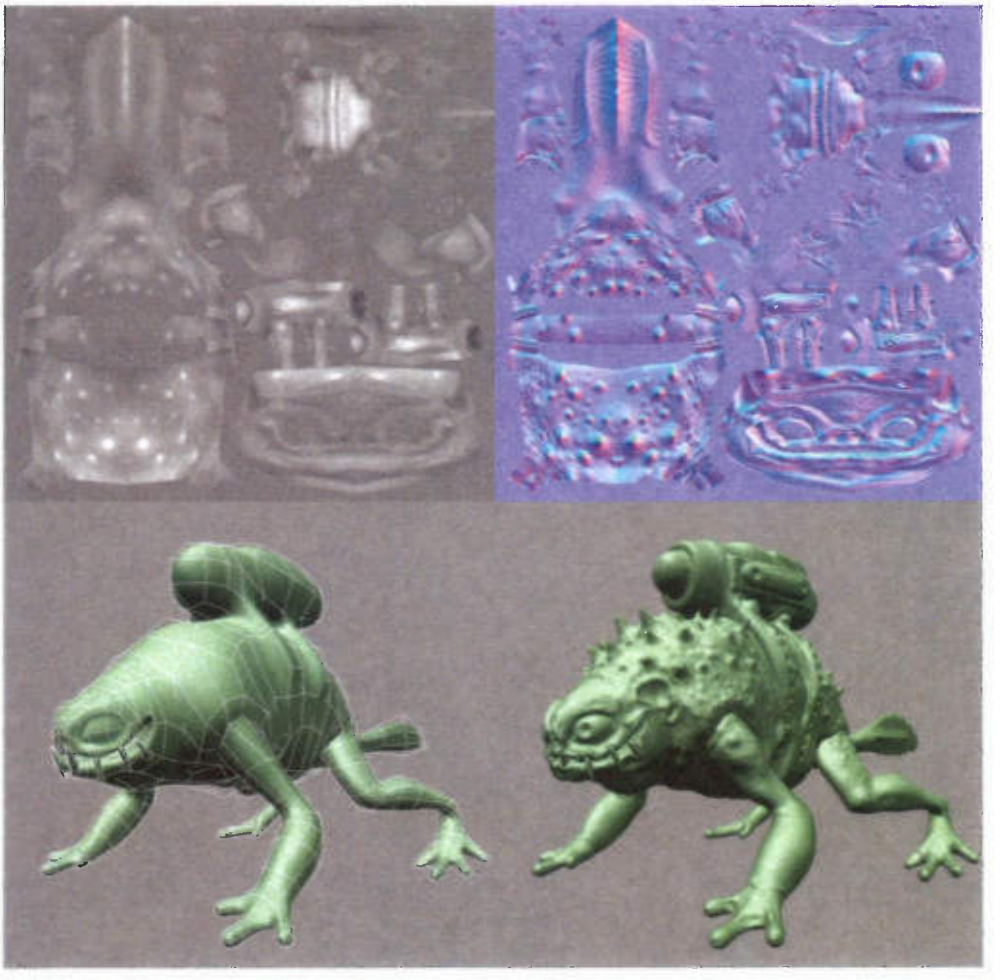


Figure 7-10. An Example of Displacement Mapping
The displacement and normal maps (top) were used to change the appearance of the low-resolution model from smooth (left) to detailed (right). Creature model courtesy of Bay Raitt, Boxrocket Entertainment.

NVIDIA Ex1023
Page Q-166

7.3 Conclusion

We have described a method to tessellate subdivision surfaces on the GPU using subdivision with a breadth-first recursion algorithm. We described the shaders necessary to perform flatness testing, to carry out the subdivision, and to compute the limit surface attributes. We also explained how to modify the shaders to add displacement-mapping support for increasing the geometric detail of subdivision surface models.

7.4 References

Catmull, E., and J. Clark. 1978. "Recursively Generated B-Spline Surfaces on Arbitrary Topology Meshes." *Computer Aided Design* 10(6), pp. 350–355.

DeRose, T., M. Kass, and T. Truong. 1998. "Subdivision Surfaces in Character Animation." In *Computer Graphics (Proceedings of SIGGRAPH 98)*, pp. 85–94.

Halstead, M., M. Kass, and T. DeRose. 1993. "Efficient, Fair Interpolation Using Catmull-Clark Surfaces." In *Computer Graphics (Proceedings of SIGGRAPH 93)*, pp. 35–44.

NVIDIA Ex1023
Page Q-167



Chapter 8

Per-Pixel Displacement Mapping with Distance Functions

William Donnelly
University of Waterloo

In this chapter, we present distance mapping, a technique for adding small-scale displacement mapping to objects in a pixel shader. We treat displacement mapping as a ray-tracing problem, beginning with texture coordinates on the base surface and calculating texture coordinates where the viewing ray intersects the displaced surface. For this purpose, we precompute a three-dimensional distance map, which gives a measure of the distance between points in space and the displaced surface. This distance map gives us all the information necessary to quickly intersect a ray with the surface. Our algorithm significantly increases the perceived geometric complexity of a scene while maintaining real-time performance.

8.1 Introduction

Cook (1984) introduced displacement mapping as a method for adding small-scale detail to surfaces. Unlike bump mapping, which affects only the shading of surfaces, displacement mapping adjusts the positions of surface elements. This leads to effects not possible with bump mapping, such as surface features that occlude each other and nonpolygonal silhouettes. Figure 8-1 shows a rendering of a stone surface in which occlusion between features contributes to the illusion of depth.

The usual implementation of displacement mapping iteratively tessellates a base surface, pushing vertices out along the normal of the base surface, and continuing until

NVIDIA Ex1023
Page Q-168

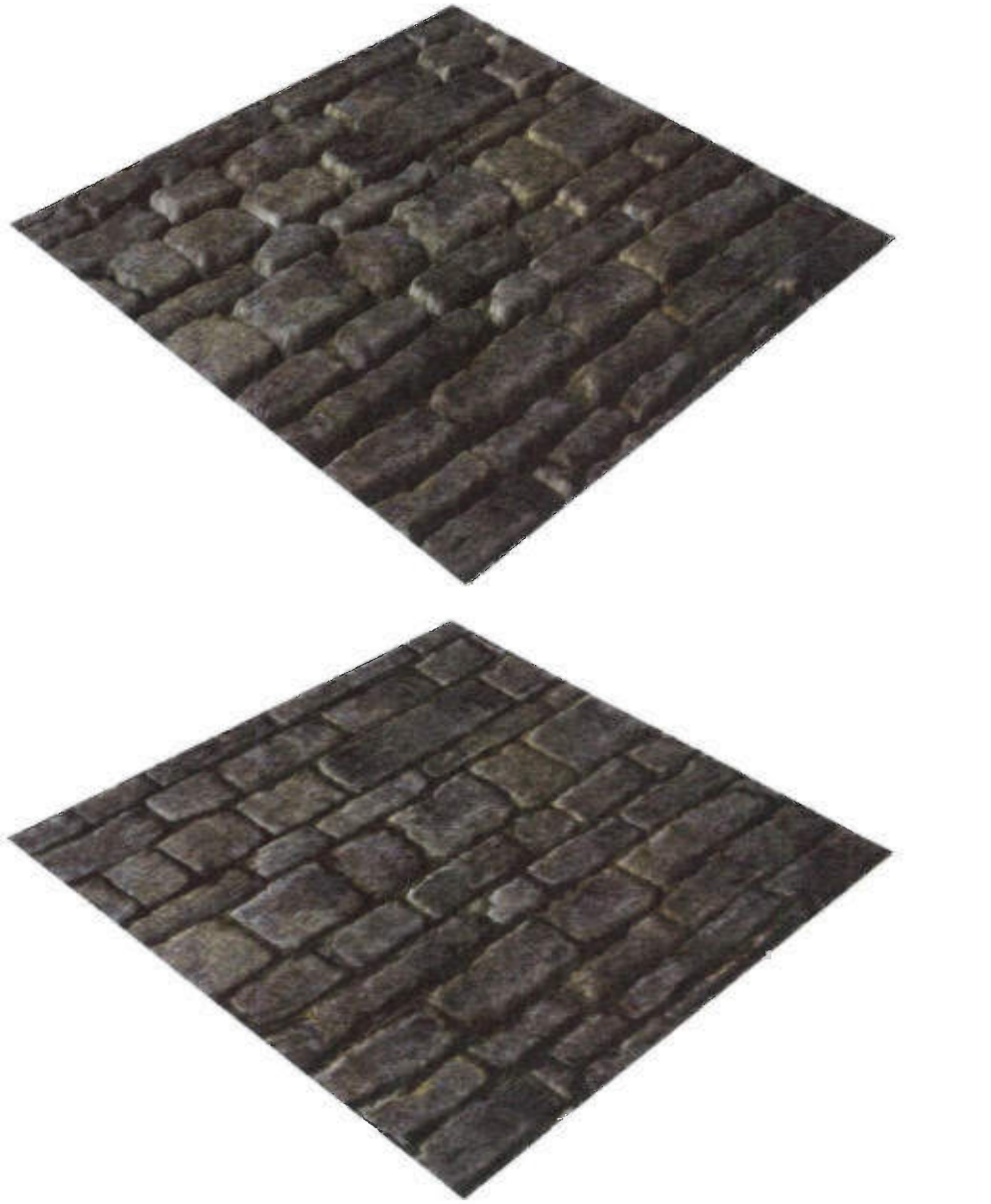


Figure 8-1. A Displaced Stone Surface
Displacement mapping (top) gives an illusion of depth not possible with bump mapping alone (bottom).

NVIDIA Ex1023
Page Q-169

the polygons generated are close to the size of a single pixel. Michael Bunnell presents this approach in Chapter 7 of this book, "Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping."

Although it seems most logical to implement displacement mapping in the vertex shading stage of the pipeline, this is not possible because of the vertex shader's inability to generate new vertices. This leads us to consider techniques that rely on the pixel shader. Using a pixel shader is a good idea for several reasons:

- **Current GPUs have more pixel-processing power than they have vertex-processing power.** For example, the GeForce 6800 Ultra has 16 pixel-shading pipelines to its 6 vertex-shading pipelines. In addition, a single pixel pipeline is often able to perform more operations per clock cycle than a single vertex pipeline.
- **Pixel shaders are better equipped to access textures.** Many GPUs do not allow texture access within a vertex shader; and for those that do, the access modes are limited, and they come with a performance cost.
- **The amount of pixel processing scales with distance.** The vertex shader always executes once for each vertex in the model, but the pixel shader executes only once per pixel on the screen. This means work is concentrated on nearby objects, where it is needed most.

A disadvantage of using the pixel shader is that we cannot alter a pixel's screen coordinate within the shader. This means that unlike an approach based on tessellation, an approach that uses the pixel shader cannot be used for arbitrarily large displacements. This is not a severe limitation, however, because displacements are almost always bounded in practice.

8.2 Previous Work

Parallax mapping is a simple way to augment bump mapping to include parallax effects (Kaneko et al. 2001). Parallax mapping uses the information about the height of a surface at a single point to offset the texture coordinates either toward or away from the viewer. Although this is a crude approximation valid only for smoothly varying height fields, it gives surprisingly good results, particularly given that it can be implemented with only three extra shader instructions. Unfortunately, because of the inherent assumptions of the technique, it cannot handle large displacements, high-frequency features, or occlusion between features.

NVIDIA Ex1023
Page Q-170

Relief mapping as presented by Policarpo (2004) uses a root-finding approach on a height map. It begins by transforming the viewing ray into texture space. It then performs a linear search to locate an intersection with the surface, followed by a binary search to find a precise intersection point. Unfortunately, linear search requires a fixed step size. This means that in order to resolve small details, it is necessary to increase the number of steps, forcing the user to trade accuracy for performance. Our algorithm does not have this trade-off: it automatically adjusts the step size as necessary to provide fine detail near the surface, while skipping large areas of empty space away from the surface.

View-dependent displacement mapping (Wang et al. 2003) and its extension, generalized displacement mapping (Wang et al. 2004), treat displacement mapping as a ray-tracing problem. They store the result of all possible ray intersection queries within a three-dimensional volume in a five-dimensional map indexed by three position coordinates and two angular coordinates. This map is then compressed by singular value decomposition, and decompressed in a pixel shader at runtime. Because the data is five-dimensional, it requires a significant amount of preprocessing and storage.

The algorithm for displacement rendering we present here is based on sphere tracing (Hart 1996), a technique developed to accelerate ray tracing of implicit surfaces. The concept is the same, but instead of applying the algorithm to ray tracing of implicit surfaces, we apply it to the rendering of displacement maps on the GPU.

8.3 The Distance-Mapping Algorithm

Suppose we want to apply a displacement map to a plane. We can think of the surface as being bounded by an axis-aligned box. The conventional displacement algorithm would render the bottom face of the box and push vertices upward. Our algorithm instead renders the top plane of the box. Then within the shader, we find which point on the displaced surface the viewer would have really seen.

In a sense, we are computing the inverse problem to conventional displacement mapping. Displacement mapping asks, "For this piece of geometry, what pixel in the image does it map to?" Our algorithm asks, "For this pixel in the image, what piece of geometry do we see?" While the first approach is the one used by rasterization algorithms, the second approach is the one used by ray-tracing algorithms. So we approach distance mapping as a ray-tracing problem.

A common ray-tracing approach is to sample the height map at uniformly spaced locations to test whether the viewing ray has intersected the surface. Unfortunately, we

NVIDIA Ex1023
Page Q-171

encounter the following problem: as long as our samples are spaced farther apart than a single texel, we cannot guarantee that we have not missed an intersection that lies between our samples. This problem is illustrated in Figure 8-2. These "overshoots" can cause gaps in the rendered geometry and can result in severe aliasing. Thus we have two options: either we accept artifacts from undersampling, or we must take a sample for each texel along the viewing ray. Figure 8-3 shows that our algorithm can render objects with fine detail without any aliasing or gaps in geometry.

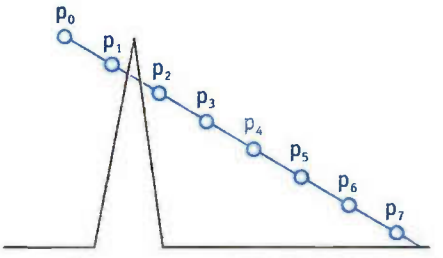


Figure 8-2. A Difficult Case for Displacement Algorithms
An algorithm that takes fixed-size steps misses an important piece of geometry.



Figure 8-3. Rendering Displaced Text
Because of the fine detail, text is a difficult case for some per-pixel displacement algorithms, particularly those based on uniformly spaced sampling. Note that our algorithm correctly captures occlusion between letters without aliasing.

NVIDIA Ex1023
Page Q-172

We can draw two conclusions from the preceding example:

1. We cannot simply query the height map at fixed intervals. It either leads to undersampling artifacts or results in an intractable algorithm.
2. We need more information at our disposal than just a height field; we need to know how far apart we can take our samples in any given region without overshooting the surface.

To solve these problems, we define the distance map of the surface. For any point p in texture space and a surface S, we define a function $\text{dist}(p, S) = \min\{d(p, q) : q \text{ in } S\}$. In other words, $\text{dist}(p, S)$ is the shortest distance from p to the closest point on the surface S. The distance map for S is then simply a 3D texture that stores, for each point p, the value of $\text{dist}(p, S)$. Figure 8-4 shows a sample one-dimensional height map and its corresponding distance map.

This distance map gives us exactly the information we need to choose our sampling positions. Suppose we have a ray with origin p_0 and direction vector $\mathbf{d}$, where $\mathbf{d}$ is normalized to unit length. We define a new point $p_1 = p_0 + \text{dist}(p_0, S) \times \mathbf{d}$. This point has the important property that if p_0 is outside the surface, then p_1 is also outside the surface. We then apply the same operation again by defining $p_2 = p_1 + \text{dist}(p_1, S) \times \mathbf{d}$, and so on. Each consecutive point is a little bit closer to the surface. Thus, if we take enough samples, our points converge toward the closest intersection of the ray with the surface. Figure 8-5 illustrates the effectiveness of this algorithm.

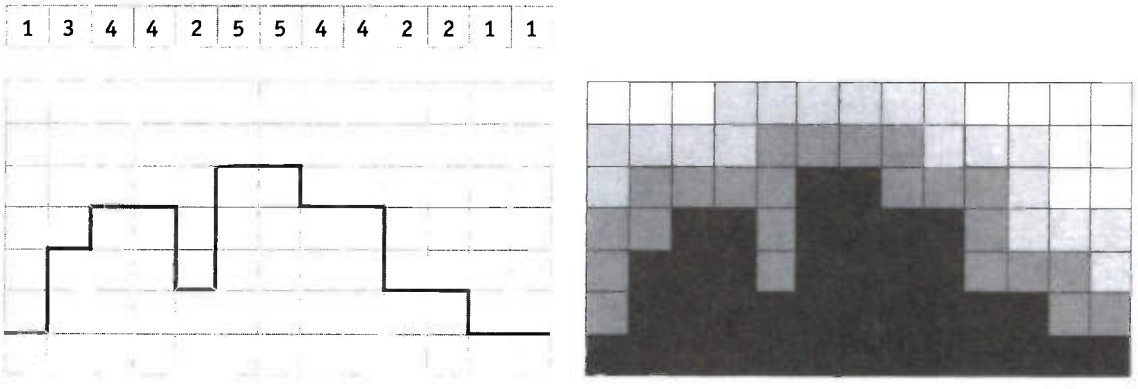


Figure 8-4. A Sample Distance Map in Two Dimensions
Left, top: A one-dimensional height map as an array. Left, bottom: The height map visualized in two dimensions. Right: The corresponding two-dimensional distance map.

NVIDIA Ex1023
Page Q-173

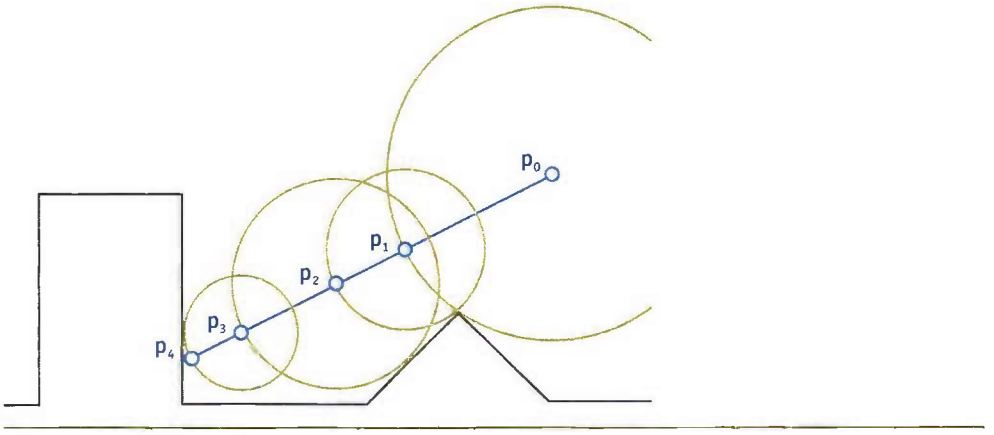


Figure 8-5. Sphere Tracing
A ray begins at point p_0. We then determine the distance to the closest point on the surface. Geometrically we can imagine expanding a sphere S_0 around p_0 until it intersects the surface. Point p_1 is then the intersection between the ray and the sphere. We repeat this process, generating points $p_2..p_4$. We can see from the diagram that point p_4 is effectively the intersection point with the surface, and so in this case the algorithm has converged in four iterations.

It is worth noting that distance functions do not apply only to height fields. In fact, a distance map can represent arbitrary voxelized data. This means that it would be possible to render small-scale detail with complex topology. For example, chain mail could be rendered in this manner.

8.3.1 Arbitrary Meshes

Up to this point, we have discussed applying distance mapping only to planes. We would like to apply distance mapping to general meshes. We do so by assuming the surface is locally planar. Based on this assumption, we can perform the same calculations as in the planar case, using the local tangent frame of the surface. We use the local surface tangents to transform the view vector into tangent space, just as we would transform the light vector for tangent space normal mapping. Once we have transformed the viewing ray into tangent space, the algorithm proceeds exactly as in the planar case.

Now that we know how to use a distance map for ray tracing, we need to know how to efficiently compute a distance map for an arbitrary height field.

NVIDIA Ex1023
Page Q-174

8.4 Computing the Distance Map

Computing a distance transform is a well-studied problem. Danielsson (1980) has presented an algorithm that runs in $O(n)$ time, where n is the number of pixels in the map. The idea is to create a 3D map where each pixel stores a 3D displacement vector to the nearest point on the surface. The algorithm then performs a small number of sequential sweeps over the 3D domain, updating each pixel's displacement vector based on neighboring displacement vectors. Once the displacements have been calculated, the distance for each pixel is computed as the magnitude of the displacement. An implementation of this algorithm is on the book's CD.

When the distance transform algorithm is finished, we have computed the distance from each pixel in the distance map to the closest point on the surface, measured in pixels. To make these distances lie in the range [0, 1], we divide each distance by the depth of the 3D texture in pixels.

8.5 The Shaders

8.5.1 The Vertex Shader

The vertex shader for distance mapping, shown in Listing 8-1, is remarkably similar to a vertex shader for tangent-space normal mapping, with two notable differences. The first is that in addition to transforming the light vector into tangent space, we also transform the viewing direction into tangent space. This tangent-space eye vector is used in the vertex shader as the direction of the ray to be traced in texture space. The second difference is that we incorporate an additional factor inversely proportional to the perceived depth. This allows us to adjust the scale of the displacements interactively.

8.5.2 The Fragment Shader

We now have all we need to begin our ray marching: we have a starting point (the base texture coordinate passed down by the vertex shader) and a direction (the tangent-space view vector).

The first thing the fragment shader has to do is to normalize the direction vector. Note that distances stored in the distance map are measured in units proportional to pixels; the direction vector is measured in units of texture coordinates. Generally, the distance map is much higher and wider than it is deep, and so these two measures of distance are quite different. To ensure that our vector is normalized with respect to the measure of distance used by the distance map, we first normalize the direction vector, and then

NVIDIA Ex1023
Page Q-175

Listing 8-1. The Vertex Shader

```
v2fConnector distanceVertex(a2vConnector a2v,
  uniform float4x4 modelViewProj,
  uniform float3 eyeCoord,
  uniform float3 lightCoord,
  uniform float invBumpDepth)
{
  v2fConnector v2f;

  // Project position into screen space
  // and pass through texture coordinate
  v2f.projCoord = mul(modelViewProj, float4(a2v.objCoord, 1));
  v2f.texCoord = float3(a2v.texCoord, 1);

  // Transform the eye vector into tangent space.
  // Adjust the slope in tangent space based on bump depth
  float3 eyeVec = eyeCoord - a2v.objCoord;
  float3 tanEyeVec;
  tanEyeVec.x = dot(a2v.objTangent, eyeVec);
  tanEyeVec.y = dot(a2v.objBinormal, eyeVec);
  tanEyeVec.z = -invBumpDepth * dot(a2v.objNormal, eyeVec);
  v2f.tanEyeVec = tanEyeVec;

  // Transform the light vector into tangent space.
  // We will use this later for tangent-space normal mapping
  float3 lightVec = lightCoord - a2v.objCoord;
  float3 tanLightVec;
  tanLightVec.x = dot(a2v.objTangent, lightVec);
  tanLightVec.y = dot(a2v.objBinormal, lightVec);
  tanLightVec.z = dot(a2v.objNormal, lightVec);
  v2f.tanLightVec = tanLightVec;

  return v2f;
}
```

multiply this normalized vector by an extra "normalization factor" of (*depth*/*width*, *depth*/*height*, 1).

We can now proceed to march our ray iteratively. We begin by querying the distance map, to obtain a conservative estimate of the distance we can march along the ray without intersecting the surface. We can then step our current position forward by that distance to obtain another point outside the surface. We can proceed in this way to

NVIDIA Ex1023
Page Q-176

generate a sequence of points that converges toward the displaced surface. Finally, once we have the texture coordinates of the intersection point, we compute normal-mapped lighting in tangent space. Listing 8-2 shows the fragment shader.

8.5.3 A Note on Filtering

When sampling textures, we must be careful about how to specify derivatives for texture lookups. In general, the displaced texture coordinates have discontinuities (for example, due to sections of the texture that are occluded). When mipmapping or anisotropic filtering is enabled, the GPU needs information about the derivatives of the texture coordinates. Because the GPU approximates derivatives with finite differences, these derivatives have incorrect values at discontinuities. This leads to an incorrect choice of mipmap levels, which in turn leads to visible seams around discontinuities.

Instead of using the derivatives of the displaced texture coordinates, we substitute the derivatives of the base texture coordinates. This works because displaced texture coordinates are always continuous, and they vary at approximately the same rate as the base texture coordinates.

Because we do not use the GPU's mechanism for determining mipmap levels, it is possible to have texture aliasing. In practice, this is not a big problem, because the derivatives of the base texture coordinates are a good approximation of the derivatives of the displaced texture coordinates.

Note that the same argument about mipmap levels also applies to lookups into the distance map. Because the texture coordinates can be discontinuous around feature edges, the texture unit will access a mipmap level that is too coarse. This in turn results in incorrect distance values. Our solution is to filter the distance map linearly without any mipmaps. Because the distance map values are never visualized directly, aliasing does not result from the lack of mipmapping here.

8.6 Results

We implemented the distance-mapping algorithm in OpenGL and Cg as well as in Sh (http://www.libsh.org). The images in this chapter were created with the Sh implementation, which is available on the accompanying CD. Because of the large number of dependent texture reads and the use of derivative instructions, the implementation works only on GeForce FX and GeForce 6 Series GPUs.

NVIDIA Ex1023
Page Q-177

Listing 8-2. The Fragment Shader

```
f2fConnector distanceFragment(v2fConnector v2f,
  uniform sampler2D colorTex,
  uniform sampler2D normalTex,
  uniform sampler3D distanceTex,
  uniform float3 normalizationFactor)
{
  f2fConnector f2f;

  // Normalize the offset vector in texture space.
  // The normalization factor ensures we are normalized with respect
  // to a distance which is defined in terms of pixels.
  float3 offset = normalize(v2f.tanEyeVec);
  offset *= normalizationFactor;

  float3 texCoord = v2f.texCoord;

  // March a ray
  for (int i = 0; i < NUM_ITERATIONS; i++) {
    float distance = f1tex3D(distanceTex, texCoord);
    texCoord += distance * offset;
  }

  // Compute derivatives of unperturbed texcoords.
  // This is because the offset texcoords will have discontinuities
  // which lead to incorrect filtering.
  float2 dx = ddx(v2f.texCoord.xy);
  float2 dy = ddy(v2f.texCoord.xy);

  // Do bump-mapped lighting in tangent space.
  // 'normalTex' stores tangent-space normals remapped
  // into the range [0, 1].
  float3 tanNormal = 2 * f3tex2D(normalTex, texCoord.xy, dx, dy) - 1;
  float3 tanLightVec = normalize(v2f.tanLightVec);
  float diffuse = dot(tanNormal, tanLightVec);

  // Multiply diffuse lighting by texture color
  f2f.COL.rgb = diffuse * f3tex2D(colorTex, texCoord.xy, dx, dy);
  f2f.COL.a = 1;

  return f2f;
}
```


NVIDIA Ex1023
Page Q-178

In all examples in this chapter, we have used a 3D texture size of $256 \times 256 \times 16$, but this choice may be highly data-dependent. We also experimented with maps up to $512 \times 512 \times 32$ for complex data sets. We set the value of `NUM_ITERATIONS` to 16 iterations for our examples. In most cases, this was more than sufficient, with 8 iterations sufficing for smoother data sets.

Our fragment shader, assuming 16 iterations, compiles to 48 instructions for the GeForce FX. Each iteration of the loop takes 2 instructions: a texture lookup followed by a multiply-add. GeForce FX and GeForce 6 Series GPUs can execute this pair of operations in a single clock cycle, meaning that each iteration takes only one clock cycle. It should be noted that each texture lookup depends on the previous one; on GPUs that limit the number of indirect texture operations, this shader needs to be broken up into multiple passes.

We tested our distance-mapping implementation on a GeForce FX 5950 Ultra and on a GeForce 6800 GT with several different data sets. One such data set is pictured in Figure 8-6. On the GeForce FX 5950 Ultra, we can shade every pixel at 1024×768 resolution at approximately 30 frames per second. If we reduce the number of iterations to 8, we obtain approximately 75 frames per second at the same resolution. On the GeForce 6800 GT, we were able to run at a consistent 70 frames per second, even shading every pixel at 1280×1024 resolution with 16 iterations.

8.7 Conclusion

We have presented distance mapping, a fast iterative technique for displacement mapping based on ray tracing of implicit surfaces. We show that the information contained in a distance function allows us to take larger steps when rays are farther from the surface, while ensuring that we never take a step so large that we create gaps in the rendered geometry. The resulting implementation is very efficient; it converges in a few iterations, and each iteration costs only a single cycle on GeForce FX and GeForce 6 Series GPUs.

In the future we would like to use the dynamic branching capabilities of Shader Model 3 GPUs to improve the performance of our technique by taking "early outs" for pixels that converge quickly. This would allow us to increase the quality of the technique by devoting more computing power to pixels that converge more slowly.

We would also like to adapt our technique to curved surfaces. Although our algorithm can be used on any model with appropriate tangents, it results in distortion in regions

NVIDIA Ex1023
Page Q-179



Figure 8-6. A Grating Rendered with Displacement Mapping

of high curvature. In particular, generalized displacement mapping (Wang et al. 2004) uses a type of tetrahedral projection to account for curved surfaces, and this method can also be applied to our algorithm. Such a tetrahedral projection can be accomplished in a vertex shader (Wylie et al. 2002).

8.8 References

Cook, Robert L. 1984. "Shade Trees." In *Computer Graphics (Proceedings of SIGGRAPH 84)* 18(3), pp. 223–231.

Danielsson, Per-Erik. 1980. "Euclidean Distance Mapping." *Computer Graphics and Image Processing* 14, pp. 227–248.

Hart, John C. 1996. "Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces." *The Visual Computer* 12(10), pp. 527–545.

NVIDIA Ex1023
Page Q-180

Kaneko, Tomomichi, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. 2001. "Detailed Shape Representation with Parallax Mapping." In *Proceedings of the ICAT 2001 (The 11th International Conference on Artificial Reality and Telexistence)*, Tokyo, December 2001, pp. 205–208.

Policarpo, Fabio. 2004. "Relief Mapping in a Pixel Shader Using Binary Search." **http://www.paralelo.com.br/arquivos/ReliefMapping.pdf**

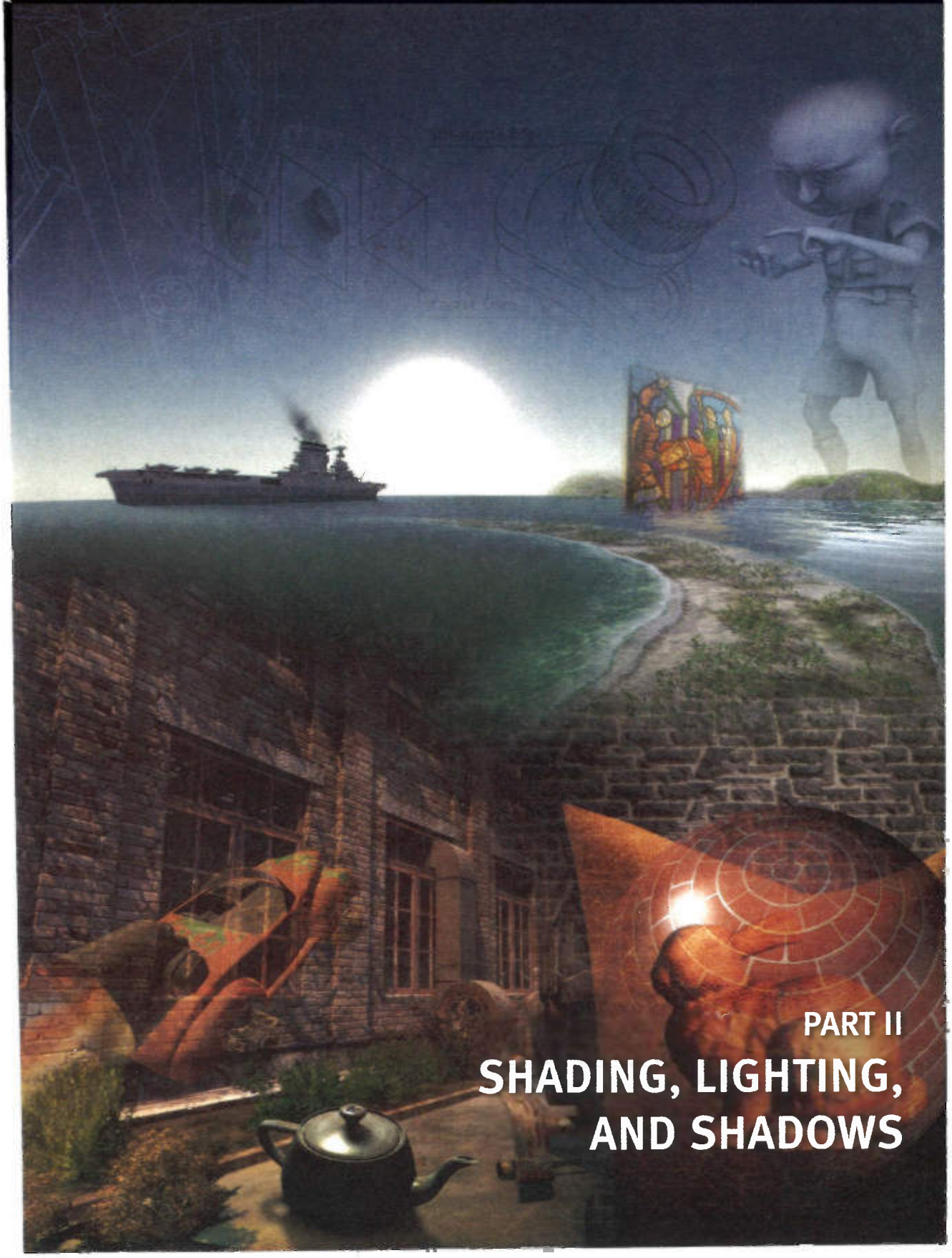
Wang, Lifeng, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. 2003. "View-Dependent Displacement Mapping." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)* 22(3), pp. 334–339.

Wang, Xi, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. 2004. "Generalized Displacement Maps." In *Eurographics Symposium on Rendering 2004*, pp. 227–234.

Wylie, Brian, Kenneth Moreland, Lee Ann Fisk, and Patricia Crossno. 2002. "Tetrahedral Projection Using Vertex Shaders." In *Proceedings of IEEE Volume Visualization and Graphics Symposium 2002*, October 2002, pp. 7–12.

Special thanks to Stefanus Du Toit for his help implementing the distance transform and writing the Sh implementation.

NVIDIA Ex1023
Page Q-181



PART II
SHADING, LIGHTING, AND SHADOWS

The subject of shading, lighting, and shadows is a broad one, which touches many areas of real-time 3D applications, from the low-level instructions that determine how a pixel is colored or a vertex is positioned, to the higher-level ideas that govern the content creation and art pipeline.

Each year, GPUs with more and more features and higher performance become available; much of that extra power and flexibility is poured into "solving" the problem of creating realistic shading, lighting, and shadows. But we are decidedly not there yet, and we may never be. Every current algorithm for real-time shadows, for example, has artifacts in certain cases, or performance problems in others, or most likely both. Similarly with lighting: you often have to trade interactivity or scene complexity for quality. This part of the book is all about making good choices among these trade-offs and creating great-looking and fast real-time renderings.

In **Chapter 9, "Deferred Shading in *S.T.A.L.K.E.R.*," Oles Shishkovtsov** of GSC Game World details the deferred shading architecture developed for the game *S.T.A.L.K.E.R.* Deferred shading has gained popularity lately, but there are a number of nonobvious pitfalls when one goes to implement this approach. This chapter covers solutions to these problems and explains how to create a robust, flexible, and fast deferred renderer.

Irradiance environment maps enable fast, high-quality diffuse and specular lighting, with one significant caveat: the lighting environment must be static. **Gary King** of NVIDIA shows how to avoid this limitation in **Chapter 10, "Real-Time Computation of Dynamic Irradiance Environment Maps"** by using newer GPU features to dramatically accelerate irradiance map creation, making it feasible in real time for use with dynamic scenes.

Accurately rendering complex materials such as denim and wool is a tricky business; the usual texture-mapping approaches don't give realistic results for such materials. In **Chapter 11, "Approximate Bidirectional Texture Functions," Jan Kautz** of the Massachusetts Institute of Technology presents a method for approximating BTFs (which are a very general description of a material's properties) using precomputation combined with a runtime calculation that is just a fairly simple pixel shader looking up into a volume texture.

NVIDIA Ex1023
Page Q-183

The amount of texture detail possible in a scene often has the single greatest impact on the overall visual quality; however, texture detail is often fundamentally limited by hardware texture-size constraints and texture memory space issues. In **Chapter 12, "Tile-Based Texture Mapping," Li-Yi Wei** of NVIDIA works around these restrictions by synthesizing larger virtual textures at runtime from a smaller set of texture tiles using pixel shaders.

The increased flexibility of modern GPUs is rapidly narrowing the gap between renderings that traditionally could only be done using software algorithms and those that are fully hardware-accelerated. In **Chapter 13, "Implementing the mental images Phenomena Renderer on the GPU," Martin-Karl Lefrançois** of mental images introduces the architecture of the mental ray renderer, used in the production of numerous feature films, and describes the implementation of its GPU-accelerated components.

The algorithms usually employed for ambient occlusion and indirect lighting are confined to limited environments that cannot be animated, thus limiting their usefulness. In **Chapter 14, "Dynamic Ambient Occlusion and Indirect Lighting," Michael Bunnell** of NVIDIA describes a technique for accelerating the computation of ambient occlusion and indirect lighting using the GPU, turning the algorithms into real-time solutions.

When visualizing complex objects and architectures, perfect photorealism is often not ideal for understanding structures and object relationships, and non-photorealistic techniques are necessary to aid comprehension. **Marc Nienhaus** and **Jürgen Döllner** of the University of Potsdam explain some techniques that make good use of the latest features of modern GPUs for achieving this goal in real time in **Chapter 15, "Blueprint Rendering and 'Sketchy Drawings.'"**

In **Chapter 16, "Accurate Atmospheric Scattering," Sean O'Neil** presents a flexible real-time hardware-accelerated technique for simulating atmospheric scattering that works as well on the ground as it does from space, producing some truly gorgeous visuals.

NVIDIA Ex1023
Page Q-184

In *GPU Gems* (Addison-Wesley, 2004), efficient, high-quality soft shadows were the focus of a number of chapters. **Yury Uralsky** of NVIDIA shows there's still a lot of life left in the topic in **Chapter 17, "Efficient Soft-Edged Shadows Using Pixel Shader Branching."** Yury covers techniques for rendering high-quality soft shadow edges as well as optimization techniques based on using pixel shader early-outs to minimize shadow computation cost.

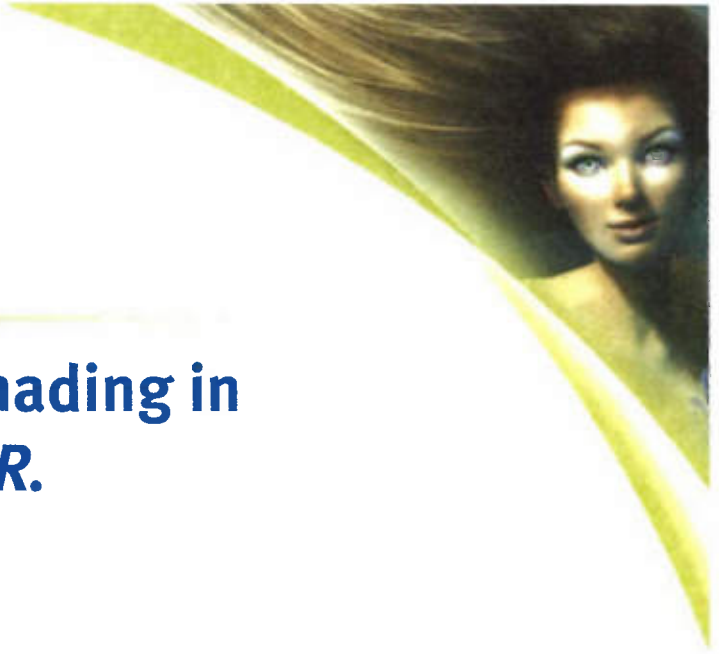
Yuri Kryachko of 1C:Maddox Games presents an overview of the water effect from the game *Pacific Fighters* in **Chapter 18, "Using Vertex Texture Displacement for Realistic Water Rendering."** The technique uses the ability of newer GPUs to perform texture fetches at the vertices of a mesh to do true displacement mapping on the water surface, resulting in very high quality and efficient animated water that looks incredibly realistic.

Finally, **Tiago Sousa** of Crytek introduces a technique for simulating refractive surfaces such as water or glass in **Chapter 19, "Generic Refraction Simulation."** This technique is a generalization of an effect used in the popular game *Far Cry*.

Looking back on the year since writing my introduction to the Lighting and Shadows section of the original *GPU Gems*, it's clear that the pace of innovation has not slowed. Indeed, many of the techniques in this section were simply impossible to achieve a year ago, and many others were impractical. I, for one, can't wait to see what new techniques people come up with in the year ahead!

Cem Cebenoyan, NVIDIA

NVIDIA Ex1023
Page Q-185



Chapter 9

Deferred Shading in *S.T.A.L.K.E.R.*

Oles Shishkovtsov
GSC Game World

This chapter is a post-mortem of almost two years of research and development on a renderer that is based solely on deferred shading and 100 percent dynamic lighting, targeted at high-end GPUs. Because no single solution can suit all needs, this chapter should not be considered a comprehensive guide to performing deferred shading.

9.1 Introduction

For those who are not familiar with the concepts of deferred shading, we recommend Hargreaves and Harris 2004. This presentation is a good introduction to the basics, and it briefly showcases a number of techniques originally developed for and used in the game *S.T.A.L.K.E.R.* With deferred shading, during scene-geometry rendering, we don't have to bother with any lighting; instead, we just output lighting properties such as the position and normal of each pixel. Later we apply lighting as a 2D post-process using this intermediate buffer—usually called a *G-buffer* (Saito and Takahashi 1990)—as input to the lighting shader.

When most people first think about deferred shading, they envision nice algorithmic properties such as perfect depth complexity for lighting, predictable performance proportional to the lights' screen-space areas, simplified scene management, and more. One additional crucial fact is usually missed: the ability to cut down on large numbers of batches, which are inevitable when dynamic shadows come into the game. Ask a

NVIDIA Ex1023
Page Q-186

representative of your favorite IHV, who will say that many games are still CPU-limited. Reduced CPU usage was one of the main reasons we chose deferred shading in *S.T.A.L.K.E.R.* Another factor was that forward shading engines (with traditional immediate shading architectures, such as those in *Doom 3* or *Far Cry*) usually unnecessarily pay the high cost of repeating the same work—vertex transform, anisotropic texture filtering, parallax mapping, normal map decompression, detail texturing, and so on. In the case of *S.T.A.L.K.E.R.*, where we were limited both by CPU and by vertex processing and had moderate overdraw, we had almost the ideal case for deferred shading. So, to meet our goal and to raise the visual bar—rendering high-quality, high-polygon content with fully dynamic lighting and shadowing—deferred shading was the inevitable choice. Figures 9-1 and 9-2 show examples of a scene generated by our forward shading and deferred shading renderers, respectively.



Figure 9-1. Screenshot from the First, Forward Shading Renderer
The original renderer was based on DirectX 8.

NVIDIA Ex1023
Page Q-187



Figure 9-2. Screenshot from the Second, Deferred Shading Renderer
The second renderer targets DirectX 9-class hardware.

9.2 The Myths

The first myth—"Deferred shading is slow on current hardware"—arises mostly from the fact that the current generation of games tries to load-balance most of the lighting work between the vertex and pixel pipelines. When all the calculations are performed at pixel level (this is the only possible way to go with deferred shading), performance will be similar, because lighting pixel shaders for deferred renderers are not that much more complicated than those for forward renderers. The only added work is G-buffer sampling and, possibly, unpacking. But your application is much less likely to become bottlenecked by the CPU or the vertex pipe. The actual mileage will vary depending on the data set and, more important, on your rendering engine's actual *overdraw*, measured as the number of pixels passing the z-test divided by the screen area.

NVIDIA Ex1023
Page Q-188

Another myth is that deferred shading is useless for rendering directional lights. In *S.T.A.L.K.E.R.*'s case, this is true for scenes with a few unshadowed directional lights; but it's false even for a single, shadow-casting directional light (such as the sun). A more detailed explanation appears in Section 9.3.2, on the optimization of the lighting phase.

Because deferred renderers process each object only once, conventional techniques for handling multiple material types (such as changing shaders per-object) do not translate well. However, by using DirectX 9 multiple render targets (MRT), we can render up to 16 material attributes during the G-buffer creation phase (four render targets of four floating-point numbers each). Ten are used to perform basic diffuse and specular lighting (three each for albedo, normal, and position, plus one for gloss), but this leaves six components for controlling the lighting functions. In *S.T.A.L.K.E.R.*, we stored material and object IDs in the spare components and defined functions in each light shader (accelerated using 2D and 3D texture lookup tables) that depended on these values. This freed us from a specific material reflection model, while also allowing us to define a huge number of light shader primitives without the combinatorial increase in the number of shaders that affects forward renderers. In addition to the traditional point, spot, and directional light primitives, we also had special light sources such as wrapped hemispherical sources and volumetric fog. This approach can be considered an inversion of the shaders in forward renderers: instead of using light properties to modify material shaders, we used material properties to modify light shaders. By adapting our material system to take advantage of the benefits of deferred shading and using textures to efficiently avoid some of the limitations, we created a deferred rendering engine in which the only limits on the types of materials were the artists' and programmers' imaginations.

During the writing of this chapter, *S.T.A.L.K.E.R.* was in the process of synchronizing the material look between two renderers: the first one targeted at DirectX 8-class hardware, and a second one targeted at high-end DirectX 9-class hardware. An interesting discovery was that more than 90 percent of materials used in the first renderer were just simple modifiers to the base albedo and emissive terms, plus a few that affected the position and surface normal. Examples are the waving of trees or grass under the wind, or different sorts of detail texturing. These modifications were performed identically in every rendering pass, so there was almost a direct mapping to the forward phase of the second renderer, the G-buffer creation phase.

NVIDIA Ex1023
Page Q-189

9.3 Optimizations

Delivering a product that pushes the newest graphics hardware to its limits while still offering a playable game requires a great deal of planning. Both technical and artistic decisions must consider the target audience and the target hardware. Failing to do so may result in significant wasted effort. This happened to us when we changed the target for *S.T.A.L.K.E.R.*'s first renderer from fixed-function T&L (DirectX 7-class) GPUs to first-generation pixel shading (DirectX 8-class) GPUs; we had to rewrite much of our tool set and engine to accommodate the change. Our goals for *S.T.A.L.K.E.R.*'s second (deferred) renderer were for it to be a drop-in replacement for the first renderer (so it would work on the same data set and content, except for the lighting environment), and for it to run with "maximal eye candy" at 30 frames per second on a GeForce 6800 Ultra at 1024×768 resolution. We encouraged our designers to complicate the second renderer's lighting environment whenever the performance exceeded this target.

9.3.1 What to Optimize

Given the structure of a deferred renderer, the obvious places that are likely to be performance bottlenecks are deferring, lighting, and post-processing. Our *S.T.A.L.K.E.R.* performance testing confirmed this:

- Average number of lights per frame in closed spaces: 50
- Average number of lights per pixel in closed spaces: 5
- Average number of lights per pixel in open spaces: less than 1
- Average geometric depth complexity: 2.5

Because one of the primary features of the second renderer is real-time shadowing, shadow creation was a significant bottleneck; in fact, in some scenes it turned out to be the largest individual bottleneck. Given this, our optimization goals were obvious:

- Limit the number of lights drawn to just those that affect the rendered image.
- Convert shadow-casting lights to unshadowed lights where possible.
- Simplify the lighting shaders as much as possible.
- Optimize post-processing and deferring passes, but not at the expense of quality.

We decided to favor quality over performance for the G-buffer creation and post-processing (G-buffer shading) passes because these have tightly bounded costs, whereas

NVIDIA Ex1023
Page Q-190

a game player is free to drop multiple torches in the same place, making lighting arbitrarily expensive. Also, most artists are comfortable optimizing for forward renderers (by reducing the number of polygons or the resolution of textures, adding good occluders, and so on), and these optimizations also benefit the deferring pass.

9.3.2 Lighting Optimizations

The most important optimization for the lighting pass is to render only those lights that actually affect the final image, and for those lights, render only the affected pixels. For *S.T.A.L.K.E.R.*, we used a hierarchical occlusion-culling system that utilized both the CPU and the GPU. Our coarsest test was sector-portal culling followed by CPU-based occlusion culling (similar to the hierarchical z-buffer [Greene et al. 1993]). In a typical closed-space frame, this reduced the number of lights by up to 30 to 50 percent. Then we used DirectX 9's occlusion query to eliminate the completely occluded lights. Finally, we used a stencil mask to tag affected pixels for each light. All together, these culling optimizations resulted in a twofold or even larger performance increase.

Even with perfect culling, though, lighting is still the most expensive aspect of a deferred renderer. Therefore, it is important to make sure that the shaders and artistic properties of the light sources are optimized as much as possible. About half of the time we spent creating *S.T.A.L.K.E.R.*'s deferred renderer was devoted to searching for ways to squeeze additional cycles out of our lighting shaders. The primary artistic decisions we made were these:

- **Does the light need to cast shadows?**
  - Many lights are needed only to mimic a global illumination-style look, and shadows just ruin the illusion.
- **Does the light need a projective image?**
  - If so, maybe the projection is the same as the shadow-map projection (which saves a few `dp4`s)?
- **Does the light need to cast shadows from translucent surfaces?** See Figure 9-3.
  - If so, are there any translucent surfaces that exist in the light's frustum?
  - If both projective image and translucent color-modifier are required, maybe we can combine them into one image?
- **Does the light need to contribute to glossy specular reflection?**
  - Many lights in *S.T.A.L.K.E.R.* were added to mimic global illumination. For these lights, we used very simple diffuse-only lighting.

NVIDIA Ex1023
Page Q-191



Figure 9-3. Cast Shadows from Translucent Objects

- Does the light move?
  - If it doesn't, we precomputed shadow-caster visibility to make shadow generation more efficient, by incrementally testing visibility for primitives that are found to be static to each other in a particular light configuration, thus forming a conservative potentially visible set (PVS) with a set of rules that can invalidate parts of it.

Additionally, we made some special optimizations for directional lights (such as the sun).

Optimizing the Sun

Although large-scale shadow mapping from directional light sources is a quite interesting and difficult problem, it is beyond the scope of this chapter. We used a single 2048×2048-pixel shadow map for most of the sun lighting, using a perspective shadow map-style projection transform to maximize area near the viewer. A demo showcasing a variety of related techniques can be found in King 2004. Sun shadows need very careful

NVIDIA Ex1023
Page Q-192

filtering and therefore have a relatively high per-pixel cost, so we want to skip as much work as possible; this is where deferred shading comes in. The first optimization comes from the fact that the sky box doesn't need to be shaded, and it occupies a significant part of the screen (30 to 40 percent is common in a typical outdoor environment). Also, in average configurations of the viewer and sun direction, approximately 50 percent of pixels face away from the sun and don't need to have complex shading applied. Finally, the pixels that have an ambient occlusion term of zero cannot be reached by sun rays, so they can also be excluded from processing. We use a shader to quickly determine these conditions per-pixel, and store the results in a stencil mask so that the GPU's hierarchical culling hardware can avoid applying complex shading on these pixels.

The final algorithm for sunlight accumulation is shown in Listing 9-1.

Note: There is an option to use a single-sample shadow term in pass 0 to avoid the lighting and shadow filtering cost for points in shadow. This doesn't work in practice because (1) the performance was lower, and (2) a single sample is insufficient, because the percentage-closer filtering (PCF) kernel in pass 1 is quite large.

This idea is more useful for Pixel Shader 3.0 dynamic branching, because it has high screen-space coherence and because these shadow samples can be utilized in the final computation. This is left as an exercise for the reader.

We can compute the theoretical performance increase due to this stencil mask: assuming that 15 percent of pixels have an ambient occlusion of zero, and 30 percent of the frame is occupied by the sky, the expected percentage of pixels that will be shaded is $(1 - 0.3) \times 0.5 \times (1 - 0.15) \approx 30$ percent, or about a $3\times$ increase in frames per second. The actual performance increase measured in *S.T.A.L.K.E.R.* was close to $2\times$.

Listing 9-1. Pseudocode for Sunlight Accumulation

```
Pass 0: Render full-screen quad only where 0x03==stencil count
        (where attributes are stored)
  If ((N dot L) * ambient_occlusion_term > 0)
    discard fragment
  Else
    color = 0, stencil = 0x01
Pass 1: Render full-screen quad only where 0x03==stencil count
  Perform light accumulation / shading
```


NVIDIA Ex1023
Page Q-193

9.3.3 G-Buffer-Creation Optimizations

The biggest individual step you can take to improve the performance of creating the G-buffer is selecting the appropriate format for the buffer itself.

Contrary to a common misconception, you don't need a feature-rich GPU to do deferred shading. All you need is DirectX 9 Pixel Shader 2.0 support; no fancy surface formats or even MRT are strictly required. All of these extras can be considered optimizations for performance and quality. Implementations exist for other platforms as well, such as the Microsoft Xbox and the Sony Playstation 2 (Geldreich et al. 2004).

First we have to define the space in which we work. Due to the limited range of components, we are forced to work in a tight, restricted space. The obvious candidates are view space and post-projection space. Then comes the question: How much performance can we dedicate to unpacking?

The drawback of any fat-format encoding (that is, having buffers larger than 32 bits per texel) is the reading speed. For example, a 64-bit texel can take twice as much time to load as a 32-bit one, even when the entire surface fits inside the texture cache (and 128-bit texels are even slower).

So let's start with the assumption (which is not the case, in fact) that we are limited by MRT surface count.We outline the major features of the most practical solutions first for position encoding and then for normal encoding. See Table 9-1.

Depth encoding is usually done in one of two ways:

1. **Storing z/w of the visible surface at each pixel.** Along with the (x, y) pixel coordinates and information from the perspective projection, this is enough information to reconstruct the 3D eye-space position.
2. **Storing distance from the near plane, thus avoiding the division in the G-buffer-creation phase.** Along with interpolated ray direction (from the eye position to the center of the pixel on the near plane), this is enough to find the exact point of intersection.

There is no row labeled "Precision of the Method" in Table 9-1 because the methods are all similar in this respect. There may be differences due to hardware precision, but assuming fp32-capable hardware, we can build the following list, from best to worst:

- Depth, encoded as R32F (single-component, 32-bit floating point)
- Post-projection space, integer encoding, all three components (x, y, z) stored
- Depth, encoded in 24-bit R8G8B8
- Eye-space, fp16 encoding, all three components (x, y, z) stored

NVIDIA Ex1023
Page Q-194

Table 9-1. Storage Options for the Position Buffer

	Depth Only (32 bits per pixel)		Full (*x*, *y*, *z*) Position (64 bits per pixel)	
	R32F	A8R8G8B8	Post-Projection Space	Eye Space
Hardware Support	GeForce FX or better Radeon 9500 or better	All	GeForce FX or better Radeon 9500 or better	GeForce FX or better Radeon 9500 or better
G-Buffer Creation Cost	Single `mov` Low bandwidth	Three `texs` Three `muls` One `mov`	One `rcp` One `mul` One `mov`	Single `mov`
Decoding Cost	One `mov` Four `dp4s` One `rcp` One `mul` for screen-space reconstruction Similar cost for ray-based reconstruction	Same as the R32F, plus one `dp3`	Four `dp4s` One `rcp` One `mul`	None
Sampling and Storage	32 bits	32 bits	64 bits	64 bits
Virtual Position Support	Limited	Limited	Arbitrary, but expensive	Arbitrary
Free Components	0	1	1	1

This ordering depends, of course, on the application, but the differences are not significant enough to worry about in a typical open-space, first-person shooter view.

The options for storing the normals (stored in eye space, which has obvious benefits for lighting calculations) are shown in Table 9-2.

Interestingly enough, *S.T.A.L.K.E.R.* was actually working using each of the encodings shown in Table 9-2 at different times during development, and on quite different hardware, with combinations limited only by the fact that rendering targets must have the same bit depth on current hardware when using MRT. The only interesting exception was the NVIDIA GeForce FX line of GPUs, where deferring was performed via DirectX 9 multi-element textures (MET), instead of multiple render targets, but was then decompressed into ordinary surfaces, mostly because of sampling performance and latency issues.

We can make quality and performance trade-offs using the options enumerated in these tables and some basic renderer statistics. Because many lights affect every pixel in *S.T.A.L.K.E.R.*, sampling and decoding affects performance more than any other properties. On the other hand, the quality of the stored normals, and the ability to support virtual position (more about this later) greatly influence the resulting image quality. Free components can be used to improve performance, too, but only in limited cases.

NVIDIA Ex1023
Page Q-195

Table 9-2. Storage Options for the Normal Buffer

	A8R8G8B8	A2R10G10B10	R16G16F	A16R16G16B16F
Hardware Support	All	Radeon 9500 or better	GeForce FX or better Radeon 9500 or better	GeForce FX or better Radeon 9500 or better
Deferring Cost	One mad	One mad	None	None
Decoding Cost	One mad One nrm	One mad One nrm	One mov One dp2a One rsq One rcp	None
Sampling and Storage	32 bits	32 bits	32 bits	64 bits
Quality	Poor	Good for rough surfaces	Excellent	Excellent
Free Components	1	1 Very low precision	0	1

In addition to position and normal, *S.T.A.L.K.E.R.*'s pipeline required three other attributes: gloss mask, an ambient occlusion term, and material index. The ideal configuration for us became: material (FX8), ambient-occlusion (FX8), gloss-mask (FX8), position (3xFP16), and normal (3xFX16). The closest configuration we were able to achieve, using DirectX-supported formats, was eye-space position and ambient occlusion stored in A16R16G16B16F, eye-space normal and material ID stored in another A16R16G16B16F, and RGB albedo and gloss stored in A8R8G8B8. Because of a limitation of current graphics hardware that requires the same bit depth for surfaces used in the MRT operation, the albedo-gloss surface was expanded to full A16R16G16B16F. This encoding could be seen as a waste of video memory bandwidth—and it is indeed. However, this "wasteful" configuration was actually the fastest on a GeForce 6800 Ultra, even when compared to smaller, lower-quality alternatives, because of its much simpler decoding. That is why *S.T.A.L.K.E.R.* dropped support for any other layout, although we do check for `MRT_INDEPENDENT_BITDEPTH` to try to utilize a 32-bit format for albedo and gloss, even if no hardware capable of doing so currently exists in the market.

9.3.4 Shadowing Optimizations

Stencil shadows and shadow maps both work well within deferred renderers. For artistic reasons, we chose to use shadow maps exclusively in *S.T.A.L.K.E.R.* This allowed us to cast shadows from semitransparent geometry and adjust the shadow boundary based on each pixel's virtual position. However, most important, we were able to generate smooth, soft shadow transitions, which is not possible with stencil shadows.

NVIDIA Ex1023
Page Q-196

We implemented only the basic types of shadowed light sources, because highly specialized implementations allowed us to optimize them greatly. The next section briefly touches on spotlights, but our main focus is on omnidirectional lights, which are the most difficult light sources from which to cast shadows, especially if you're concerned about performance.

Efficient Omni Lights

There are three major options for omnidirectional shadow mapping in deferred renderers:

- Using a cube map for storing distance from the light center, R32F or A8R8G8B8 packed.
- Using a 2D surface with "unrolled" cube-map faces, with reindexing done through a small cube map (called *virtual shadow depth* cube texture [King and Newhall 2004]).
- Treating the point light as six spotlights and rendering each separately.

There is another technique, called "dual-paraboloid shadow maps," but it is not generally applicable to deferred shading, because a nonlinear projection applied first at the vertex level and then at the pixel level can lead to major artifacts.

Let's outline the major benefits and drawbacks, as shown in Table 9-3. Here, by *scalability* we mean using different sizes for different faces, which can enable packing shadow maps for many lights on one surface. *Continuity* means we are able to change the resolution smoothly from, say 1024×1024 to 16×16 in one-pixel decrements.

Treating point lights like six spotlights is difficult for forward shading engines due to the amount of retransformed geometry for every face, but for deferred shading, it is probably the best choice. *S.T.A.L.K.E.R.* was doing a lot of shadow-filtering work, so that was the option we chose. One nuance to keep in mind: leave a small border around each spotlight (small, but larger than the filter-kernel radius) to avoid filtering glitches at frustum edges.

9.4 Improving Quality

The architecture of a deferred renderer can greatly affect the final image quality, as can the decisions made on what to store in the G-buffer. This section briefly outlines a few decisions and tricks we used in *S.T.A.L.K.E.R.* to achieve the level of visual quality we needed.

NVIDIA Ex1023
Page Q-197

Table 9-3. Comparison of Omni Lighting Techniques

	Cube Map	Virtual Shadow Depth Cube Texture	Six Spotlights
Scalability and Continuity	Low Few fixed sizes All faces are the same	Moderate Faces can be of different sizes, but only from a few fixed sets	Excellent Any variation of sizes is possible
Hardware Filtering Support	No	Yes	Yes
Cost of Filtering	Moderate	Excellent for bilinear Moderate for arbitrary percentage-closer filtering	Excellent
Render Target Switches	Six	One	One
Packing Support	No	Yes	Yes
Cost of Screen Space Stencil Masking	Low	Low	Moderate Some stencil overdraw
Memory Cost and Bandwidth Usage	High Surface is almost unusable for everything else	Moderate Few fixed sizes limits packing ability	Excellent

9.4.1 The Power of "Virtual Position"

Simulated displacement mapping is extremely popular in next-generation engines. (See Chapter 8 of this book, "Per-Pixel Displacement Mapping with Distance Functions," for a related technique.) The simplest (and most popular) of these approaches is parallax mapping (Kaneko et al. 2001, Welsh 2004). It works reasonably well with most lighting models, except when it comes to shadows. It looks really strange when the shadows are still, but the underlying surface changes due to different eye position or lighting conditions. Our trick to hide such objectionable effects was to move the eye-space position slightly away from the surface in the direction of the per-pixel normal, in proportion to the height map value at the original sampling point, as shown in Figures 9-4 and 9-5. Doing so adds a greater sense for the bumpiness of the surface close to shadow boundaries or really close to light sources. Another, unintended, effect of this trick was alleviating most of the biasing problems associated with shadow maps. The reason why this works is simple: with good filtering, the worst "surface acne" appears in areas with low heights—which is almost desirable, because in real life, pits are usually darker from self-occlusion.

NVIDIA Ex1023
Page Q-198

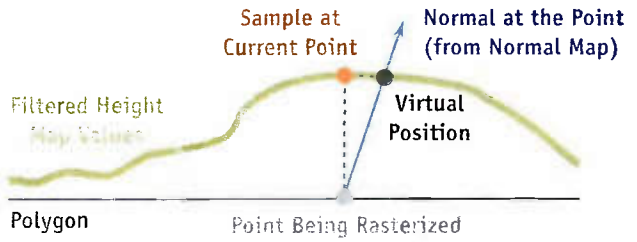


Figure 9-4. The Virtual Position Trick
This technique simulates parallax effects in bump-mapped geometry. It offsets the point being shaded based on the bump map's normal and the height of the bump map surface at the point.



Figure 9-5. The Virtual Position Trick in Action
Note how the shadow follows the bump on the fence in the lower right.

9.4.2 Ambient Occlusion

Our most ambitious goal in writing *S.T.A.L.K.E.R.*'s second renderer was to create a purely dynamic renderer that didn't depend on any precalculation, especially for the lighting environment. We met the goal, with two exceptions:

NVIDIA Ex1023
Page Q-199

1. The sector-portal structure and low-polygonal geometry used for occlusion culling was already available from the data set created for the first renderer.
2. The ambient occlusion term was available too, but it was stored mostly in light maps, used by the first renderer.

Using visibility data may seem like a compromise, but this data doesn't require much preprocessing (we don't use any sort of PVS and mostly rely on occlusion culling). On the other hand, using light maps for the ambient occlusion term seemed to be a big waste of both memory and designer time. Instead, we stored the occlusion term in an unused per-vertex component, and this worked well enough for *S.T.A.L.K.E.R.* The way we handle it: tessellating original geometry up to a constant edge length (0.5 meters in our case), then applying adaptive tessellation, estimating both edge error and the triangle-middle-point error up to a minimum edge length (0.1 meters in our case). The next step was simplification based on the Quadric Error Metric algorithm (see Garland and Heckbert 1998 and Hoppe 1999) with intermediate errors recomputed at each iteration with a simple placement policy of three variants: edge middle, first, or second vertex. We allow no more than a 10 percent increase in vertex count and use a fixed maximal error limit, but in practice we did not reach this limit on vertices in any of our levels. Although we still allowed the usage of light maps to store the ambient occlusion term, we encouraged our artists to shift to a vertex-based solution. The result: our first renderer was changed to use the vertex-based ambient occlusion, too.

9.4.3 Materials and Surface-Light Interaction

The generic way of doing materials in a deferred renderer—store all the information the light-shader needs—is the right way to go in the future, because "Hey! The real world is shaded by the same shader." The alternative is to use something like material index and to apply different fragment programs to different materials branching on this index—this is fairly questionable on today's hardware though.

To maintain the target frame rate, we limited ourselves to a material index into a light-response lookup table. After many experiments to find an optimal re-parameterization for this texture, we settled on the simplest and most intuitive way: indexing the volume texture by ($N \cdot L$, $N \cdot H$, *material*). This allows a fairly large range of materials to be simulated, from dimmed surfaces to glossy reflecting ones, and even a faux-anisotropic reflection model. Figure 9-6 shows an example.

The beauty of this solution is that materials that come from adjacent layers in the texture can be blended by the texture-sampling unit for free. We also discovered that when

NVIDIA Ex1023
Page Q-200



Figure 9-6. Metallic Material Forced for All Surfaces

the layers are ordered by increasing shininess (or decreasing roughness), we needed only four layers to represent almost everything in *S.T.A.L.K.E.R.* So, the result was a $64 \times 256 \times 4$, A8L8 texture summing up to a 128 K table with fairly coherent access from the pixel shader. We left the diffuse lookup at this relatively high resolution (64 samples), because the function it stores may not be linear, and linear filtering that is done by hardware isn't always acceptable.

9.5 Antialiasing

A deferred renderer is just incompatible with current hardware-assisted antialiasing, unfortunately (Hargreaves and Harris 2004). Thus, antialiasing becomes solely the responsibility of the application and the shader; we cannot rely on the GPU alone. Because aliasing itself arises from the mismatched frequencies of the source signal and of the destination discrete representation, a good approximation of an antialiasing filter is just a low-pass filter, which is simple blurring. This is a zero-cost operation in the

NVIDIA Ex1023
Page Q-201

console world, where any TV display works like a low-pass filter anyway. In the PC world, we need an alternative. Our solution was to trade some signal frequency at the discontinuities for smoothness, and to leave other parts of the image intact. This was performed in a way similar to the edge-detection filters used in nonphotorealistic applications: We detect discontinuities in both depth and normal direction by taking 8 + 1 samples of depth and finding how depth at the current pixel differs from the ideal line passed through opposite corner points. The normals were used to fix issues such as a wall perpendicular to the floor, where the depth forms a perfect line (or will be similar at all samples) but an aliased edge exists. The normals were processed in a similar cross-filter manner, and the dot product between normals was used to determine the presence of an edge. Listing 9-2 shows the code.

The two detectors were then multiplied to produce a single value indicating how much the current pixel "looks like an edge." This value was used to offset four bilinear texture lookups into the composited (near-final) back buffer. The result was automatic weighting of samples with a very strong edge-detection policy that seamlessly handles edge and alpha-test/texkill aliasing without blurring other parts of the image. See Figure 9-7 for a sample result.

Listing 9-2. Edge-Detection Shader Used for Antialiasing

```
struct v2p
{
  float4 tc0: TEXCOORD0;  // Center
  float4 tc1: TEXCOORD1;  // Left Top
  float4 tc2: TEXCOORD2;  // Right Bottom
  float4 tc3: TEXCOORD3;  // Right Top
  float4 tc4: TEXCOORD4;  // Left Bottom
  float4 tc5: TEXCOORD5;  // Left / Right
  float4 tc6: TEXCOORD6;  // Top / Bottom
};

////////////////////////////////////////////////////////////////////////
uniform sampler2D s_distort;
uniform half4 e_barrier;  // x=norm(~.8f), y=depth(~.5f)
uniform half4 e_weights;  // x=norm, y=depth
uniform half4 e_kernel;   // x=norm, y=depth

////////////////////////////////////////////////////////////////////////
half4 main(v2p I) : COLOR
{
```


NVIDIA Ex1023
Page Q-202

Listing 9-2 *(continued)*. Edge-Detection Shader Used for Antialiasing

```
    // Normal discontinuity filter
    half3 nc = tex2D(s_normal, I.tc0);
    half4 nd;
    nd.x =  dot(nc, (half3)tex2D(s_normal, I.tc1));
    nd.y =  dot(nc, (half3)tex2D(s_normal, I.tc2));
    nd.z =  dot(nc, (half3)tex2D(s_normal, I.tc3));
    nd.w =  dot(nc, (half3)tex2D(s_normal, I.tc4));
    nd -= e_barrier.x;
    nd = step(0, nd);
    half ne = saturate(dot(nd, e_weights.x));

    // Opposite coords
    float4 tc5r = I.tc5.wzyx;
    float4 tc6r = I.tc6.wzyx;

    // Depth filter : compute gradiental difference:
    // (c-sample1)+(c-sample1_opposite)
    half4 dc = tex2D(s_position, I.tc0);
    half4 dd;
    dd.x = (half)tex2D(s_position, I.tc1).z +
           (half)tex2D(s_position, I.tc2).z;
    dd.y = (half)tex2D(s_position, I.tc3).z +
           (half)tex2D(s_position, I.tc4).z;
    dd.z = (half)tex2D(s_position, I.tc5).z +
           (half)tex2D(s_position, tc5r).z;
    dd.w = (half)tex2D(s_position, I.tc6).z +
           (half)tex2D(s_position, tc6r).z;
    dd = abs(2 * dc.z - dd)- e_barrier.y;
    dd = step(dd, 0);
    half de = saturate(dot(dd, e_weights.y));

    // Weight
    half w = (1 - de * ne) * e_kernel.x;  // 0 - no aa, 1=full aa

    // Smoothed color
    // (a-c)*w + c = a*w + c(1-w)
    float2 offset = I.tc0 * (1-w);
    half4 s0 = tex2D(s_image, offset + I.tc1 * w);
    half4 s1 = tex2D(s_image, offset + I.tc2 * w);
    half4 s2 = tex2D(s_image, offset + I.tc3 * w);
    half4 s3 = tex2D(s_image, offset + I.tc4 * w);
    return (s0 + s1 + s2 + s3)/4.h;
}
```


NVIDIA Ex1023
Page Q-203



Figure 9-7. Weight Computed by the Edge-Detection Shader

There is one side note to this approach: the parameters/delimiters tweaked for one resolution do not necessarily work well for another; even worse, they often do not work at all. That is because the lower the resolution, the more source signal is lost during discretization, and blurring becomes a worse approximation of an antialiasing filter. Visually, you get more and more false positives, and the picture becomes more blurred than necessary. However, lowering the blur radius according to resolution works fine.

9.5.1 Efficient Tone Mapping

Both high-dynamic-range lighting and automatic exposure control are natural, easy, and relatively cheap extensions to deferred renderers. But the way *S.T.A.L.K.E.R.* handles this is somewhat different.

The main difference is that we estimate scene luminance based only on the luminance of a blurred bloom surface. So the desired luminance should be set really low, to something like 0.01.

NVIDIA Ex1023
Page Q-204

When building the bloom target, we blend information from the current frame into the previous frame's bloom target. This gives sharper highlights near the really bright parts and allows use of a smaller filter kernel; we used a simple 2×2 cross filter. The side effect of this approach is the appearance of a slight motion blur, but this was even desired by our artists.

After the final average luminance was determined, we blended it into a 1×1 R32F render target to perform adaptation (as in the DirectX 9 SDK sample); however, the blend speed needed to be significantly lower than the bloom's blend speed to avoid resonance effects.

Our initial implementation used the classical tone-map operator similar to the log-average one described in the DirectX SDK. We changed this to a simple linear scale and linear average luminance estimation, because our artists complained that the original one reduced contrast. This works better for us because our data set was designed around the low-dynamic-range rendering used in the first renderer.

The result is spectacular and extremely efficient: enabling tone mapping with automatic exposure control has less than a 1 percent impact on frame rate.

9.5.2 Dealing with Transparency

Unfortunately, deferred renderers are incompatible with any sort of alpha blending. Depth peeling (Everitt 2001) is not an option, given current GPU performance levels.[1] We handled transparency with a straightforward hack: we used forward rendering for transparent primitives with lighting but without shadows. This was done by copying-and-pasting from the first renderer's source tree.

9.6 Things We Tried but Did Not Include in the Final Code

Because of our goal to raise the visual bar higher than any game ever written, even without deep content modification (content was almost fixed at the time of writing, except for constant small changes required by gameplay flow), we tried a few ideas that didn't make it into the actual commercial game.

1. There is another option, usually called "screen-door transparency," but its use is fairly questionable as a generic solution. See Foley et al. 1990.

NVIDIA Ex1023
Page Q-205

9.6.1 Elevation Maps

The concept of elevation maps can be found in Dietrich 2000. The first assumption was that because the layers are drawn only in the first (attribute-deferring) phase, they should not greatly affect performance. Another assumption was that with little position modification (perhaps 1 cm total), we should be able to hide layering with a relatively small number of layers, something like three to five layers.

After implementation of this scheme, we found the following:

- To make 1 cm of displacement, we need at least 32 layers at 1024×768 resolution, so that layering becomes invisible, but even then, the algorithm fails around the sharp edges, where it produces something like "fur."
- Deferring even 4 layers costs an incredible amount of fill and shading power (especially including `texkill`, which can work against early-z), dropping the frame rate by up to 30 percent.
- The antialiasing method described earlier proved to be quite good at masking out artifacts between layers.

Although we decided to drop the scheme altogether, it was fun to see adaptive displacement mapping in almost real time, approximately 3 frames per second with 64 layers. The idea can still be useful for an application with a cheaper G-buffer creation phase.

9.6.2 Real-Time Global Illumination

An old dream for the author of this chapter was to see global illumination in an interactive application, which doesn't depend on any precomputation and works with 100 percent dynamic lighting conditions and a similarly dynamic environment. The algorithm we chose is similar to photon mapping (Jensen 2001), but without any sort of "final gathering." (Although not strictly related, one may want to read a paper of Ingo Wald's, such as Wald 2003.) All the lights were modified to track the changes in their local environment by casting hundreds of rays (on the CPU) in random directions; selecting the fixed amount of the brightest secondary, indirect lights; adding them to the database; and then repeating the same process on them until the minimal threshold was reached or the fixed number of iterations ended. Then everything continued as usual, up to the end of render-frame, where all the secondary lights were just destroyed (cached, in fact).

The first results were at least strange from a visual standpoint: lots of bright spots everywhere, crossed by weird shadows. Then the secondary light was modified to be more

NVIDIA Ex1023
Page Q-206

like a hemispherical spot, with the major contribution along the primary axis (parent reflection vector) and gradually lowering power to the sides. The shadows were disabled altogether, but CPU-based rays were increased several times to provide some sense of occlusion. For the sake of optimization, the specular contribution from the secondary lights was dropped; also the lights used no material information at all—they were pure Blinn-style. The result really looks like a global illumination and was obtained almost in real time: up to 10 frames per second with up to 500 indirect lights, where direct-only lighting runs at around 50 to 60 frames per second on our target hardware.

Although this frame rate is unacceptable for a real-time game application, one can extrapolate the performance growth curve of current GPUs—approximately doubling in performance every year—to hope that in two years, this approach could be practical.

9.7 Conclusion

Deferred shading, although not appropriate for every game, proved to be a great rendering architecture for accomplishing our goals in *S.T.A.L.K.E.R.* It gave us a rendering engine that leverages modern GPUs and has lower geometry-processing requirements, lower pixel-processing requirements, and lower CPU overhead than a traditional forward shading architecture. And it has cleaner and simpler scene management to boot. Once we worked around the deficiencies inherent in a deferred shader, such as a potentially restricted material system and the lack of antialiasing, the resulting architecture was both flexible and fast, allowing for a wide range of effects. See Figure 9-8 for an example. Of course, the proof is in the implementation. In *S.T.A.L.K.E.R.*, our original forward shading system, despite using significantly less complex and interesting shaders, actually ran slower than our final deferred shading system in complex scenes with a large number of dynamic lights. Such scenes are, of course, exactly the kind in which you need the most performance!

9.8 References

Dietrich, Sim. 2000. "Elevation Maps." **http://www.nvidia.com/object/Elevation_Maps_Paper.html**

Everitt, C. 2001. "Interactive Order-Independent Transparency." NVIDIA technical report. **http://developer.nvidia.com/object/Interactive_Order_Transparency.html**

NVIDIA Ex1023
Page Q-207



Figure 9-8. *S.T.A.L.K.E.R.* in Action

Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. 1990. *Computer Graphics: Principles and Practice*. Addison-Wesley.

Garland, M., and P. Heckbert. 1998. "Simplifying Surfaces with Color and Texture Using Quadric Error Metrics." In *IEEE Visualization 98*, October 1998, pp. 263–269.

Geldreich, Rich, Matt Pritchard, and John Brooks. 2004. "Deferred Lighting and Shading." Presentation at Game Developers Conference 2004. **http://www.gdconf.com/archives/2004/pritchard_matt.ppt**

Greene, Ned, Michael Kass, and Gavin Miller. 1993. "Hierarchical Z-Buffer Visibility." In *Computer Graphics (Proceedings of SIGGRAPH 93)*, pp. 231–238.

Hargreaves, Shawn, and Mark Harris. 2004. "Deferred Shading." Presentation. **http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf**

NVIDIA Ex1023
Page Q-208

Hoppe, H. 1999. "New Quadric Metric for Simplifying Meshes with Appearance Attributes." In *IEEE Visualization 99*, October 1999, pp. 59–66.

Jensen, Henrik Wann. 2001. *Realistic Image Synthesis Using Photon Mapping*. AK Peters.

Kaneko, Tomomichi, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. 2001. "Detailed Shape Representation with Parallax Mapping." In *Proceedings of the ICAT 2001 (The 11th International Conference on Artificial Reality and Telexistence)*, Tokyo, December 2001, pp. 205–208.

King, Gary. 2004. Practical Perspective Shadow Mapping demo. **http://download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/PracticalPSM.zip**

King, Gary, and William Newhall. 2004. "Efficient Omnidirectional Shadow Maps." In *ShaderX 3*, edited by Wolfgang Engel. Charles River Media.

Saito, T., and T. Takahashi. 1990. "Comprehensible Rendering of 3-D Shapes." In *Computer Graphics (Proceedings of SIGGRAPH 90)* 24(4), August 1990, pp. 197–206.

Wald, Ingo. 2003. Instant Global Illumination, Version II. **http://www.openrt.de/Gallery/IGI2/**

Welsh, Terry. 2004. "Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces." **http://www.infiscape.com/doc/parallax_mapping.pdf**

NVIDIA Ex1023
Page Q-209



Chapter 10

Real-Time Computation of Dynamic Irradiance Environment Maps

Gary King
NVIDIA Corporation

Environment maps are a popular image-based rendering technique, used to represent spatially invariant[1] spherical functions. This chapter describes a fully GPU-accelerated method for generating one particularly graphically interesting type of environment map, *irradiance environment maps*, using DirectX Pixel Shader 3.0 and floating-point texturing. This technique enables applications to quickly approximate complex global lighting effects in dynamic environments (such as radiosity from dynamic lights and dynamic objects). For brevity, this chapter assumes that the reader has working knowledge of environment maps, particularly cube maps (Voorhies and Foran 1994) and dual-paraboloid maps (Heidrich and Seidel 1998).

10.1 Irradiance Environment Maps

Imagine a scene with k directional lights, with directions $d_1..d_k$ and intensities $i_1..i_k$, illuminating a diffuse surface with normal n and color c. Given the standard OpenGL and DirectX lighting equations, the resulting pixel intensity B is computed as:

$$B = c \sum_{j=1..k} \max\left(0,\ d_j \cdot n\right) i_j.$$

Equation 10-1. Diffuse Reflection

1. Kevin Bjorke's chapter in *GPU Gems* (Bjorke 2004) describes a technique to use environment maps in spatially varying ways by applying a spatially varying function to the input texture coordinate.

NVIDIA Ex1023
Page Q-210

In this context, an environment map with k texels can be thought of as a simple method of storing the intensities of k directional lights, with the light direction implied from the texel location. This provides an efficient mechanism for storing arbitrarily complex lighting environments; however, the cost of computing B increases proportionally to the number of texels in the environment map. What we would like is a mechanism for precomputing the diffuse reflection, so that the per-pixel rendering cost is independent of the number of lights/texels.

Looking at Equation 10-1, we notice a few things: First, all surfaces with normal direction n will return the same value for the sum. Second, the sum is dependent on just the lights in the scene and the surface normal. What this means is that we can compute the sum for any normal n in an offline process executed once per environment map, and store the result in a second environment map, indexed by the surface normal. This second environment map is known as the *diffuse irradiance environment map* (or just the diffuse environment map), and it allows us to illuminate objects with arbitrarily complex lighting environments with a single texture lookup. (This includes natural lighting environments generated from photographs, created by Miller and Hoffman [1984], popularized by Paul Debevec [1998], and demonstrated in Figure 10-1.) Some simple pseudocode for generating diffuse environment maps[2] is shown in Listing 10-1. Figure 10-2b shows an example of a diffuse environment map.



Figure 10-1. An Image Generated Using Irradiance Environment Maps
Comparison of an object lit by a single directional light (left) and a real-time irradiance environment map (right).

2. For simplicity, this pseudocode does not properly weight the texels in the input environment map (to account for the sampling rate differences across environment maps). The example code included on the book's CD properly handles this.

NVIDIA Ex1023
Page Q-211

Listing 10-1. Diffuse Convolution Pseudocode

```
diffuseConvolution(outputEnvironmentMap, inputEnvironmentMap)
{
  for_all {T0: outputEnvironmentMap}
    sum = 0
    N = envMap_direction(T0)
    for_all {Ti: inputEnvironmentMap}
      L = envMap_direction(Ti)
      I = inputEnvironmentMap[Ti]
      sum += max(0, dot(L, N)) * I
    T0 = sum
  return outputEnvironmentMap
}
```

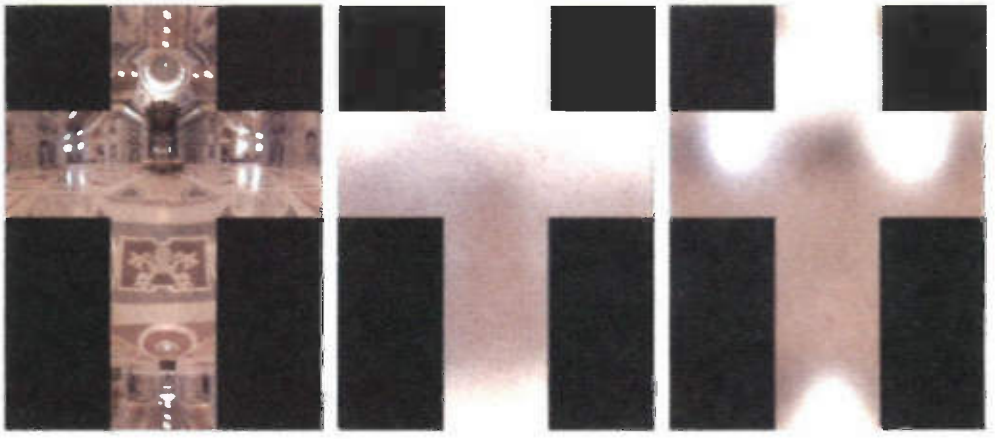


Figure 10-2. Comparing Irradiance Environment Maps
(a) A cube environment map of St. Peter's Basilica. The resulting (b) diffuse and (c) specular maps. St. Peter's light probe © 1999 Paul Debevec, http://www.debevec.org/Probes/.

This concept can be trivially extended to handle popular symmetric specular functions, such as the Phong reflection model. Instead of creating an irradiance environment map indexed by the surface normal, we create one indexed by the view-reflection vector R; and instead of computing `max(0, dot(L, N))` for each input texel, we compute `pow(max(0, dot(R, L)), s)`. These are known as *specular irradiance environment maps*.

NVIDIA Ex1023
Page Q-212

10.2 Spherical Harmonic Convolution

Although rendering with irradiance environment maps is extremely efficient (just one lookup for a diffuse reflection or two for diffuse and specular), generating them using the algorithm in Listing 10-1 is not. For M input texels and N output texels, generating a single irradiance map is an $O(MN)$ operation. In more concrete terms, if we want to generate a diffuse cube map with an edge length of just 32 texels, from an input cube map with an edge length of 64 texels, the cost is $32 \times 32 \times 6 \times 64 \times 64 \times 6 \approx 151$ million operations. Even with CPU speeds approaching 4 GHz, real-time generation of irradiance environment maps cannot be achieved using such a brute-force algorithm.

At SIGGRAPH 2001, a technique was presented that reduced the cost of computing irradiance environment maps from $O(MN)$ to $O(MK)$, with $N >> K$ (Ramamoorthi and Hanrahan 2001b). The key insight was that diffuse irradiance maps vary extremely slowly with respect to the surface normal, so they could be generated and stored more efficiently by using the frequency-space representation of the environment map. To convert a spherical function (such as an environment map) into its frequency-space representation, we use the *real spherical harmonic transform*[3]:

$$L_l^m = \int_S L(\theta,\phi)\, y_l^m(\theta,\phi) \sin\theta \, d\theta \, d\phi.$$

Equation 10-2. Spherical Harmonic Transform

For N discrete samples (such as an environment map), Equation 10-2 becomes:

$$L_l^m = \sum_{i=0}^{N-1} L(i)\, y_l^m(i)\, d\omega(i),$$

Equation 10-3. Discrete Spherical Harmonic Transform

where

$$\sum_{i=0}^{N-1} d\omega(i) = 4\pi$$

is the solid angle subtended by each sample,[4] and definitions for y_l^m (the spherical harmonic basis functions) can be found in Ramamoorthi and Hanrahan 2001b, Green 2003, or any spherical harmonics reference.

3. For those with signal-processing backgrounds, the spherical harmonic transform is analogous to the Fourier transform in Cartesian coordinate systems.

4. The definition of $d\omega$ will depend on the type of environment map used, because the different projections

NVIDIA Ex1023
Page Q-213

While a detailed discussion of spherical harmonics is outside the scope of this chapter, the References section lists a number of excellent articles that provide in-depth explanations (and other applications) of spherical harmonics. In order to use spherical harmonics for computing irradiance maps, all we need to know are some basic properties:

- Any band l has $2l + 1$ coefficients m, defined from $-l..l$. Therefore, an lth-order projection has $(l + 1)^2$ total coefficients.
- The basis functions are linear and orthonormal.
- Convolving two functions in the spatial domain is equivalent to multiplying them in the frequency domain.

The last property (known as the *convolution theorem* in signal-processing parlance) is an extremely powerful one, and the reason why we can use spherical harmonics to efficiently generate irradiance maps: Given two functions that we want to convolve (the Lambertian BRDF and a sampled lighting environment, for example), we can compute the convolved result by separately projecting each function into an lth-order spherical harmonic representation, and then multiplying and summing the resulting $(l + 1)^2$ coefficients. This is more efficient than brute-force convolution because for diffuse lighting (and low-exponent specular), the L_l^m coefficients (Equation 10-3) rapidly approach 0 as l increases, so we can generate very accurate approximations to the true diffuse convolution with a very small number of terms. In fact, Ramamoorthi and Hanrahan (2001a) demonstrate that $l = 2$ (9 coefficients) is all that is required for diffuse lighting, and $l = \sqrt{s}$ is sufficient for specular lighting (where s is the Phong exponent). This reduces the runtime complexity of the example convolution to just $9 \times 64 \times 64 \times 6 \approx 221$ thousand operations per function (442 thousand total, 1 for the lighting environment, and 1 for the reflection function), compared to the 151 million operations using the brute-force technique.

We can further improve the spherical harmonic convolution performance by noticing that the reflection function coefficients are constant for each function (Lambert, Phong, and so on). Therefore, each function's coefficients can be computed once (in a preprocess step) and be reused by all convolutions. This optimization reduces the runtime complexity of the example convolution to just 221 thousand operations.

used by different environment map types (such as spherical, cubic, and azimuth-angle) will cause the relative sampling densities to vary uniquely across the unit sphere. Therefore, to generate a correct result, we need to use the correct solid angle subtended by each texel. The included demo has functions that calculate $d\omega$ for both dual-paraboloid and cubic environment maps.

NVIDIA Ex1023
Page Q-214

The only remaining detail is that our diffuse environment map is now stored in a spherical harmonic representation. To convert it back into its spatial representation,[5] we generate a new environment map by using the inverse spherical harmonic transform (the tilde indicates that the reconstructed function is an approximation):

$$\tilde{L}(\theta, \phi) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} L_l^m y_l^m(\theta, \phi).$$

Equation 10-4. Inverse Spherical Harmonic Transform

Application of this formula is trivial: for every texel in the output environment map, multiply the spherical harmonic basis function by the coefficients. This is an $O(KN)$ process, so the total runtime complexity for generating irradiance environment maps is $O(KN + KM)$.

10.3 Mapping to the GPU

With Pixel Shader 3.0 and floating-point textures (features supported on GeForce 6 Series GPUs), mapping spherical harmonic convolution to the GPU becomes a trivial two-pass operation[6]: one pass to transform the lighting environment into its spherical harmonic representation, and one to convolve it with the reflection function and transform it back into the spatial domain.

10.3.1 Spatial to Frequency Domain

To transform a lighting environment into its spherical harmonic representation, we need to write a shader that computes Equation 10-3. This equation has some interesting properties that can be exploited to make writing a shader a very straightforward process:

- The output is $(l + 1)^2$ three-vector floats (one for each of red, green, and blue).
- The input is N three-vector floats, N scalar $d\omega$ values, and N scalar y_l^m values.
- For any input texel i from any environment map L, $y_l^m(i)\, d\omega(i)$ is a scalar constant.

The last point is the important one: it means that we can generate one texture per (l, m) pair as an offline process. This can then be mapped onto the GPU using $(l + 1)^2$

5. Ramamoorthi and Hanrahan (2001b) also demonstrate how to apply the spherical harmonic coefficients directly, without generating an output environment map.

6. For a sufficiently motivated programmer, the convolution could be performed in multiple Pixel Shader 2.0 passes; however, this distinction does not apply to the author of this chapter.

NVIDIA Ex1023
Page Q-215

rendering passes, where each pass generates a three-vector of spherical harmonic coefficients by looping over all N texels in the input environment map, and multiplying each by the corresponding texel in the appropriate lookup table. We can reduce this to a single pass by tiling the lookup tables, and rendering all of the $(l + 1)^2$ three-vector coefficients at once, as separate pixels in an output texture. Using Pixel Shader 3.0's VPOS input register, we can trivially map the output pixel location to a tile in the lookup table, allowing each of the output pixels to project the input environment map onto the appropriate spherical harmonic basis function (scaled by $d\omega$). This is shown in HLSL pseudocode in Listing 10-2 and demonstrated in Figure 10-3.

Listing 10-2. Spherical Harmonic Projection Pseudocode

```
float2 map_to_tile_and_offset( tile, offsetTexCoord )
{
  return (tile / numTiles) +
         (offsetTexCoord / (numTiles * numSamples))
}

float3 SH_Projection( inputEnvironmentMap,
                      dωY_lm,
                      currentL_lm : VPOS ) : COLOR
{
  sum = 0
  for_all {T_0: inputEnvironmentMap}
    T_1 = map_to_tile_and_offset(currentL_lm, T_0)
    weighted_SHbasis = dωY_lm[T_1]
    radiance_Sample = inputEnvironmentMap[T_0]
    sum += radiance_Sample * weighted_SHbasis
  return sum
}
```

Note that for simplicity, you may want a separate lookup table for each face in the environment map. The included demo uses dual-paraboloid environment maps for input, so two weight lookup tables are generated.

10.3.2 Convolution and Back Again

As described in Section 10.2, the convolution operator in frequency space is just multiplying the L_l^m coefficients generated in Section 10.3.1 with a set of precomputed coefficients for the desired reflection function (for clarity, we refer to these as the A_l^m coefficients), and then evaluating Equation 10-4 for each texel in the output environment map. This is very

NVIDIA Ex1023
Page Q-216

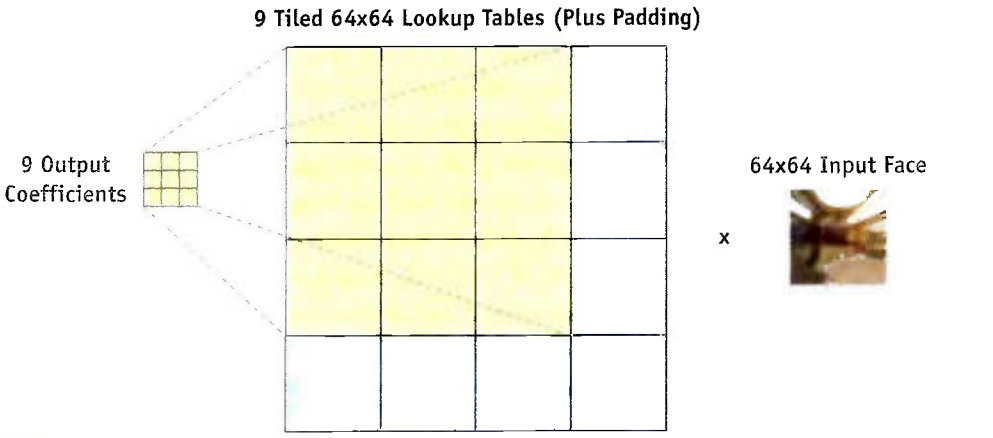


Figure 10-3. Mapping Output Coefficients to Input Lookup Table Tiles for One Face

similar to the process we used with Equation 10-3, and we can use tiling to execute this operation in a single shader pass, too. Because $A_l^m y_l^m$ is a scalar constant for any given (l, m) pair and texel location, we can precompute $(l+1)^2$ environment maps that store these values. If the desired output environment map is a cube map, we can instead precompute six weighting textures (one for each cube-map face), tile the (l, m) coefficients inside each face, and generate the final cube map by iteratively setting each face as the active render target, binding the appropriate weight texture, and executing the pixel shader.

The pixel shader used to perform the convolution plus inverse transformation is very similar to the one used in Section 10.3.1; however, instead of executing an N-element loop over K output values, we execute a K-element loop over M output values. Pseudocode is provided in Listing 10-3.

Listing 10-3. Convolution and Evaluation Pseudocode

```
float3 SH_Evaluation( L_lm,
                      A_lm Y_lm,
                      thetaPhi : VPOS ) : COLOR
{
  sum = 0
  for_all {T_0: L_lm}
    T_1 = map_to_tile_and_offset(T_0, thetaPhi)
    weighted_BRDF = A_lm Y_lm[T_1]
    light_environment = inputEnvironmentMap[T_0]
    sum += light_environment * weighted_BRDF
  return sum
}
```


NVIDIA Ex1023
Page Q-217

As mentioned in footnote 5, Ramamoorthi and Hanrahan (2001b) demonstrate an alternative method for performing diffuse convolution by generating a 4×4 matrix from the first nine L_l^m terms and using this to transform the normal vector. When per-vertex lighting is sufficient, this approach may be more efficient than the one described in this chapter; however, because the convolved cube-map resolution is generally much lower than the screen resolution, the approach described here offers significant performance advantages when per-pixel lighting is desired, or in scenes with high polygon counts.

10.4 Further Work

So far, we've managed to show how to use GPUs to accelerate what is traditionally an offline process. However, because this operation executes in milliseconds on modern GPUs, by using render-to-texture functionality to generate the input environment maps, applications can practically generate diffuse and specular cube maps for dynamic environments. The accompanying demo generates irradiance maps in excess of 300 frames per second on a GeForce 6800 GT. Flat ambient lighting can be replaced with a cube-map lookup that varies as the sun moves through the sky, or as flares are tossed into a small room.

One of the deficiencies of using irradiance environment maps for scene lighting is that correct shadowing is extremely difficult, if not impossible. One technique that offline renderers use to compensate for this is computing a global shadowing term (called *ambient occlusion*) that is used to attenuate the illumination from the environment map (Landis 2002). For static objects, the ambient occlusion value can be precomputed and stored; however, for dynamic objects, this is not possible. In Chapter 14 of this book, "Dynamic Ambient Occlusion and Indirect Lighting," Mike Bunnell provides a method for efficiently approximating ambient occlusion values, so that this same technique can be applied to dynamic characters.

Also, spherical harmonics have a number of valuable properties not explored in this chapter. For example, component-wise linear interpolation between two sets of spherical harmonic coefficients is equivalent to linear interpolation of the spatial representations. Component-wise addition of two sets of coefficients is equivalent to addition of the spatial representations. These two properties (combined with the convolution theorem) provide a powerful set of primitives for approximating numerous complex, global lighting effects. This concept is described in detail in Nijasure 2003.

NVIDIA Ex1023
Page Q-218

10.5 Conclusion

Rendering dynamic, complex lighting environments is an important step for interactive applications to take in the quest for visual realism. This chapter has demonstrated how one method used in offline rendering, irradiance maps, can be efficiently implemented on modern GPUs. By cleverly applying and extending the techniques presented here, real-time applications no longer need to choose between complex static lighting and simple dynamic lighting; they can have the best of both.

10.6 References

Bjorke, K. 2004. "Image-Based Lighting." In *GPU Gems*, edited by Randima Fernando, pp. 307–321. Addison-Wesley. *This chapter describes a technique to use environment maps to represent spatially variant reflections.*

Debevec, P. 1998. "Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography." In *Computer Graphics (Proceedings of SIGGRAPH 98)*, pp. 189–198.

Green, R. 2003. "Spherical Harmonic Lighting: The Gritty Details." **http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf**

Heidrich, W., and H.-P. Seidel. "View-Independent Environment Maps." *Eurographics Workshop on Graphics Hardware 1998*, pp. 39–45.

Kautz, J., et al. 2000. "A Unified Approach to Prefiltered Environment Maps." *Eurographics Rendering Workshop 2000*, pp. 185–196.

Landis, H. 2002. "Production-Ready Global Illumination." Course 16 notes, SIGGRAPH 2002.

Miller, G., and C. Hoffman. 1984. "Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments." **http://athens.ict.usc.edu/ReflectionMapping/illumap.pdf**

Nijasure, M. 2003. "Interactive Global Illumination on the GPU." **http://www.cs.ucf.edu/graphics/GPUassistedGI/GPUGIThesis.pdf**

Ramamoorthi, R., and P. Hanrahan. 2001a. "A Signal-Processing Framework for Inverse Rendering." In *Proceedings of SIGGRAPH 2001*, pp. 117–128.

Ramamoorthi, R., and P. Hanrahan. 2001b. "An Efficient Representation for Irradiance Environment Maps." In *Proceedings of SIGGRAPH 2001*, pp. 497–500.

Voorhies, D., and J. Foran. 1994. "Reflection Vector Shading Hardware." In *Proceedings of SIGGRAPH 94*, pp. 163–166.

NVIDIA Ex1023
Page Q-219



Chapter 11

Approximate Bidirectional Texture Functions

Jan Kautz
Massachusetts Institute of Technology

In this chapter we present a technique for the easy acquisition and rendering of realistic materials, such as cloth, wool, and leather. These materials are difficult to render with previous techniques, which mostly rely on simple texture maps. The goal is to spend little effort on acquisition and little computation on rendering but still achieve realistic appearances.

Our method, which is based on recent work by Kautz et al. (2004), arises from the observation that under certain circumstances, the material of a surface can be acquired with very few images, yielding results similar to those achieved with full *bidirectional texture functions* (or BTFs—we give a full definition in the first section). Rendering with this approximate BTF amounts to evaluating a simple shading model and performing a lookup into a volume texture. Rendering is easily possible using graphics hardware, where it achieves real-time frame rates. Compelling results are achieved for a wide variety of materials.

11.1 Introduction

One of the long-sought goals in computer graphics is to create photorealistic images. To this end, realistic material properties are a necessity. It is especially important to capture the fine spatial variation and mesostructure of materials, because they convey a lot of

NVIDIA Ex1023
Page Q-220

information about the material. Figure 11-1 compares a rendering of a piece of cloth using only a single diffuse texture map and using our method, which preserves spatial variation and self-shadowing. The comparison shows how important the mesostructure of a material is, and it demonstrates that for many materials, simple texture mapping or even bump mapping is not sufficient to render it realistically.

Unfortunately, the acquisition of such material properties is fairly tedious and time-consuming (Dana et al. 1999, Lensch et al. 2001, Matusik et al. 2002). This is because material properties are specified by the four-dimensional *bidirectional reflectance distribution function* (BRDF), which depends on the local light and viewing directions; the Phong model is an example of a BRDF. If spatial variation across a surface is to be captured, this function (which is the bidirectional texture function we mentioned earlier [Dana et al. 1999]) becomes six-dimensional. Obviously, the acquisition of such a high-dimensional function requires taking many samples, resulting in large acquisition times (and storage requirements), which is the reason why BTF measurements are not readily available.

The benefit of capturing all this data is that BTFs generate very realistic renderings of materials, because they capture not only the local reflectance properties for each point on a surface, but also all nonlocal effects, such as self-shadowing, masking, and interreflections within a material, as seen in Figure 11-1.

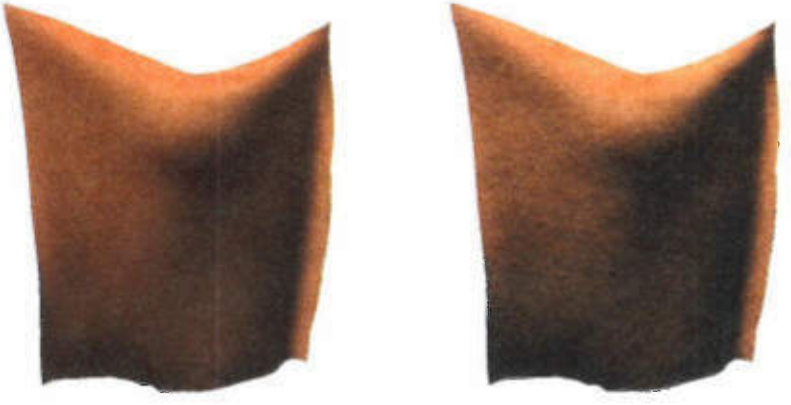


Figure 11-1. Comparing Simple Diffuse Texture Mapping and Our Method
(a) Simple diffuse texture map. (b) Our BTF method, preserving detail.

NVIDIA Ex1023
Page Q-221

Instead of acquiring a full BTF and rendering with that, we show that it is sufficient to acquire only some slices of the BTF. This allows for a simple acquisition setup (which can be done manually) and an even simpler rendering method, which runs in real time on GPUs even a few years old. An artist can also easily manipulate an acquired material, for example, making it more or less specular by manipulating a few parameters. To make this work, we assume that the captured material patch is isotropic (that is, its appearance does not depend on its orientation) and has only small-scale features (that is, no large self-shadowing and masking effects). Later on, we discuss in more detail why this is necessary.

11.2 Acquisition

The acquisition and preprocessing steps are very simple, resulting in a fast acquisition procedure.

11.2.1 Setup and Acquisition

The setup is illustrated in the top of Figure 11-2. A camera is looking orthographically at the material to be acquired. The material is assumed to be planar, so it is best to place the material patch on a flat surface. We then illuminate the surface with a parallel light source (a distant point light source works fine) at different incident elevation angles. The light source is moved only along an arc; that is, it is not rotated around the surface normal. With the light source moving along an arc, the material reveals small self-shadowing effects, which are very characteristic of many materials.

The light source elevation angles are (roughly) spaced in 10-degree steps; that is, we acquire ten images for every material. For real-time rendering, low-dynamic-range images work fine, as shown in Section 11.4; however, high-dynamic-range images can also be acquired.

In Figure 11-2, you can see the acquired images for a few different light source angles. We acquire all images manually, but even then the acquisition of a material takes only a few minutes, a big advantage over other methods (Dana et al. 1999, Matusik et al. 2002, Sattler et al. 2003).

11.2.2 Assembling the Shading Map

We now assemble our acquired images as a stack of images. To this end, we first compute the average radiance (or intensity) a_k of each acquired image k. Because we want to index efficiently into this stack of images later on, it is useful to resample the stack so

NVIDIA Ex1023
Page Q-222

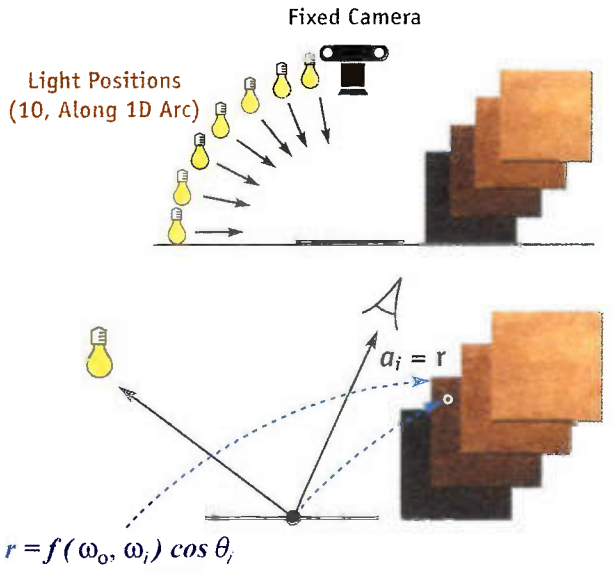


Figure 11-2. An Overview of the Capture Methodology
Top: We capture images for different light directions and for a fixed orthogonal view, resulting in the shading map. Bottom: For each image, we compute the average reflected radiance (that is, the average intensity). During rendering, we compute the value r based on some user-defined lighting model (such as Phong) and use this value to do a per-fragment lookup into our stack of images according to the average intensity. Finally, we scale the value with the intensity of the light source.

that the average radiance increases linearly. This is done by first computing what the average radiance for slice i should be, where i refers to a slice in the resampled stack:

$$r_i = \frac{i \times a_{\max}}{N-1}, \qquad i = 0..N-1 \tag{1}$$

Here r_i denotes the desired radiance for slice i, $a_{\max}$ is the maximum acquired average radiance, and N is the number of slices. For each resampled slice i we now take the two original images whose average radiances a_{k1} and a_{k2} are above and below the desired radiance r_i and linearly interpolate between those two images accordingly. The resulting stack of new images is called the *shading map*, as shown in Figure 11-3. From now on, the values stored in the shading map are interpreted as BRDF times cosine value, as explained in the next section.

NVIDIA Ex1023
Page Q-223

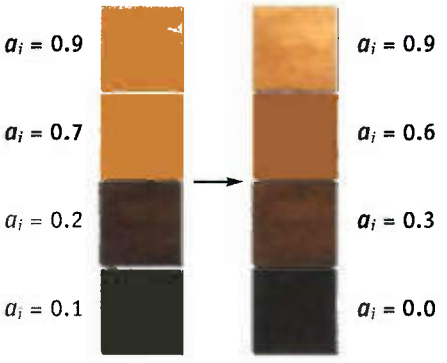


Figure 11-3. Creating a Shading Map
The input images are resampled and form the shading map, in which the average intensity increases linearly for every image.

11.3 Rendering

As stated before, rendering with our data is fairly simple. First we give a high-level description and then go into more details.

Consider shading a fragment with an ordinary lighting model—such as the Phong model—and a point light source. In that case, we compute the amount of reflected light by evaluating the lighting model with the current view and light direction and a set of parameters such as diffuseness, specularity, and so on. This value gives us a rough idea of how bright the pixel should be. Now that we have figured out its brightness, we can use our stack of images to shade it better. We select the image in our stack that has the same average brightness as our pixel. We now look up the appropriate texel from that image and use this value instead. This will introduce the fine detail such as self-shadowing, which we have captured in our image stack, and make the rendering much more realistic.

11.3.1 Detailed Algorithm

Here we describe the computation necessary for one fragment only; obviously, it has to be repeated for every fragment. We assume that for a fragment, we are given local view and light directions (ω_o, ω_i), as well as a set of texture coordinates (s, t).

First, we compute the value r by evaluating a lighting model, that is, the BRDF f_r (which can be any BRDF model) multiplied by the cosine of the incident angle:

NVIDIA Ex1023
Page Q-224

$$r(\omega_o, \omega_i) = \sigma f_r(\omega_o, \omega_i) \cos\theta_i.$$

The value σ is used to fine-tune the shade of the material. It is needed if the acquisition is not done with a unit light source or if only low-dynamic-range images are captured.

For example, for the simple Phong model, this computation boils down to the following formula:

$$r(\omega_o, \omega_i) = \sigma\left(k_d \cos\theta_i + k_s\left(\omega_o \times \omega_{i,\,reflected}\right)^E\right),$$

where k_d and k_s are the diffuse and specular coefficients, respectively, and E is the specular exponent.

Now we compute the amount of reflected light by multiplying the intensity of the incident light with the value $S(r)$ from our shading map:

$$L_0(\omega_o) = S\left(r(\omega_o, \omega_i)\right) I(\omega_i),$$

where $I(\omega_i)$ is the intensity of the light source (coming from ω_i). Using the value $r(\omega_o, \omega_i)$, we do a lookup into our shading map using the operator $S()$. It performs the lookup with the coordinates $(s, t, r/a_{\max} \times (N-1))$, where (s, t) signifies the position in the map, and $r/a_{\max} \times (N-1)$ is the layer to be chosen (as before, $a_{\max}$ is the maximum acquired value and N is the number of slices). Trilinear filtering of the shading map samples should be used to avoid blockiness and contouring artifacts.

One special case needs to be treated. If the required value r is higher than the maximum average value $a_{\max}$ in our shading map, we can either clamp the result at $a_{\max}$ or scale the brightest image in our shading map to the desired level r_i. The smallest average radiance r_0 is always zero (see Equation 1), and no special treatment is required.

11.3.2 Real-Time Rendering

The algorithm for real-time rendering using graphics hardware is straightforward. We load the shading map as a 3D (volume) texture onto the graphics hardware. In a vertex shader, we compute $r(\omega_o, \omega_i)$ (for a single-point light source). The result of the vertex shader is a set of texture coordinates $(s, t, r/a_{\max})$, which is then used to look up into the shading map. A fragment shader performs the multiplication with the intensity of the light source. See Listing 11-1 for a vertex and fragment shader performing this operation.

NVIDIA Ex1023
Page Q-225

Newer graphics hardware supports dependent texture lookups in the fragment shaders. With this feature, it is possible to compute the BRDF (times the cosine) on a per-fragment basis and then do a dependent texture lookup into the shading map, which avoids undersampling artifacts that may occur when using a vertex shader.

Listing 11-1. The Vertex and Pixel Shaders Needed for Rendering
In this example, we used the enhanced Phong shading model.

```
// Vertex Shader
void btfv( float4 position : POSITION,
           float3 normal   : NORMAL,
           float3 tex      : TEXCOORD0,
       out float4 HPOS     : POSITION,
       out float3 TEX      : TEXCOORD0,
       out float4 COL0     : TEXCOORD1,
          uniform float4x4 mvp,
          uniform float3 lightpos, uniform float3 eye,
          uniform float lightintensity, uniform float Ka,
          uniform float Kd, uniform float Ks, uniform float N )
{
  // compute light direction
  float3 L = lightpos - (float3)position;
  L = normalize( L );

  // compute reflected direction
  float3 R = (2.0f * dot(eye, normal) * normal - eye);

  // lighting (enhanced, physically correct Phong model)
  const float pi = 3.14159265;
  float costh = dot(normal, L);
  float specular = pow(dot(R, L), N);
  float intensity = Kd/pi + Ks * (N + 2.0) * specular/(2.0 * pi);
  intensity *= costh; // * cos(th)

  // check boundaries
  if ( diffuse < 0.0 || intensity < 0.0 ) intensity = 0.0;
  if ( intensity > 1.0 ) intensity = 1.0;

  // add ambient
  intensity += Ka;

  // set coordinates for lookup into volume
  TEX.z = intensity;  // based on lighting, a_max assumed to be 1
  TEX.xy = tex.xy;    // copy the 2D part from application
```


NVIDIA Ex1023
Page Q-226

Listing 11-1 *(continued)*. The Vertex and Pixel Shaders Needed for Rendering

```
  // set light source intensity for scaling in fragment shader
  COL0.rgb = lightintensity;

  // transform position
  HPOS = mul(mvp, position);
}


// Fragment Shader
float3 btfp( float3 tc    : TEXCOORD0,
             float3 color : TEXCOORD1,
             uniform sampler3D material : TEXUNIT0) : COLOR
{
  float3 r = (float3)tex3D( material, tc ).xyz; // lookup into volume
  return r * color;                             // scale by intensity
}
```

11.4 Results

We will show several renderings with acquired materials. All materials were acquired using ten different light source positions. The images were cropped to 1024×1024 and then downscaled to 512×512. In order to be compatible with graphics hardware, which usually requires power-of-two texture sizes, we resampled these ten images down to eight images during linearization. The linearization takes about 5 seconds for each set of images. For all renderings, we have used the Phong model.

In Figure 11-4, you can see different renderings done with our real-time implementation. The materials shown are jeans (Stanford bunny and pants), suede (Stanford bunny), leather (cat), pressed wood (head), wax (head), nylon (pants), cloth (piece of cloth), and wool (piece of knit cloth).

Figure 11-5 shows a rendering of a sweater made of wool. The left image was done with a full BTF (6,500 images, compressed using principal components analysis with 16 components). The right image was done with our technique. Differences are visible mainly at grazing angles, where masking effects become important, which cannot be reproduced by our technique. This also results in slight color differences (although some color difference is also caused by the compression). Considering that wool is one of the materials that violates our initial assumptions (it has a larger-scale structure and is anisotropic), our method does fairly well.

NVIDIA Ex1023
Page Q-227

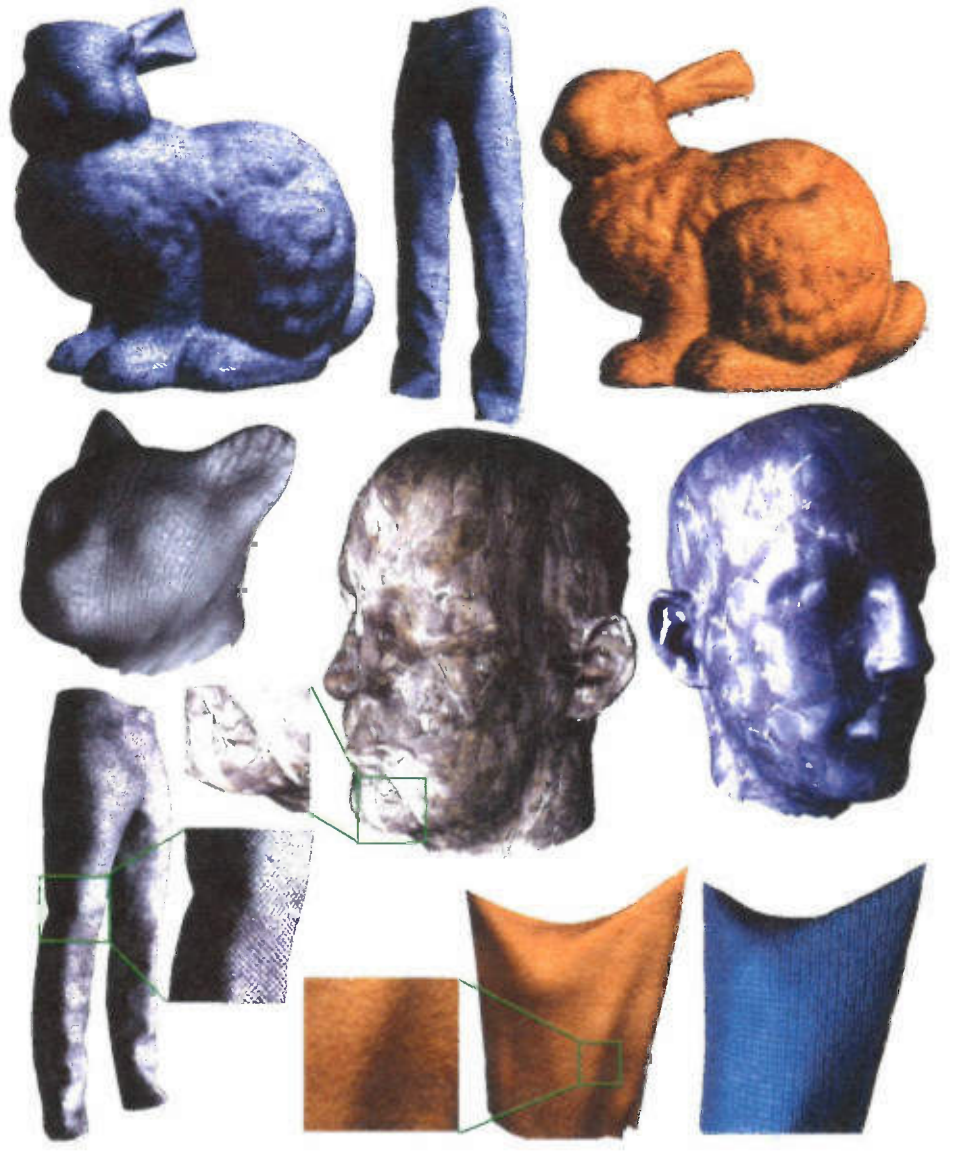


Figure 11-4. Different Models and Materials
The materials shown are jeans, suede, leather, pressed wood, wax, nylon, cloth, and wool. All models are rendered in real time.

NVIDIA Ex1023
Page Q-228



(a) (b)

Figure 11-5. A Sweater Made of Wool
(a) Full BTF. (b) New method.

11.4.1 Discussion

Why is it sufficient to acquire images for a fixed view and only several incident illumination directions? As stated earlier, we are acquiring only the change in illumination due to a material's surface structure, and for this purpose our method suffices. The artist supplies the actual BRDF.

We believe that this technique works so well because a human observer expects a certain kind of variation in materials. Judging from the results, this variation is captured by our technique and gives a good impression of the actual materials.

Generally speaking, materials with small-scale variations, such as the cloth, suede, jeans, and other materials shown, work best. Materials with spatially varying reflectance properties (such as different specularities, and so on) are captured well, too, as can be seen in the candle wax and pressed wood in Figure 11-4.

For certain materials, though, this technique works less well. For example, the acquisition of highly specular materials is problematic, mainly because of our use of a point light source as an approximation to a parallel light source. Our method assumes materials with a fine-scale structure. This is because our technique models variation with the

NVIDIA Ex1023
Page Q-229

incidence angle only, not with the azimuth. Therefore, shadows from larger structures cannot be captured or represented well, because they always appear in the same direction. Furthermore, the materials are assumed to be isotropic.

Even for materials violating these assumptions, our technique might still be suitable for a fast preview, because it is much better than regular texturing, as Figure 11-1 shows.

11.5 Conclusion

We have presented a method that enables acquisition and rendering of complex spatially varying materials using only a few images. This empirical method does not capture the actual BRDF but only how illumination changes due to fine surface structure; the BRDF can be adapted later. Real-time rendering using the technique can be easily implemented.

11.6 References

Dana, K., B. van Ginneken, S. Nayar, and J. Koenderink. 1999. "Reflectance and Texture of Real-World Surfaces." *ACM Transactions on Graphics* 18(1), January 1999, pp. 1–34.

Kautz, J., M. Sattler, R. Sarlette, R. Klein, and H.-P. Seidel. 2004. "Decoupling BRDFs from Surface Mesostructures." In *Proceedings of Graphics Interface 2004*, May 2004, pp. 177–184.

Lensch, H., J. Kautz, M. Goesele, W. Heidrich, and H.-P. Seidel. 2001. "Image-Based Reconstruction of Spatially Varying Materials." In *12th Eurographics Workshop on Rendering*, June 2001, pp. 103–114.

Matusik, W., H. Pfister, A. Ngan, P. Beardsley, and L. McMillan. 2002. "Image-Based 3D Photography Using Opacity Hulls." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, July 2002, pp. 427–437.

Sattler, M., R. Sarlette, and R. Klein. 2003. "Efficient and Realistic Visualization of Cloth." In *Eurographics Symposium on Rendering 2003*, June 2003, pp. 167–178.

NVIDIA Ex1023
Page Q-230



Chapter 12

Tile-Based Texture Mapping

Li-Yi Wei
NVIDIA Corporation

Many graphics applications such as games and simulations frequently use large textures for walls, floors, or terrain. There are several issues with large textures. First, they can consume significant storage, in either disk space, system memory, or graphics memory. Second, they can consume significant bandwidth, during either initial texture loading or rendering. Third, most graphics cards impose an upper limit on individual texture sizes. For example, the GeForce 6800 has an upper resolution limit of 4,096×4,096 for RGBA8 textures, and many other GPUs have stricter limitations.

One possible solution to address these problems is texture compression. However, existing GPU texture-compression techniques such as DXT have a limited compression ratio. In addition, texture compression cannot handle the problem of texture size limit.

An alternative approach is texture tiling, as shown in Figure 12-1. For example, if we have a large wall or floor consisting of repeating tiles, then obviously we don't have to store all the tiles. Instead, we could have just one tile and repeat it on the wall. For more complex patterns, we can dice up the wall or floor into smaller polygons and apply different texture tiles or texture coordinate transformations for each polygon. The advantage of this approach is that we can achieve infinite compression ratio in theory, because we can produce an arbitrarily large output from a few tiles. The disadvantage, however, lies in the complication of application code and data.

Introduction

NVIDIA Ex1023
Page Q-231

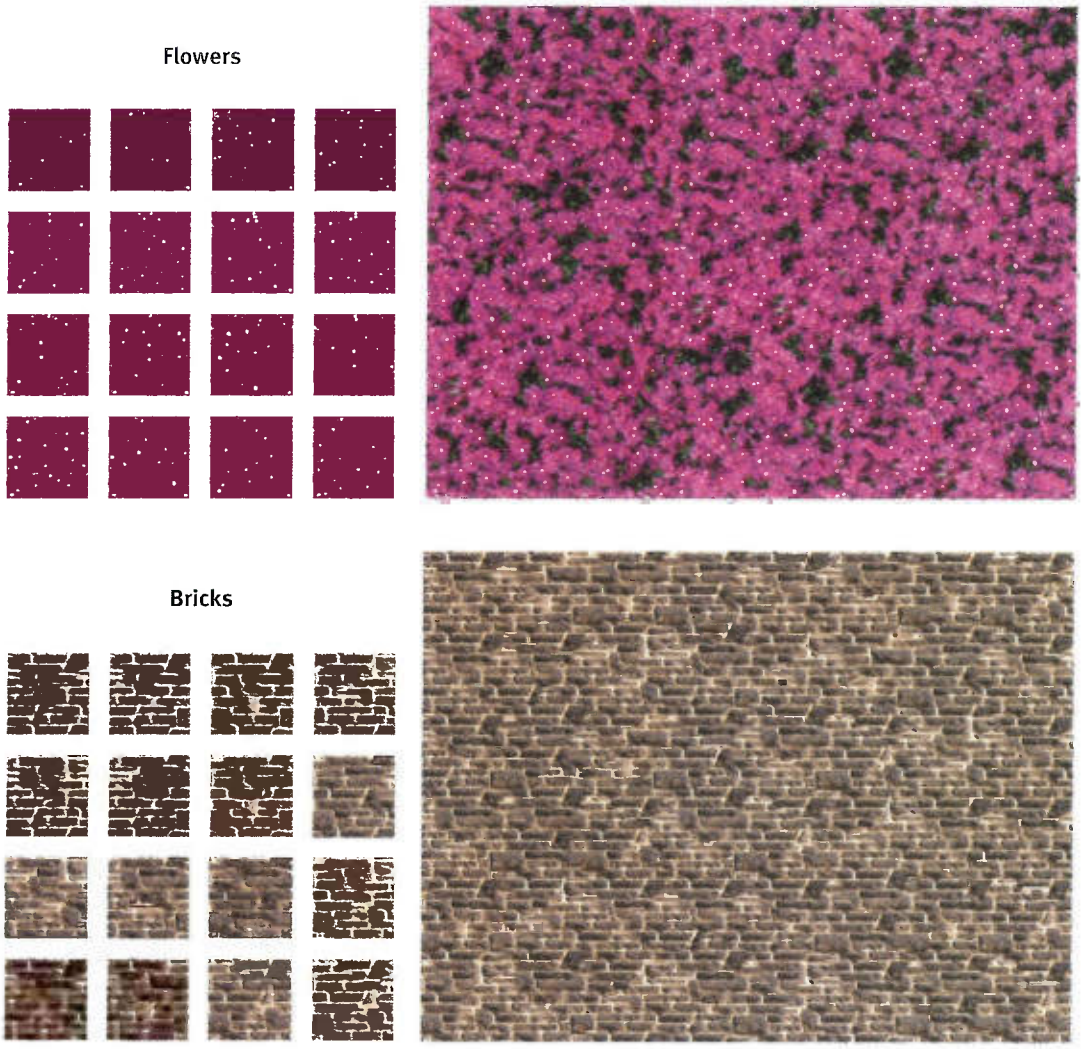


Figure 12-1. A Tile-Based Texture Map
Given a small set of input texture tiles (left), our system provides a large virtual texture map (right) without storing the entire texture. Our approach supports native hardware texture filtering and does not require modifying the geometry or texture coordinates of the application.

Our goal is to present a tile-based texture-mapping scheme that is transparent to the application. Specifically, we do not require changing the original geometry or texture coordinates in order to use texture tiles. Instead, we implement the texture tiling as a fragment program that handles the details of accessing and filtering the tiled texture.

NVIDIA Ex1023
Page Q-232

12.1 Our Approach

We represent a large texture as a small set of tiles and assemble the tiles into a large virtual texture on the fly for answering texture requests. To minimize visual repetition, we assemble the tiles nonperiodically. There is a huge literature on nonperiodic tiling, but for our method, we chose Wang tiles for their simplicity (Cohen et al. 2003). Wang tiles are square tiles in which each tile edge is encoded with a color. A valid tiling requires all shared edges between tiles to have matching colors. When a set of Wang tiles is filled with texture patterns that are continuous across matching tile edges, a valid tiling from such a set can produce an arbitrarily large texture without pattern discontinuity. A sample Wang tiling is shown in Figure 12-2.

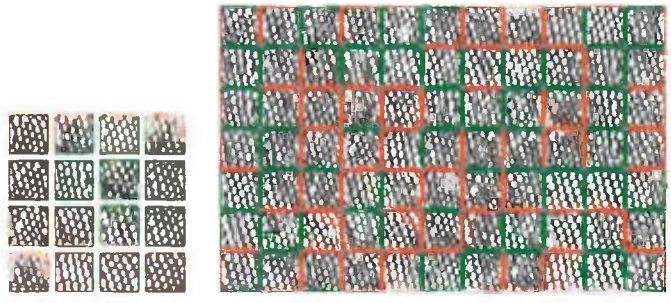


Figure 12-2. Texture Tiling Using Wang Tiles
The input tiles are shown on the left, and a sample tiling is shown on the right. In this example, we have two vertical and horizontal tile edge colors, represented as red and green, respectively. In the output, note that adjacent tiles share identical edge colors.

We extend the Wang tiling approach to a GPU-friendly implementation. Given a texture request (s, t), we first figure out on which tile it lands, based on the (s, t) location. We then fetch the corresponding texels from that input texture tile. Figure 12-3 illustrates the procedure. We carefully pack the texture tiles into a single texture map so that we can perform texture fetching and filtering on this single texture via texturing hardware.

12.2 Texture Tile Construction

There are several methods to construct the Wang tile set. One possibility is to draw the tiles manually; all we need to do is to ensure that the drawings are continuous across tile boundaries while producing sufficiently interesting tiling results. Figures 12-5 and 12-6, later in this chapter, show a tile set drawn by me and the tiling results, respectively. This manual approach works only for line art.

NVIDIA Ex1023
Page Q-233

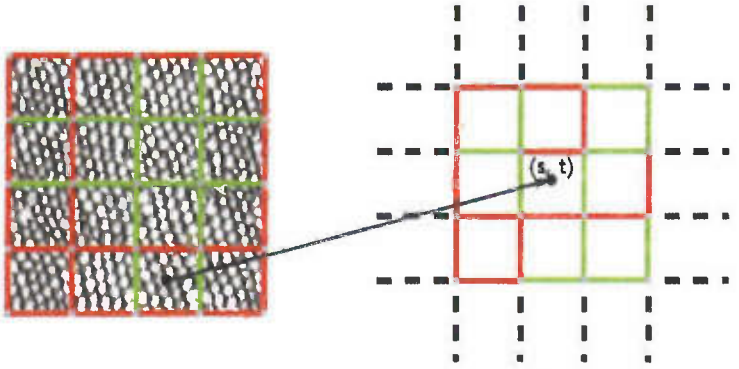


Figure 12-3. An Overview of Tile-Based Texture Mapping
Left: The packed input tiles. Right: The output virtual texture. Given a texture request (s, t), our algorithm first determines on which tile the request lands. The algorithm then fetches the corresponding texels from that input tile.

For natural texture patterns such as those from photographs, there are algorithmic ways to build the tile set, as described in Cohen et al. 2003, Section 3.2. We have found that this approach works well for many natural texture patterns, and we provide the source code on the accompanying CD. The sample tiles previously shown in Figure 12-1 were generated with this approach.

There are additional methods for constructing the tiles; we refer readers to Cohen et al. 2003 for more details.

12.3 Texture Tile Packing

One common problem with previous texture tiling or atlas approaches is the filtering issue. Specifically, given a collection of tiles, how do we perform filtering for the output virtual texture, especially when the sample is on the tile border and requires texels from multiple tiles for bilinear or trilinear filtering?

There are several possible solutions, among them performing texture filtering in the fragment program, or packing all tiles into a single texture and adding boundary padding between the tiles so that bilinear filtering would be correct. The first of these is computationally expensive, and the second consumes a large amount of texture memory.

NVIDIA Ex1023
Page Q-234

In our approach, we pack all the texture tiles into a single texture map without any boundary padding, in a way that makes it possible to perform the texture filtering directly via the texturing hardware without fragment program emulation. We achieve this by packing a set of Wang tiles into a single texture map, ensuring that any tile edges shared between two tiles have identical edge color encoding. Because the tiles are built with a continuous pattern across identical edge colors, bilinear filtering directly across adjacent tiles produces the correct result. It is possible to prove that, for a complete tile set (that is, with all possible edge color combinations), it is always possible to pack the tiles in such a way. In addition, each tile is used only once, so there is no wasted texture memory.

Here we describe how the packing can be done, without going into detailed mathematical proofs, which can be found in Wei 2004. For clarity, we begin with packing in one dimension. We assume that we have a set of tiles with only one horizontal edge color and that we would like to pack them horizontally. The packing can be achieved with the following equation:

$$TileIndex1D(e_1, e_2) = \begin{cases} 0, & e_1 = e_2 = 0 \\ e_1^2 + 2 \times e_2 - 1, & e_1 > e_2 > 0 \\ 2 \times e_1 + e_2^2, & e_2 > e_1 \geq 0 \\ (e_1 + 1)^2 - 2, & e_1 = e_2 > 0 \\ (e_1 + 1)^2 - 1, & e_1 > e_2 = 0 \end{cases}. \quad (1)$$

Basically, given a tile with left edge e_1 and right edge e_2, where e_1 and e_2 are integer indices starting from 0, this function computes the horizontal location of this tile within the packing. A sample 1D packing with three vertical edge colors can be found in Figure 12-4a.

The packing in 2D can be performed by computing vertical and horizontal indices separately using Equation 1. In other words, the horizontal tile location is computed from the left and right edges only, and the vertical tile location is computed from the top and bottom edges only. A 2D sample packing with three vertical and horizontal edge colors can be found in Figure 12-4b. The computation can be expressed in Cg-like code as shown in Listing 12-1.

NVIDIA Ex1023
Page Q-235

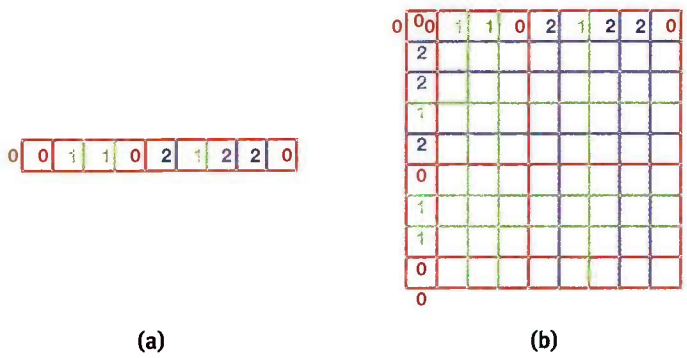


Figure 12-4. Examples of Tile Packing
(a) 1D packing with three colors. (b) 2D packing with three vertical and horizontal colors.

Listing 12-1. Code for Tile Packing

```
float TileIndex1D(const float e1, const float e2)
{
  float result;
  if (e1 < e2) result = (2 * e1 + e2 * e2);
  else if (e1 == e2)
    if (e1 > 0) result = ((e1 + 1) * (e1 + 1) - 2);
    else result = 0;
  else
    if (e2 > 0) result = (e1 * e1 + 2 * e2 - 1);
    else result = ((e1 + 1) * (e1 + 1) - 1);
  return result;
}

float2 TileIndex2D(const float4 e)
{
  float2 result;
  // orthogonally compute the horizontal and vertical locations
  // the x, y, z, w components of e store
  // the south, east, north, and west edge coding
  result.x = TileIndex1D(e.w, e.y);
  result.y = TileIndex1D(e.x, e.z);
  return result;
}
```


NVIDIA Ex1023
Page Q-236

12.4 Texture Tile Mapping

Given that the output virtual texture is composed from the tile set, we need to know which tile is used for each region of the texture. Specifically, for each texture coordinate (s, t), we need to be able to determine efficiently on which tile it lands.

There are two possible solutions for mapping tiles into the output texture. The first approach is simply to precompute the mapping and store the result into an index texture map. For example, if the output is composed of 128×128 tiles, then we use a 128×128 index texture to store the (*row*, *column*) location of each tile as in the input pack. Because the index texture map will have limited size, this approach isn't able to handle arbitrarily large textures. However, it is more than enough for most applications, and it is very efficient. Note that this index map does not require a mipmap pyramid because we do not require mipmap filtering for the (*row*, *column*) locations.

The second approach does not need this index texture at all; instead, the tile for each output region can be computed on the fly via a special fragment program with hashing. This approach has the advantage that it does not consume an extra texture map, and it does not suffer from the restriction on index texture map size, as in the first approach. However, in our current implementation, we have found the second approach too slow for real-time applications; so for the rest of this chapter, we discuss only the first approach. However, the second approach might be faster in future-generation GPUs, and we refer interested readers to Wei 2004 for more details.

Given that we have a precomputed index texture, the process for our tile-based texture mapping is actually pretty simple, and is best explained by the actual Cg program, as shown in Listing 12-2. Given the input coordinate, we first scale it with respect to the number of tiles in the output. We then use this scaled value to fetch the index texture, which will tell us which input tile to use. Finally, we use the fractional part of the scaled texture coordinate to fetch the final texel from the selected input tile. Note that in the final texture fetch, we need to specify the derivatives `ddx` and `ddy` to ensure the correct level-of-detail computation.

Listing 12-2. Code for Tile Mapping

```
struct FragmentInput
{
  float4 tex : TEX0;
  float4 col : COL0;
};
```


NVIDIA Ex1023
Page Q-237

Listing 12-2 *(continued)*. Code for Tile Mapping

```
struct FragmentOutput
{
  float4 col : COL;
};

float2 mod(const float2 a, const float2 b)
{
  // emulate integer mod operations in a float-only environment
  return floor(frac(a/b) * b);
}

FragmentOutput fragment(FragmentInput input,
  uniform sampler2D tilesTexture,
  uniform samplerRECT indexTexture)
{
  FragmentOutput output;

  // osizeh and osizev are the number of output tiles
  // they are compile-time constants
  float2 mappingScale = float2(osizeh, osizev);
  // the scaling is required for RECT textures
  // we will also need its fractional part in the final tex read
  float2 mappingAddress = input.tex.xy * mappingScale;
  // figure out which input tile to use
  float4 whichTile = texRECT(indexTexture,
                             mod(mappingAddress, mappingScale));

  // isizeh and isizev are the number of input tiles
  // they are compile-time constants
  float2 tileScale = float2(isizeh, isizev);
  // this is to ensure the correct input mipmap level is accessed
  float2 tileScaledTex = input.tex.xy * float2(osizeh/isizeh,
                                               osizev/isizev);

  // fetch the final texel color
  // note we need to specify ddx/ddy explicitly
  output.col = tex2D(tilesTexture,
                     (whichTile.xy + frac(mappingAddress))/tileScale,
                     ddx(tileScaledTex), ddy(tileScaledTex));
  return output;
}
```


NVIDIA Ex1023
Page Q-238

12.5 Mipmap Issues

An inherent limitation of any tile-based texturing system is that mipmaps at lower resolutions are incorrect. There are actually several issues with mipmaps.

First, the maximum number of mipmap levels is constrained by the input tiles, which are of much smaller size than the output. For example, assuming we pack 4×4 tiles each with 32×32 pixels into a single 128×128 texture map, then the maximum number of mipmap levels is 8. Obviously, for any output texture larger than 128×128, we will not be able to access the coarser resolutions. In our experience, this is usually not a problem, because the tiles would reduce to a homogeneous void at lower resolutions anyway.

We also need to be careful when constructing the mipmap texture for the packed input. If you just use the automatic mipmap generation, chances are that the tiles will not be legal Wang tiles at lower mipmap resolutions. This can produce annoying aliasing artifacts when filtering across the tile boundary. Figure 12-5 demonstrates such a case.

As shown in Figure 12-5, in the output we have two tiles, ▮ and ▮, adjacent to each other. If we have a request (s, t) that lands on the right edge of ▮, and assuming the filter kernel is sufficiently large, as can happen in lower-resolution mipmap levels, then a correct filtering should involve a majority of the tile ▮. Unfortunately, because we always perform filtering in the input packing, the filtering would involve a majority of ◗ instead, and this is obviously incorrect. A more serious problem occurs when the filter footprint just crosses the boundary between ▮ and ▮. In this case, the filtering result would change discontinuously in the input, with an abrupt switch from filtering tiles ▮ and ◗ to ▰ and ▮. And this artifact could exhibit itself both spatially and temporally.

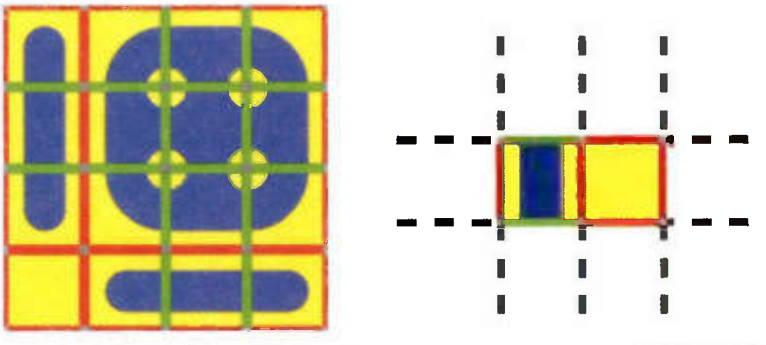


Figure 12-5. Artifacts Caused by Filtering
Involving tiles ▮, ▮, ◗, and ▰. Left: Packed input tiles. Right: Output virtual texture.

NVIDIA Ex1023
Page Q-239

To remove these artifacts, we have to sanitize the tiles at all lower-resolution mipmap levels to ensure that they are legal Wang tiles. We achieve this by averaging all tile edges with the same color ID across all tiles. This does not produce a theoretically correct result (unless you store the entire lower-resolution mipmap levels for the output), but it at least eliminates popping visual artifacts. An implementation of this algorithm is available on the accompanying CD.

Figure 12-6 demonstrates the sanitization effects on the rendering. Without tile correction, the rendering would contain noticeable discontinuity, as shown in case (a). With correction on tile edges, the results look much better, as shown in case (b). A more correct approach would be to sanitize both edges and corners, but we have found that this usually produces over-blurry results, as case (c) demonstrates. In our experience, because most textures are homogeneous at lower mipmap resolutions, it is usually sufficient to sanitize edges only.

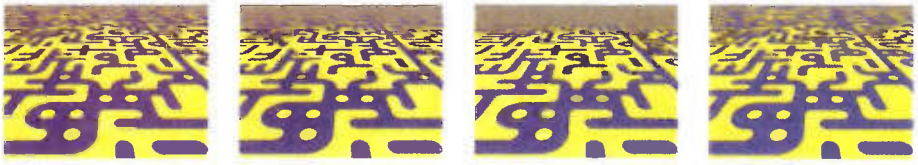


Figure 12-6. Filtering Results at Lower-Resolution Mipmap Levels
(a) Incorrect lower-resolution tiles. (b) Corrected tile edges only. (c) Corrected tile edges and corners. (d) Traditional texture mapping.

12.6 Conclusion

We have presented a tile-based texture-mapping system that generates a large virtual texture from a small set of tiles. We have described a Cg program implementation that does not require changing geometry or texture coordinates in the application. Our implementation requires two texture reads to emulate one large virtual texture, and is roughly 50 to 25 percent slower than a single-texture program (as measured on a GeForce 6800 GT, depending on various factors such as filtering mode and original program complexity). However, the speed is still fast enough for most applications, and the savings in texture memory is substantial. There are quality limitations of our approach at lower-resolution mipmap levels, but as long as the tile set is built correctly, the result looks reasonable despite being theoretically incorrect.

NVIDIA Ex1023
Page Q-240

12.7 References

Cohen, Michael F., Jonathan Shade, Stefan Hiller, and Oliver Deussen. 2003. "Wang Tiles for Image and Texture Generation." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)* 22(3), pp. 287–294.

Glanville, R. Steven. 2004. "Texture Bombing." In *GPU Gems*, edited by Randima Fernando, pp. 323–338. Addison-Wesley.

Lefebvre, Sylvain, and Fabrice Neyret. 2003. "Pattern Based Procedural Textures." In *Proceedings of SIGGRAPH 2003 Symposium on Interactive 3D Graphics*, pp. 203–212.

Losasso, Frank, and Hugues Hoppe. 2004. "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids." *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2004)* 23(3), pp. 769–776.

Tanner, Christopher, Christopher Migdal, and Michael Jones. 1998. "The Clipmap: A Virtual Mipmap." In *Proceedings of SIGGRAPH 98*, pp. 151–158.

Wei, Li-Yi. 2004. "Tile-Based Texture Mapping on Graphics Hardware." In *Proceedings of Graphics Hardware 2004*, pp. 55–63.

I would like to thank the people who helped me in my original paper (Wei 2004), Hugues Hoppe for pointing out the mipmap issue, Shaun Ho for recommending the submission of this technique for this book, and my NVIDIA colleagues for their help and support.

NVIDIA Ex1023
Page Q-241



Chapter 13

Implementing the mental images Phenomena Renderer on the GPU

Martin-Karl Lefrançois
mental images

13.1 Introduction

mental images' rendering software mental ray is the rendering component of many leading 3D content-creation tools, including industrial CAD, product design, and architectural design software packages. mental ray is widely used for the creation of visual effects and feature animation films, as well as for high-quality visualization and lighting simulation in industrial and architectural design. Now that modern GPUs can execute complex programs to compute the color of each shaded pixel, there is an opportunity to apply this power to rendering complex scenes that previously could be handled only by a CPU-based renderer. Given the substantial floating-point computational power available on GPUs, doing these computations on the GPU can be much faster than on the CPU. This chapter describes some of the work that mental images has done to add support for GPU-based rendering to mental ray.

Because the per-fragment mathematical computation done by a GPU can now be identical to the computation done by a software renderer, it is possible to use the GPU to accelerate rendering of the final image. Although some techniques, such as software ray tracing, cannot yet be replaced with an efficient GPU-based solution due to the limitations of current hardware, there are many scenes that do not require these techniques. Such scenes can be entirely rendered by the GPU. In some cases, we still have to combine

NVIDIA Ex1023
Page Q-242

software and hardware rendering to achieve the final image, but hardware technology evolves so fast that we will see more and more of the work shifting to the GPU.

One of the main difficulties with rendering high-quality imagery on the GPU is making it possible to use the same shaders on the GPU as on the CPU. Production companies use a carefully developed set of shaders that they combine to create the final effect they want. With most GPU programming languages, it is difficult to efficiently combine shaders, because the languages are designed for monolithic programs that implement a complete effect. However, the release of Cg 1.2 was a breakthrough for the creation of complex shaders on the GPU: with the addition of shader interfaces and unsized arrays to the language, it became possible to combine shaders at runtime to generate GPU programs that render the desired final effect.

In this chapter, we briefly review mental ray's 3.3/3.4 shading architecture and describe how we convert combinations of mental ray shaders to GPU programs. The techniques we use provide a powerful and flexible means of creating complex visual effects. We are now able to accelerate rendering of final images with the GPU without sacrificing quality or visual richness.

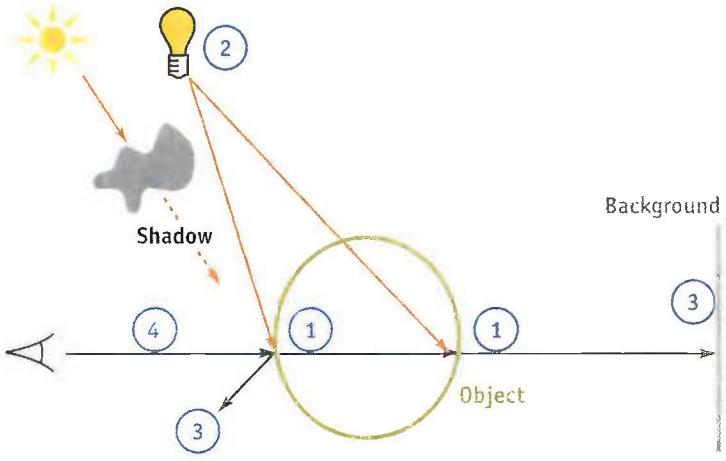
13.2 Shaders and Phenomena

For high-quality offline rendering, the concept of a "shader" is a slightly different one than for interactive rendering on GPUs. On the GPU, vertex shaders and fragment shaders operate on the input geometry to compute the color of each pixel. In mental ray, shaders are plug-in modules that are used in materials, light sources, cameras, and other elements to control a wide range of effects, from surface material, volume, and camera lens properties to compositing. High-end visual effects and feature animation production companies keep separate effect components in separate shaders that are combined automatically at runtime; it often makes more sense to divide particular visual effects into smaller elements that separately represent the environment and what interacts with it. As an example, objects carry surface shaders that define the response of the surface to incident light (that is, the BRDF). Lights are separate scene elements that carry light shaders that define the light emission characteristics. Both need to be combined at runtime to perform shading. Additionally, there are volume shaders; lens shaders attached to cameras; texture, environment, and other shaders that all contribute to shading and must be collected to create the fragment program.

NVIDIA Ex1023
Page Q-243

When a software renderer renders a scene, different shaders are called in different stages of the pipeline. In the following example, the first shader to be called is the material shader, which defines the color of the illuminated surface. Then the light shaders of all of the lights illuminating this surface will be called in order to determine how much illumination they shine on the surface. Light shaders behave differently depending on whether they are point, spot, or area lights. If the surface is reflective, the environment shader may be called to add its contribution to the surface. If ray tracing is enabled, a ray might hit another object, whose material shader will then be called. If the object is transparent, the material and light shaders will be called again until full opacity is achieved. In addition to all those shaders, volume shaders, such as fog, can affect the final image. The volume shader, if attached to the camera, will be called for every sample. Figure 13-1 illustrates these basic concepts.

The mental ray renderer further extends this idea to include Phenomenon components (Driemeyer 2005). These look like shaders when they are applied to the scene, but their functionality is implemented in a different way. Whereas shaders usually function as standalone components, Phenomena encapsulate sets of subshader nodes connected to form graphs, as well as associated rendering options, geometry, material, volumes, and other elements that are automatically integrated at appropriate points in the scene. (Phenomena are a superset of Maya's Hypershade graphs, for example.) Both shaders



Shaders: (1) Material, (2) Light, (3) Environment, (4) Volume

Figure 13-1. Shaders in a Scene

NVIDIA Ex1023
Page Q-244

and Phenomena have *interface parameters* that feed data into them and *result parameters* that return the result of their evaluation. In the case of Phenomena, the result parameters of the Phenomenon are taken from a specific shader that is called the *main root shader* of the Phenomenon.

Once defined, the shader can be used to control the surface properties of a material. Shaders and Phenomena can be connected to form shader graphs. In mental ray, users can assign the result parameter of another shader to an interface parameter.

Figure 13-2 shows an example of a Phenomenon. It uses three input parameter values to compute the result. These parameter values in turn may be computed by other shaders, may be constants set by the user, may be the results of texture lookups, and so on. Note that in this case, the final result value is computed by another "sub" Phenomenon in the blue box. This ability to "wire up" Phenomenon graphs in many different ways, using modular pieces, makes them a powerful paradigm for artists and designers.

In the schematic diagrams in this chapter, boxes represent shaders or Phenomena; their interface parameters (such as "ambient," "diffuse," or "specular") are on the left side and the result parameters are on the right side. Phenomenon instances are useful as prepackaged effects: a wood Phenomenon, for example, may have parameters that define colors and wood grain turbulence, and different predefined instances of it can supply parameter values to make it look like oak, redwood, birch, or other variations.

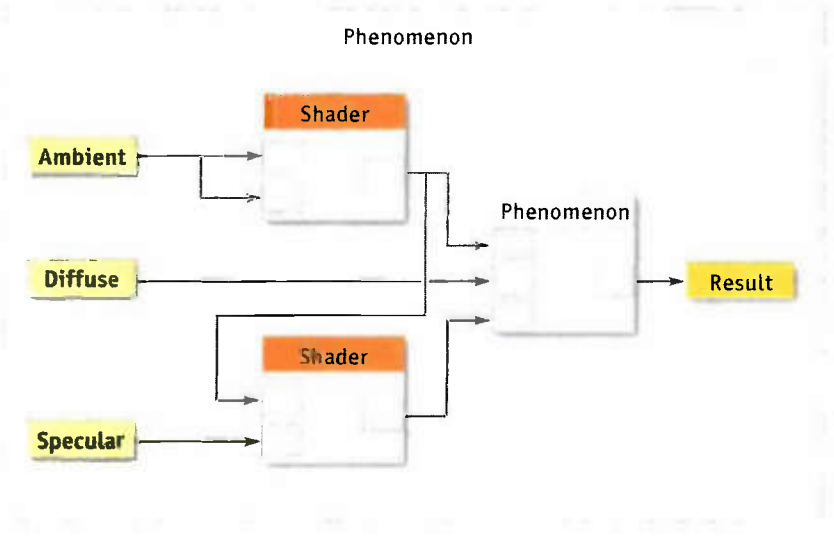


Figure 13-2. An Example of a Phenomenon

NVIDIA Ex1023
Page Q-245

When using shaders or Phenomena in a scene, you need to know three properties:

- The name of the shader or Phenomenon
- Its interface parameters
- Its result parameter

The shader name is identical to the name of the Cg structure type that implements it. The interface parameters are a set of named values passed to the shader or Phenomenon when it runs. For example, a shader implementing Phong shading could have three interface parameters of type `color` named `ambient`, `diffuse`, and `specular` and one of type `scalar` named `exponent`. Finally, the result parameters of a shader represent the values returned by the shaders. Most of the time, this is a single value of type `color`, but other types and structures of multiple values are also possible.

13.3 Implementing Phenomena Using Cg

Because GPUs do not support loading multiple shaders and then having one shader call another shader at runtime, mental ray must collect the Phenomenon—including all subshaders that feed its input parameters, root shaders, and assigned shaders (such as light shaders that illuminate the surface)—into a single fragment program that can be downloaded to the GPU. The shader interfaces feature introduced in Cg 1.2 (Pharr 2004) facilitates this process.

In our system, we have predefined an abstract Cg interface type for each of the parameter types that may be passed as input, or computed as output, by a shader component. Shaders and Phenomena are written to use these abstract interface types when declaring input and output parameters. At runtime, concrete instances of each subshader are created using the Cg runtime library, and the outputs of each internal subshader are connected to the proper inputs, as shown in Figure 13-3.

The rest of this section provides details on the GPU programs we use, the interface types we define, and how output parameters are connected to the inputs of other shaders to form graphs.

13.3.1 The Cg Vertex Program and the Varying Parameters

We have found that for the majority of effects we want to achieve, we can use the same vertex program. This vertex program primarily serves as a simple pass-through for the

NVIDIA Ex1023
Page Q-246

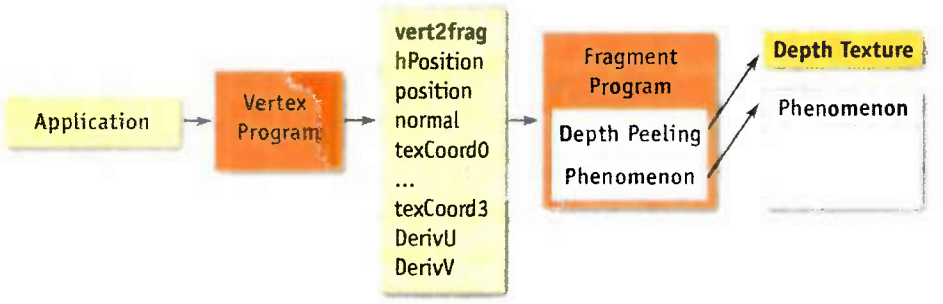


Figure 13-3. How a Phenomenon Fits into the Graphics Pipeline

data coming from the application. The most common varying parameters are transformed in the vertex shader and passed to the fragment shader.

In addition to the homogeneous position, the following varying parameters are sent to the fragment programs: the position, the normal, four texture coordinates, and the surface's parametric derivatives in U and V for bump mapping. All data, except texture coordinates, are transformed to camera space for simplicity. For example, the position in the fragment corresponds to the vector from the camera to the rendered pixel in eye space, and the length of the vector corresponds to the distance from the camera. In the fragment program, transformation matrices are available to convert from one space to another if needed. The varying parameters from the vertex program to the fragment program are stored in a structure called `vert2frag`, shown in Listing 13-1.

Listing 13-1. The Varying Parameters Sent to the Fragment Program

```
struct vert2frag
{
  float4 hPosition : POSITION;  // Homogeneous position
  float2 texCoord0 : TEXCOORD0; // Texture coordinates
  float2 texCoord1 : TEXCOORD1; // Texture coordinates
  float2 texCoord2 : TEXCOORD2; // Texture coordinates
  float2 texCoord3 : TEXCOORD3; // Texture coordinates
  float3 position  : TEXCOORD4; // Eye-space position
  float3 normal    : TEXCOORD5; // Eye-space normal
  float3 derivU    : TEXCOORD6; // Derivative U
  float3 derivV    : TEXCOORD7; // Derivative V
  float2 screen    : WPOS;      // Screen space
  float4 face      : COLOR0;    // 1 front, -1 back
};
```


NVIDIA Ex1023
Page Q-247

13.3.2 The `main()` Entry Point for Fragment Shaders

All fragment programs composed from component shaders by mental ray must start with an entry function. To be able to translate Phenomena to the GPU, we had to come up with a common main entry point. For all shader graphs, we create a `main()` function that calls the appropriate Phenomenon root shader, which triggers the evaluation of the entire graph of shaders.

This main entry point also handles some general operations required in all shaders. For example, it includes a test for discarding pixels nearer than the z-depth texture, which is used for order-independent transparency.

13.3.3 The General Shader Interfaces

Support for interfaces was introduced in Cg 1.2. These interfaces provide a way to abstract how member functions in structures are implemented. We use them heavily to implement the value returned to a parameter of a shader. To be able to connect the output of one shader with the input parameters of other shaders, we have to define general Cg interface types that are used by all shaders. These default Cg interface types and their implementation are automatically inserted in each program. For each *parameter type* found in the description of the shader, an equivalent interface exists. As an example, if an input parameter is of type `scalar` in the declaration of the shader, it needs to be declared `miiScalar` in the Cg implementation of the shader.

Table 13-1 shows the corresponding parameter and interface types.

Table 13-1. mental ray Basic Type and Corresponding Cg Interfaces

mental ray Type	Cg Interface Type	Return Value
`light`	`miiLight`	`misLightOut`
`boolean`	`miiBoolean`	`bool`
`integer`	`miiInteger`	`int`
`scalar`	`miiScalar`	`float`
`vector`	`miiVector`	`float3`
`color`	`miiColor`	`float4`
`color texture`	`miiColorTexture`	`float4`
`matrix`	`miiMatrix`	`float4x4`

NVIDIA Ex1023
Page Q-248

Each Cg interface type defines a single evaluation method, named `eval()`, that returns a value of the corresponding concrete type. For example:

```
// integer
interface miiInteger (
  int eval(vert2frag params);
);

// vector
interface miiVector (
  float3 eval(vert2frag params);
);
```

To write a shader that serves as a concrete implementation of an interface type, we simply define a struct that implements the appropriate `eval()` method. This allows us to "plug in" the struct wherever the corresponding interface type is called for.

As an example, the following case shows the implementation of `miiScalar`, in which we want a simple constant value to be assigned to an interface parameter. We create and connect a simple implementation of the interface, which causes a constant value to be returned by the `eval()` method:

```
// base scalar implementation
struct miScalar : miiScalar {
  float s;
  float eval(vert2frag params) {
    return s;
  }
};
```

13.3.4 Example of a Simple Shader

mental ray needs to know how shaders are defined to properly call them and assign values to the shader's parameter. Here is an example of a shader declaration as it can be found when parsing a mental ray scene:

```
declare shader
  color "mib_illum_phong" (
    color "ambience",
    color "ambient",
    color "diffuse",
    color "specular",
```


NVIDIA Ex1023
Page Q-249

```
    scalar "exponent",   # phong exponent
    integer "mode",      # light selection mode 0..2
    array light "lights"
  )
end declare
```

When writing fragment shaders, we follow a simple rule: in the Cg file implementing the shader, we define a structure that implements the interface corresponding to the return value of the shader as required by the declaration of the shader. The structure should also have the input parameters of the shader and an `eval()` method for the return interface.

The name of the structure is the same name found in the declaration of the shader. It must implement the return type interface also found in the shader declaration. All of the shader's required input parameters must be defined using their equivalent type interfaces. They also must have the same name as found in the shader declaration.

For a single shader, an instance of the shader will be created and connected to the main root of the Phenomenon. And when no shaders are attached to the input parameters, a simple interface that evaluates to a constant is connected, the variability of which is set to `CG_LITERAL` using the Cg runtime. Setting the variability to `CG_LITERAL` allows the Cg compiler to optimize the resulting fragment shader (by using constant folding) to a greater degree than if a uniform value were used.

As shown in Listing 13-2, the name of the structure is the same as the shader declaration. The structure implements the interface `miiColor`, which corresponds to the output parameter. Not all input parameters are needed; for example, the parameter `mode` is ignored in the Cg shader.

The graph in Figure 13-4 shows an example of the connection of values in the shader. Some parameters are omitted for clarity.

Listing 13-2. Cg Implementation of the Phong Illumination Model

```
struct mib_illum_phong : miiColor
{
  miiColor  ambience, ambient, diffuse, specular;
  miiScalar exponent;
  miiLight  lights[];

  // Implementation of miiColor
  float4 eval(vert2frag p) {
    float4 result;
```


NVIDIA Ex1023
Page Q-250

Listing 13-2 *(continued)*. Cg Implementation of the Phong Illumination Model

```
    float4 ldiffuse = diffuse.eval(p);
    float4 lspecular = specular.eval(p);
    float lexponent = exponent.eval(p)

    result = ambience.eval(p) * ambient.eval(p);

    // Material calculation for each light
    for ( int i = 0; i < lights.length; i++ ){
    // Light shader evaluation
      misLightOut light = lights[i].eval(p);

      // Lambert's cosine law
      float d = max(0, light.dot_nl);
      result += d * ldiffuse * light.color;

      // Phong's cosine power
      if (d > 0) {
        float s = mi_phong_specular(lexponent, light.dir,
                                    vdir, p.normal);
        result += s * lspecular * light.color;
      }
    }

    return result;
  }
};
```

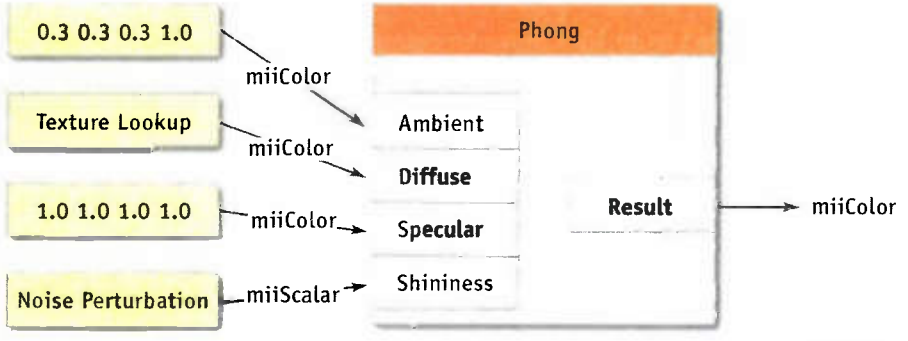


Figure 13-4. An Example of a Simple Shader

NVIDIA Ex1023
Page Q-251

13.3.5 Global State Variables

Shaders may need additional information that is not passed to them via interface parameters. For example, it might be important to know the current object being rendered or the resolution of the final image. Also, many shaders need to convert from one coordinate space to another, which requires access to various transformation matrices. Our system uses specific global state variables to provide these values. For example, when creating a program, our system iterates through its global parameters looking for any named "`miToWorld`". If the program contains this parameter name and if the type corresponds to `float4x4`, it is initialized with the appropriate transformation matrix. Table 13-2 shows the variables that can be used.

These state variables are created as shared parameters using the Cg runtime. When we find a known state parameter as a program's global parameter, we just connect it to the corresponding global uniform parameter. When the value changes during a frame, we only have to update the value of the shared parameter to have the value updated in each shader that has a parameter bound to it. This use of uniform values is illustrated in Figure 13-5.

13.3.6 Light Shaders

Light shaders implement the characteristics of a light source. For example, a spotlight shader would use the illumination direction to attenuate the amount of light emitted within a cone. A light shader is called whenever a material shader uses the light interface to evaluate a light. In production scenes, we often find a lot of different lights illuminating a model. In our implementation, all light shaders must return a structure containing the amount of contributed color from the light, the direction to the light, and the dot product of the light direction and the normal. Because surface shaders are written to call out to the abstract light interface to get these values back, it is easy to use any type of light that implements this interface with any type of surface shader.

Light shaders are also responsible for shadow casting, as illustrated in Figure 13-6. In the light shader, we can compute whether or not the fragment is currently visible to the light source, using a shadow map rendered in a previous pass or a software-generated one. If the fragment is not visible, the light will typically return a black color, meaning there will be no light contribution to the illumination of the material.

Listing 13-3 is a sample implementation of a spotlight.

NVIDIA Ex1023
Page Q-252

Table 13-2. mental ray Global State Variables

Usage	State Variable Name	Cg Type	Description
Material	`miToWorld`	`float4x4`	Matrix for transforming camera-space coordinates found in `vert2frag.position` to world space.
	`miFromWorld`	`float4x4`	Matrix for transforming world-space coordinates to camera space.
	`miToObject`	`float4x4`	Matrix for transforming camera-space coordinates to object space.
	`miFromObject`	`float4x4`	Matrix for transforming object-space coordinates to camera space.
Light Shaders Only	`miLightOrigin`	`float3`	Position of the light in camera space.
	`miLightDir`	`float3`	Direction of the light in camera space.
	`miLightSpread`	`float`	Spread angle for spotlights.
	`miLightType`	`int`	Type of the light: 0 = point, 1 = spot, 2 = directional
	`miLightFace`	`int`	Lighting face: 1 = front, 2 = back, 0 = both
	`miToLight`	`float4x4`	Matrix for transforming camera-space coordinates to light space.
	`miFromLight`	`float4x4`	Matrix for transforming light-space coordinates to camera space.
Camera	`miCameraAspect`	`float`	Aspect ratio: height × aspect = width
	`miCameraAperture`	`float`	Aperture size: atan(focal/aperture) = field of view
	`miCameraFocal`	`float`	Focal length: atan(focal/aperture) = field of view
	`miCameraClip`	`float2`	Clip plane: x = near, y = far
	`miCameraRes`	`float2`	Resolution of the window.
	`miCameraOrtho`	`bool`	Is the camera orthographic?
Shadow Map (Light Shaders Only)	`miShadowMap`	`bool`	If true, the global shadow map option is on.
	`miShadowMapTex`	`samplerRECT[1|6]`	Shadow map textures. 1 for spot and directional lights, 6 for point lights.
	`miShadowMapTransform`	`float4x4`	Matrix for transforming `vert2frag.position` to light space.
	`miShadowMapWindow`	`float2`	The transformation to apply to UV coordinates.
	`miShadowMapSize`	`float2`	The resolution of the shadow map.
	`miShadowMapBias`	`float`	The bias applied to the shadow map.
Shader	`miRayEnvironment`	`miiColor`	Connection to the environment shader.

NVIDIA Ex1023
Page Q-253

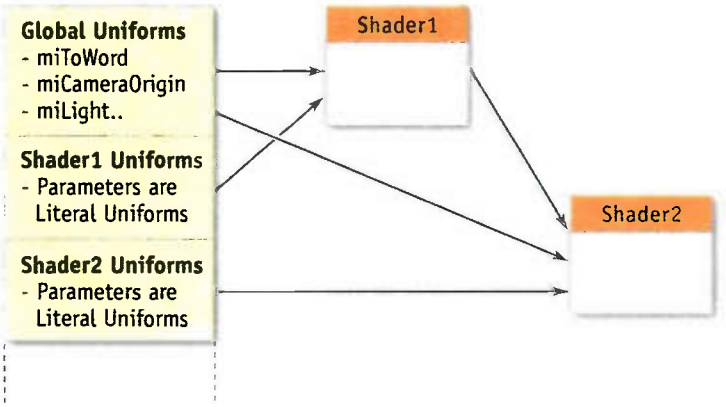


Figure 13-5. How Global Uniforms Are Shared

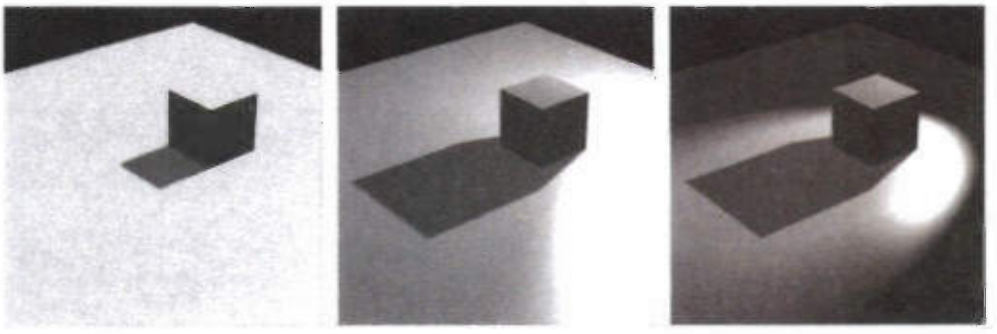


Figure 13-6. Directional, Point, and Spotlight Illumination and Shadow Map

Listing 13-3. Partial Cg Implementation of the mental ray Base Spotlight

```
struct mib_light_spot : miiLight {
  // parameters of the shader
  miiColor color;
  miiBoolean shadow;
  miiScalar factor;
  miiBoolean atten;
  miiScalar start;
  miiScalar stop;
  miiScalar cone;

  // Information from the light instance
  // position of the light
  float3 miLightOrigin;
```


NVIDIA Ex1023
Page Q-254

Listing 13-3 *(continued)*. Partial Cg Implementation of the mental ray Base Spotlight

```
  // direction of the light
  float3 miLightDir;
  // cos angle of the spread
  float miLightSpread;

  bool miShadowMap;              // Shadow active or not?
  samplerRECT miShadowMapTex[1]; // light shadow map
  float4x4 miShadowMapTransfo;   // From camera to light space
  float2 miShadowMapWindow;      // size of the window
  float2 miShadowMapSize;        // size of the shadow map
  float miShadowMapBias;

  // Evaluation of the light
  misLightOut eval(vert2frag p) {
    misLightOut ret_val;
    float4 lcolor = color.eval(p);

    // vector to light
    ret_val.dir = normalize(miLightOrigin - p.position);
    ret_val.dot_nl = dot(p.normal, ret_val.dir);

    // shadow
    if (shadow.eval(p) && miShadowMap) {
      float lfactor = factor.eval(p);
      . . .
      lcolor.rgb *= lerp(lfactor, 1, map_z/4);
    }

    // cone
    . . .

    // dist attenuation
    . . .

    ret_val.color = lcolor;
    return ret_val;
  }
};
```


NVIDIA Ex1023
Page Q-255

13.3.7 Texture Shaders

Texture shaders typically return a color from a texture image. They can also be procedural, like the classic marble or wood shaders. When a shader input parameter is attached to a texture image, an explicit `miColorTexture` parameter is created and its texture ID is mapped to the `sampler2D` parameter. By enabling automatic Cg runtime texture parameter management with a call to `cgGLSetManageTextureParameters()`, we do not have to handle the activation of the texture when the program is bound. This way we can load all textures and simply use `cgSetTextureParameter()` to access the image in the shader.

All evaluation methods of the basic interface need only one parameter, the `vert2frag` structure, except for the `miiColorTexture` interface. In addition to the usual varying parameters, we also add a `float2` parameter to pass the UV coordinates.

One nice thing about texture shaders is the ability to generate procedural texture coordinates. Usually UVs are attached to each vertex and are often generated from texture projections. These projections can be created in fragment programs, which can resolve all the artifacts that often arise when applying spherical mappings or other parametric projections at the vertex level.

Figure 13-7 (on the next page) shows a texture shader output, connected to the diffuse parameter of the Phong shader. The texture coordinates are calculated per fragment with a spherical projection.

Listing 13-4 is an example of a simple texture-lookup shader that returns a color.

Listing 13-4. Cg Implementation of the mental ray Texture Lookup Shader

```
struct mib_texture_lookup : miiColor
{
  miiColorTexture tex;
  miiVector coord;
  float4 eval(vert2frag p) {
    // Evaluation of the spherical projection shader
    float3 lcoord = coord.eval(p);
    // Evaluate to a simple tex2D call
    float4 ret_val = tex.eval(p, lcoord.xy);
    return ret_val;
  }
};
```


NVIDIA Ex1023
Page Q-256



Figure 13-7. A Texture Shader Connected to the Diffuse Parameter of `mib_illum_phong`

13.3.8 Bump Mapping

In Figure 13-8, the normal vector used for the illumination of a material is perturbed by the bump-map shader. To apply bump mapping to a material shader, we have to modify the varying state normal before the evaluation of the material shader. We do this by adding an extra shader node to our graph, as illustrated in Figure 13-9. The interface of this node takes two parameters: the first one calls the normal modifier and returns the value to the varying state normal; the second parameter calls the material shader. Because the normal used by the material shader has been modified before its evaluation, the lighting is appropriately affected by the bump.

Because this is a shader returning a vector, the shader implementing the normal modifier can be of any type. It can be a normal texture lookup, it can modify the normal based on multiple texture lookups, or it can be procedural. The derivatives sent through the varying parameter ensure that you always sample in the same direction and that the bumps rotate appropriately with the object.

NVIDIA Ex1023
Page Q-257



Figure 13-8. `mib_illum_phong` with the Normal Modified Using a Bump-Map Shader

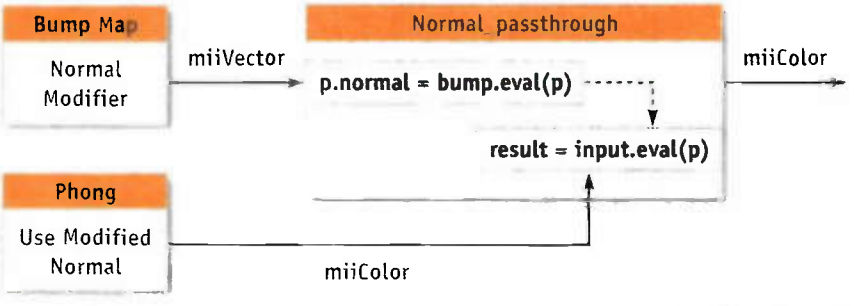


Figure 13-9. The Structure of a Bump Map Shader

13.3.9 Environment and Volume Shaders

The environment shader is an optional extra root of Phenomena that is evaluated if a ray goes to infinity. It can be called for reflective or transparent surfaces, or simply when there are no objects visible in a particular pixel. To make sure we are calling the environment shader for all such pixels of the rendered image, we draw a plane at epsilon in front of the camera's far clip plane.

NVIDIA Ex1023
Page Q-258

Because fully general ray tracing is not yet supported on the GPU, reflective or refractive materials do not have the option of sending rays and hitting other objects or themselves. Therefore, they can only reflect their environment. To call the global environment shader graph from another shader, we must define a global state variable. In Figure 13-10, we show the connection between the graph and the global state variable `miRayEnvironment` of type `miiColor`.

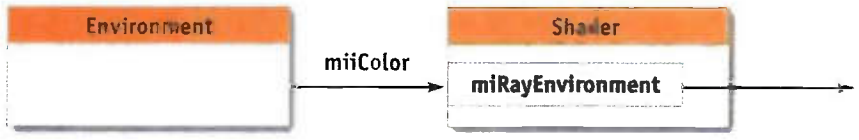


Figure 13-10. The Structure of an Environment Shader

Volume shaders are another Phenomenon root. These shaders need to be called for every rendered fragment. The volume shader attached to the camera affects the rendered color depending on different factors. One simple volume model is fog, which attenuates the color based on the distance to the camera. Because volume shaders need to be called for every fragment, the shader node is placed automatically on top of the main Phenomenon root, as illustrated in Figure 13-11.



Figure 13-11. The Structure of a Volume Shader

In Figure 13-12, the blue shader for the spikes does not need to be modified due to the fog effect applied in this scene. The layer fog shader attached to the camera modifies the output of each of the scene's shaders, giving an atmospheric effect to the scene.

13.3.10 Shaders Returning Structures

Shaders are not limited to returning a single value; they can have multiple results. The values computed in such shaders are returned as a structure. A shader input parameter

NVIDIA Ex1023
Page Q-259

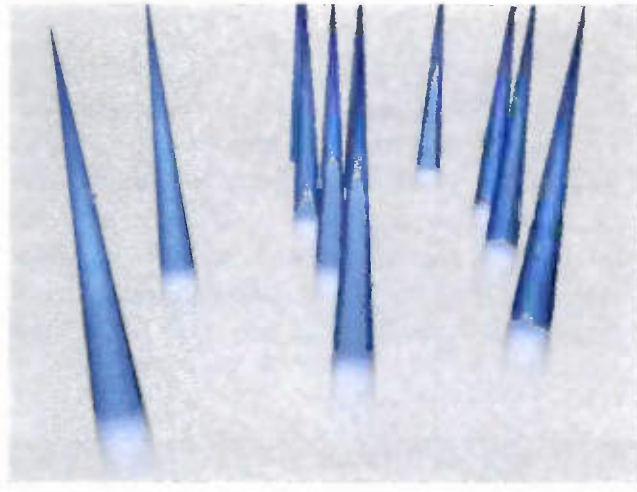


Figure 13-12. Image Rendered Using a Volume Shader

might connect to one of the members of the structure. Because Cg cannot connect interfaces of different types, we must implement converters.

First, the shader that returns the structure must implement the interface of this structure. This interface is needed to connect to a converter. All members of the structure must have a converter node in order to be connected to a shader parameter. A shader returning a color and a vector will have to implement two converters: a structure `mystruct_outColor` implementing `miiColor` and a structure `mystruct_outVector` implementing `miiVector`. The converter has its input parameter connected to the output of the shader and returns a member of that structure. The resulting structure member can now be connected to the destination shader parameter.

Converters can be used by many shaders. To avoid duplicating them, we simply include the converter code using an `#include` statement. Figure 13-13 shows the conversion from an output structure containing a vector to an `miiVector`.

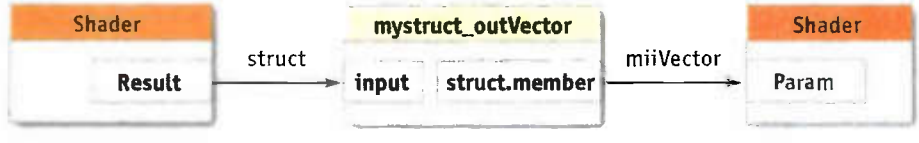


Figure 13-13. A Shader that Returns a Structure

NVIDIA Ex1023
Page Q-260

13.3.11 Rendering Hair

In mental ray, hairs are rendered as Bezier curves. Hair width can be specified using a global radius, a per-hair radius, or a per-vertex radius. The approximation of the curves can generate millions of triangles when rendered with the GPU, each often smaller than a fragment. Because hairs are converted to triangles, they can be treated like any other geometry. We can attach a Phenomenon and use the varying parameter to follow the tangent of the hair, the distance from the root relative to the tip, and so on. This information is sufficient to implement any lighting model, because there is no fundamental difference between a hair Phenomenon and any other material Phenomenon. Figure 13-14 shows an example.



Figure 13-14. A Rendered Hairy Bean

13.3.12 Putting It All Together

When rendering a scene, we start collecting all the lights that need to cast shadow maps. mental ray also uses a Cg shader for evaluating the distance to the light, and the result is stored in a floating-point pbuffer.

NVIDIA Ex1023
Page Q-261

We then convert all Phenomena to Cg and construct a unique tree from the different roots. If the resulting shader tree compiles successfully for the profile of the graphics card, the object is drawn by the GPU; otherwise, software renders it.

Because some materials are transparent, we draw the entire scene using an order-independent-transparency technique, also known as depth peeling (Everitt 2001). The shaders know the distance between each level and adjust the color contribution to the frame buffer. This is, for example, how we render hair primitives.

When all render passes are completed in hardware, the frame buffer is merged using the z-depth value and alpha to blend with the software pass.

During this process, a number of things can go wrong. For example, there might not be enough video or texture memory, or the Cg shader may not compile because of hardware limitations. In any of these cases, the objects in which errors occurred are rendered with software.

13.4 Conclusion

Phenomena converted to Cg and Phenomena evaluated by software produce the same result, and in the end, the hardware-rendered image is not noticeably different from the software-rendered version, which was our original goal. With hardware multisampling and supersampling, we can achieve comparable image quality as with software, and rendering on the GPU takes just a small fraction of the time needed for software to render a frame.

In complex production scenes, compiling the numerous shaders takes some time during preprocessing of the scene. Sometimes the Cg compiler overhead can reduce the overall gain from high-speed hardware rendering. But when shaders are compiled and all textures reside in video memory, there is no doubt that using the GPU to render the scene is far faster than using the CPU. With numerous test scenes, we have found that GPUs consistently render an order of magnitude or more faster than CPUs.

GPUs are now so fast that we can push the limit on the numbers of triangles in a scene. Because all triangles do not need to be resident in memory, we can render scenes that are very expensive to render in software. On the other hand, mental ray's software renderer provides functionality that cannot yet be achieved by hardware. Knowing the limitations of the hardware, we can create scenes to exploit its strengths and reduce rendering time by one or two orders of magnitude.

NVIDIA Ex1023
Page Q-262

At the time this chapter was written, we noticed that the time to render a frame was being cut by more than a factor of two per year. Each new generation of GPUs provides more power and more memory, and overcomes more limitations. In the near future, current hardware limitations will disappear and will make the usage of GPUs even more common. We think that by 2010, almost all rendering will be GPU based.

13.5 References

Driemeyer, Thomas, ed. 2005. *Rendering with Mental Ray*, 3rd ed. Springer.

Everitt, Cass. 2001. "Order-Independent Transparency." Technical report. NVIDIA Corporation. Available online at **http://developer.nvidia.com/view.asp?IO=order_independent_transparency**

Pharr, Matt. 2004. "An Introduction to Shader Interfaces." In *GPU Gems*, edited by Randima Fernando, pp. 537–550. Addison-Wesley.

NVIDIA Ex1023
Page Q-263



Chapter 14

Dynamic Ambient Occlusion and Indirect Lighting

Michael Bunnell
NVIDIA Corporation

In this chapter we describe a new technique for computing diffuse light transfer and show how it can be used to compute global illumination for animated scenes. Our technique is efficient enough when implemented on a fast GPU to calculate ambient occlusion and indirect lighting data on the fly for each rendered frame. It does not have the limitations of precomputed radiance transfer (PRT) or precomputed ambient occlusion techniques, which are limited to rigid objects that do not move relative to one another (Sloan 2002). Figure 14-1 illustrates how ambient occlusion and indirect lighting enhance environment lighting.

Our technique works by treating polygon meshes as a set of surface elements that can emit, transmit, or reflect light and that can shadow each other. This method is so efficient because it works without calculating the visibility of one element to another. Instead, it uses a much simpler and faster technique based on approximate shadowing to account for occluding (blocking) geometry.

14.1 Surface Elements

The first step in our algorithm is to convert the polygonal data to surface elements to make it easy to calculate how much one part of a surface shadows or illuminates another. Figure 14-2 illustrates the basic concept. We define a surface element as an oriented disk with a position, normal, and area. An element has a front face and a back

NVIDIA Ex1023
Page Q-264



Figure 14-1. Adding Realism with Ambient Occlusion and Indirect Lighting
The scene on the left uses only environment lighting and looks very flat. The middle scene adds soft shadows using ambient occlusion. The scene on the right adds indirect lighting for an extra level of realism.

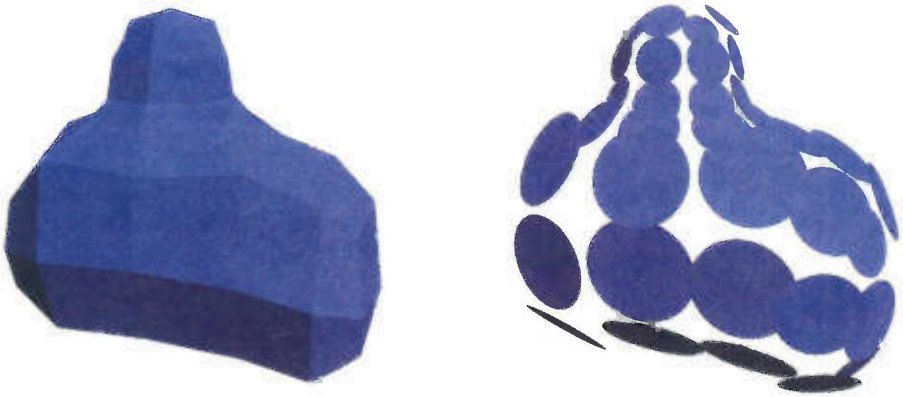


Figure 14-2. Converting a Polygonal Mesh to Elements
Left: A portion of a polygonal mesh. Right: The mesh represented as disk-shaped elements.

face. Light is emitted and reflected from the front-facing side. Light is transmitted and shadows are cast from the back. We create one element per vertex of the mesh. Assuming that the vertices are defined with a position and normal already, we just need to calculate the area of each element. We calculate the area at a vertex as the sum of one-third of the area of the triangles that share the vertex (or one-fourth of the area for quads). Heron's formula for the area of a triangle with sides of length a, b, and c is:

NVIDIA Ex1023
Page Q-265

$$\sqrt{s(s-a)(s-b)(s-c)},$$

where s is half the perimeter of the triangle: $(a + b + c)/2$.

We store element data (position, normal, and area) in texture maps because we will be using a fragment program (that is, a pixel shader) to do all the ambient occlusion calculations. Assuming that vertex positions and normals will change for each frame, we need to be able to change the values in the texture map quickly. One option is to keep vertex data in a texture map from the start and to do all the animation and transformation from object space to eye (or world) space with fragment programs instead of vertex programs. We can use render-to-vertex-array to create the array of vertices to be sent down the regular pipeline, and then use a simple pass-through vertex shader. Another, less efficient option is to do the animation and transformation on the CPU and load a texture with the vertex data each frame.

14.2 Ambient Occlusion

Ambient occlusion is a useful technique for adding shadowing to diffuse objects lit with environment lighting. Without shadows, diffuse objects lit from many directions look flat and unrealistic. Ambient occlusion provides soft shadows by darkening surfaces that are partially visible to the environment. It involves calculating the accessibility value, which is the percentage of the hemisphere above each surface point not occluded by geometry (Landis 2002). In addition to accessibility, it is also useful to calculate the direction of least occlusion, commonly known as the *bent normal*. The bent normal is used in place of the regular normal when shading the surface for more accurate environment lighting.

We can calculate the accessibility value at each element as 1 minus the amount by which all the other elements shadow the element. We refer to the element that is shadowed as the *receiver* and to the element that casts the shadow as the *emitter*. We use an approximation based on the solid angle of an oriented disk to calculate the amount by which an emitter element shadows a receiver element. Given that A is the area of the emitter, the amount of shadow can be approximated by:

$$1 - \frac{r \cos\theta_E \max\left(1,\ 4\cos\theta_R\right)}{\sqrt{\frac{A}{\pi} + r^2}}.$$

Equation 14-1. Shadow Approximation

NVIDIA Ex1023
Page Q-266

As illustrated in Figure 14-3, θ_E is the angle between the emitter's normal and the vector from the emitter to the receiver. θ_R is the corresponding angle for the receiver element. The max(1, 4 × cos θ_R) term is added to the disk solid angle formula to ignore emitters that do not lie in the hemisphere above the receiver without causing rendering artifacts for elements that lie near the horizon.

Here is the fragment program function to approximate the element-to-element occlusion:

```
float ElementShadow(float3 v, float rSquared, float3 receiverNormal,
                    float3 emitterNormal, float emitterArea)
{
// we assume that emitterArea has already been divided by PI
  return (1 - rsqrt(emitterArea/rSquared + 1)) *
         saturate(dot(emitterNormal, v)) *
         saturate(4 * dot(receiverNormal, v));
}
```

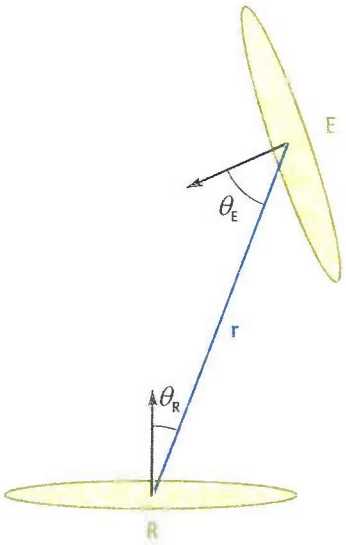


Figure 14-3. The Relationship Between Receiver and Emitter Elements
Receiver element R receives light or shadow from emitter E with r as the distance between the centers of the two elements.

14.2.1 The Multipass Shadowing Algorithm

We calculate the accessibility values in two passes. In the first pass, we approximate the accessibility for each element by summing the fraction of the hemisphere subtended by

NVIDIA Ex1023
Page Q-267

every other element and subtracting the result from 1. After the first pass, some elements will generally be too dark because other elements that are in shadow are themselves casting shadows. So we use a second pass to do the same calculation, but this time we multiply each form factor by the emitter element's accessibility from the last pass. The effect is that elements that are in shadow will cast fewer shadows on other elements, as illustrated in Figure 14-4. After the second pass, we have removed any double shadowing. However, surfaces that are triple shadowed or more will end up being too light. We can use more passes to get a better approximation, but we can approximate the same answer by using a weighted average of the combined results of the first and second passes. Figure 14-5 shows the results after each pass, as well as a ray-traced solution for comparison. The bent normal calculation is done during the second pass. We compute the bent normal by first multiplying the normalized vector between elements and the form factor. Then we subtract this result from the original element normal.

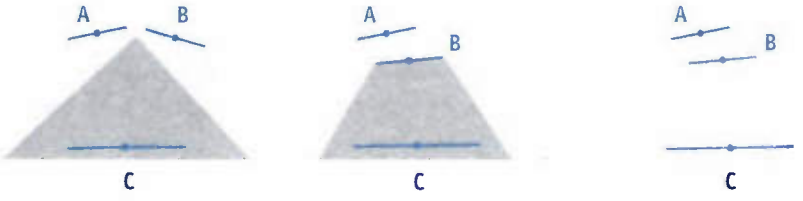


Figure 14-4. Correcting for Occlusion by Overlapping Objects
Left: Elements A and B correctly shadow C after the first pass. Middle: In this arrangement, B casts too much shadow on C. Right: B's shadow is adjusted by the second pass to shadow C properly.

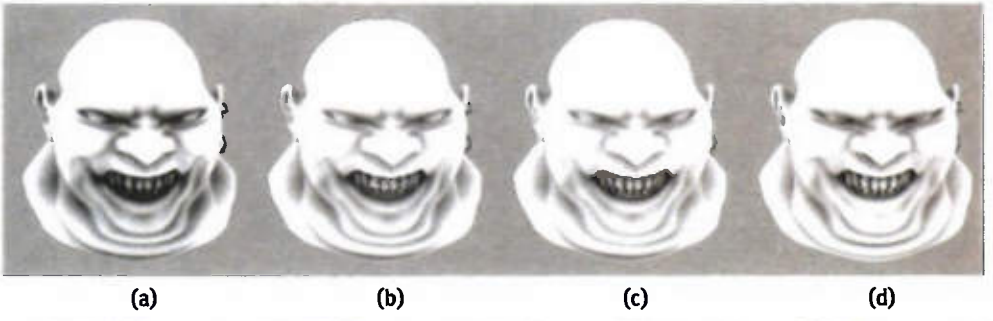


Figure 14-5. Comparing Models Rendered with Our Technique to Reference Images
Model rendered with ambient occlusion accessibility value calculated after (a) one pass, (b) two passes, and (c) three passes of the shader. (d) Image made by tracing 200 random rays per vertex for comparison. Model courtesy of Bay Raitt.

NVIDIA Ex1023
Page Q-268

We calculate the occlusion result by rendering a single quad (or two triangles) so that one pixel is rendered for each surface element. The shader calculates the amount of shadow received at each element and writes it as the alpha component of the color of the pixel. The results are rendered to a texture map so the second pass can be performed with another render. In this pass, the bent normal is calculated and written as the RGB value of the color with a new shadow value that is written in the alpha component.

14.2.2 Improving Performance

Even though the element-to-element shadow calculation is very fast (a GeForce 6800 can do 150 million of these calculations per second), we need to improve our algorithm to work on more than a couple of thousand elements in real time. We can reduce the amount of work by using simplified geometry for distant surfaces. This approach works well for diffuse lighting environments because the shadows are so soft that those cast by details in distant geometry are not visible. Fortunately, because we do not use the polygons themselves in our technique, we can create surface elements to represent simplified geometry without needing to create alternate polygonal models. We simply group elements whose vertices are neighbors in the original mesh and represent them with a single, larger element. We can do the same thing with the larger elements, creating fewer and even larger elements, forming a hierarchy. Now instead of traversing every single element for each pixel we render, we traverse the hierarchy of elements. If the receiver element is far enough away from the emitter—say, four times the radius of the emitter—we use it for our calculation. Only if the receiver is close to an emitter do we need to traverse its children (if it has any). See Figure 14-6. By traversing a hierarchy in this way, we can improve the performance of our algorithm from $O(n^2)$ to $O(n \log n)$ in practice. The chart in Figure 14-7 shows that the performance per vertex stays consistent as the number of vertices in the hierarchy increases.

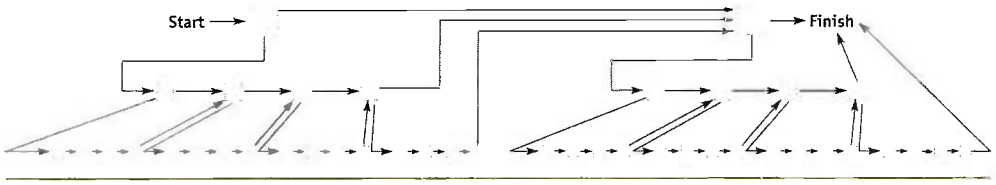


Figure 14-6. Hierarchical Elements
Elements are traversed in a hierarchy; child elements are traversed only if necessary.

NVIDIA Ex1023
Page Q-269

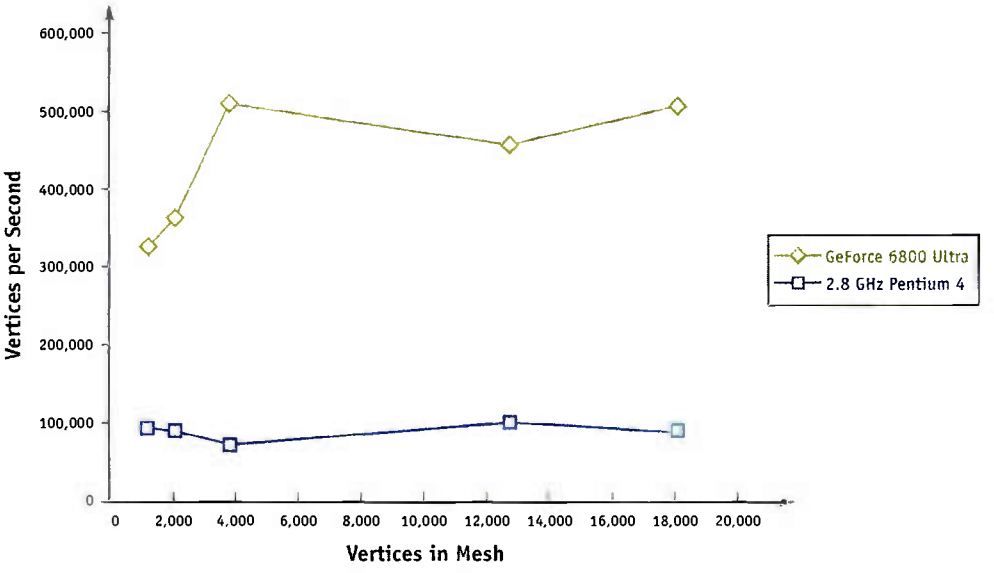


Figure 14-7. Ambient Occlusion Shader Performance for Meshes of Different Densities

We calculate a parent element's data using its direct descendants in the hierarchy. We calculate the position and normal of a parent element by averaging the positions and normals of its children. We calculate its area as the sum of its children's areas. We can use a shader for these calculations by making one pass of the shader for each level in the hierarchy, propagating the values from the leaf nodes up. We can then use the same technique to average the results of an occlusion pass that are needed for a following pass or simply treat parent nodes the same as children and avoid the averaging step. It is worth noting that the area of most animated elements varies little, if at all, even for nonrigid objects; therefore, the area does not have to be recalculated for each frame.

The ambient occlusion fragment shader appears in Listing 14-1.

Listing 14-1. Ambient Occlusion Shader

```
float4 AmbientOcclusion(
  float4 position : WPOS,
  float4 normOffset : TEX1,
  uniform samplerRECT lastResultMap : TEXUNIT0,
  uniform samplerRECT positionMap : TEXUNIT1,
  uniform samplerRECT elementNormalMap : TEXUNIT2,
  uniform samplerRECT indexMap : TEXUNIT3) : COL
{
```


NVIDIA Ex1023
Page Q-270

Listing 14-1 *(continued)*. Ambient Occlusion Shader

```
    float eArea;       // emitter area
    float4 ePosition; // emitter position
    float4 eNormal;    // emitter normal
    float3 rPosition = texRECT(positionMap, position.xy).xyz;
    float3 rNormal = texRECT(elementNormalMap, position.xy).xyz;
    float3 v;          // vector from receiver to emitter
    float total = 0;  // used to calculate accessibility
    float4 eIndex = float2(0.5, 0.5); // index of current emitter
    float3 bentNormal = rNormal;      // initialize with receiver normal
    float value;
    float d2;          // distance from receiver to emitter squared

    while (emitterIndex.x != 0) { // while not finished traversal
      ePosition = texRECT(positionMap, emitterIndex.xy);
      eNormal = texRECT(elementNormalMap, emitterIndex.xy);
      eArea = emitterNormal.w;
      eIndex = texRECT(indexMap, emitterIndex.xy); // get next index
      v = ePosition.xyz - rPosition; // vector to emitter
      d2 = dot(v, v) + 1e - 16;   // calc distance squared, avoid 0
      // is receiver close to parent element?
      if (d2 < -4*emitterArea) {  // (parents have negative area)
        eIndex.xy = eIndex.zw;    // go down hierarchy
        emitterArea = 0;          // ignore this element
      }
      v *= rsqrt(d2);             // normalize v
      value = SolidAngle(v, d2, rNormal, eNormal.xyz, abs(eArea)) *
              texRECT(resultMap, position.xy).w; // modulate by last result
      bentNormal -= value * v;    // update bent normal
      total += value;
    }
    if (!lastPass)                // only need bent normal for last pass
      return saturate(1 - total); // return accessibility only
    else return float4(normalize(bentNormal), 1 - total);
}
```


NVIDIA Ex1023
Page Q-271

14.3 Indirect Lighting and Area Lights

We can add an extra level of realism to rendered images by adding indirect lighting caused by light reflecting off diffuse surfaces (Tabellion 2004). We can add a single bounce of indirect light using a slight variation of the ambient occlusion shader. We replace the solid angle function with a disk-to-disk radiance transfer function. We use one pass of the shader to transfer the reflected or emitted light and two passes to shadow the light.

For indirect lighting, first we need to calculate the amount of light to reflect off the front face of each surface element. If the reflected light comes from environment lighting, then we compute the ambient occlusion data first and use it to compute the environment light that reaches each vertex. If we are using direct lighting from point or directional lights, we compute the light at each element just as if we are shading the surface, including shadow mapping. We can also do both environment lighting and direct lighting and sum the two results. We then multiply the light values by the color of the surface element, so that red surfaces reflect red, yellow surfaces reflect yellow, and so on. Area lights are handled just like light-reflective diffuse surfaces except that they are initialized with a light value to emit.

Here is the fragment program function to calculate element-to-element radiance transfer:

```
float FormFactor(float3 v, float d2, float3 receiverNormal,
                 float3 emitterNormal, float emitterArea)
{
  // assume that emitterArea has been divided by PI
  return emitterArea * saturate(dot(emitterNormal, v)) *
    saturate(dot(receiverNormal, v)) / (d2 + emitterArea);
}
```

$$\frac{A \cos\theta_E \cos\theta_R}{\pi r^2 + A}$$

Equation 14-2. Disk-to-Disk Form Factor Approximation

We calculate the amount of light transferred from one surface element to another using the geometric term of the disk-to-disk form factor given in Equation 14-2. We leave off the visibility factor, which takes into account blocking (occluding) geometry. Instead we use a shadowing technique like the one we used for calculating ambient occlusion—only this time we use the same form factor that we used to transfer the light. Also, we multiply the shadowing element's form factor by the three-component light value instead of a single-component accessibility value.

NVIDIA Ex1023
Page Q-272

We now run one pass of our radiance-transfer shader to calculate the maximum amount of reflected or emitted light that can reach any element. Then we run a shadow pass that subtracts from the total light at each element based on how much light reaches the shadowing elements. Just as with ambient occlusion, we can run another pass to improve the lighting by removing double shadowing. Figure 14-8 shows a scene lit with direct lighting plus one and two bounces of indirect lighting.

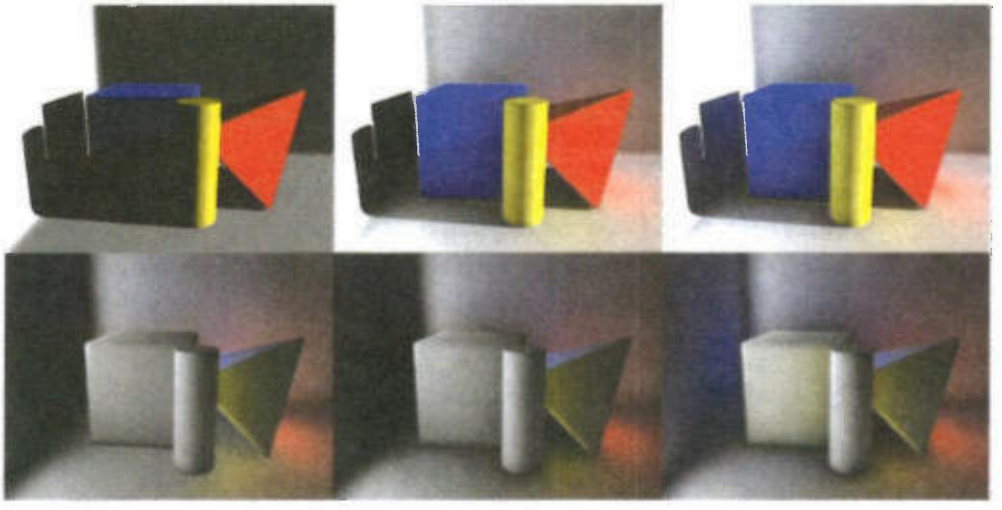


Figure 14-8. Combining Direct and Indirect Lighting
Top, left to right: Scene lit with direct lighting, with direct lighting plus one bounce of indirect lighting, and with direct lighting plus two bounces of indirect lighting. Bottom, left to right: Indirect lighting after one pass, after two passes (one bounce), and after two bounces (four passes total).

14.4 Conclusion

Global illumination techniques such as ambient occlusion and indirect lighting greatly enhance the quality of rendered diffuse surfaces. We have presented a new technique for calculating light transfer to and from diffuse surfaces using the GPU. This technique is suitable for implementing various global illumination effects in dynamic scenes with deformable geometry.

NVIDIA Ex1023
Page Q-273

14.5 References

Landis, Hayden. 2002. "Production-Ready Global Illumination." Course 16 notes, SIGGRAPH 2002.

Pharr, Matt, and Simon Green. 2004. "Ambient Occlusion." In *GPU Gems*, edited by Randima Fernando, pp. 279–292. Addison-Wesley.

Sloan, Peter-Pike, Jan Kautz, and John Snyder. 2002. "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)* 21(3), pp. 527–536.

Tabellion, Eric, and Arnauld Lamorlette. 2004. "An Approximate Global Illumination System for Computer Generated Films." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)* 23(3), pp. 469–476.

NVIDIA Ex1023
Page Q-274



Chapter 15

Blueprint Rendering and "Sketchy Drawings"

Marc Nienhaus
University of Potsdam, Hasso-Plattner-Institute

Jürgen Döllner
University of Potsdam, Hasso-Plattner-Institute

In this chapter, we present two techniques for hardware-accelerated, image-space nonphotorealistic rendering: blueprint rendering and sketchy drawing.

Outlining and enhancing visible and occluded features in drafts of architecture and technical parts are essential techniques for visualizing complex aggregate objects and for illustrating the position, layout, and relation of their components. Blueprint rendering is a novel multipass rendering technique that outlines visible and nonvisible visually important edges of 3D objects. The word *blueprint* in its original sense is defined by Merriam-Webster as "a photographic print in white on a bright blue ground or blue on a white ground used especially for copying maps, mechanical drawings, and architects' plans." Blueprints consist of transparently rendered features, represented by their outlines. Thus, blueprints make it easy to understand the structure of complex, hierarchical object assemblies such as those found in architectural drafts, technical illustrations, and designs.

In this sense, blueprint rendering can become an effective tool for interactively exploring, visualizing, and communicating spatial relations. For instance, they can help guide players through dungeon-like environments and highlight hidden chambers and components in a computer game, or they can make it easier for artists to visualize and design game levels in a custom level editor.

In contrast, rendering in a "sketchy" manner can be used to communicate visual ideas and to illustrate drafts and concepts, especially in application areas such as architectural

NVIDIA Ex1023
Page Q-275

and product designs. In particular, sketchy drawings encourage the exchange of ideas when people are reconsidering drafts.

Sketchy drawing is a real-time rendering technique for sketching visually important edges as well as inner color patches of arbitrary 3D geometries even beyond their geometrical boundary. Generally speaking, with sketchy drawing we sketch the outline of 3D geometries to imply vagueness and "crayon in" inner color patches exceeding the sketchy outline as though they have been painted roughly. Combining both produces sketchy, cartoon-like depictions that can be applied to enhance the visual attraction of characters and 3D scenery.

Because of the way they're implemented (with depth sprite rendering) and the fact that they work on a per-object basis, both blueprint and sketchy rendering can easily be integrated into any real-time graphics application.

15.1 Basic Principles

Blueprint and sketchy renderings are both built on three basic principles: (1) to preserve images of intermediate renderings of the scene's geometry, (2) to implement an edge enhancement technique, and (3) to apply depth sprite rendering. We briefly describe these concepts in the following sections.

15.1.1 Intermediate Rendering Results

We denote 2D data derived from 3D geometry and rendered into textures as *intermediate rendering results*. In doing so, we implement G-buffers (Saito and Takahashi 1990) that store geometric properties of 3D geometry. (See also Chapter 9 of this book, "Deferred Shading in *S.T.A.L.K.E.R.*," for application of this idea to realistic rendering.) In particular, we generate a normal buffer and a z-buffer by rendering encoded normal and z-values of 3D geometry directly into 2D textures, as shown in Figure 15-1. In a multipass rendering technique, we generate intermediate rendering results once and can reuse them in subsequent rendering passes. Technically, we use render-to-texture capabilities of graphics hardware. Doing so also allows us to render high-precision (color) values to high-precision buffers (such as textures).

15.1.2 Edge Enhancement

To extract visually important edges from 3D geometry on a per-object basis, we implement an image-space edge enhancement technique (Nienhaus and Döllner 2003).

NVIDIA Ex1023
Page Q-276

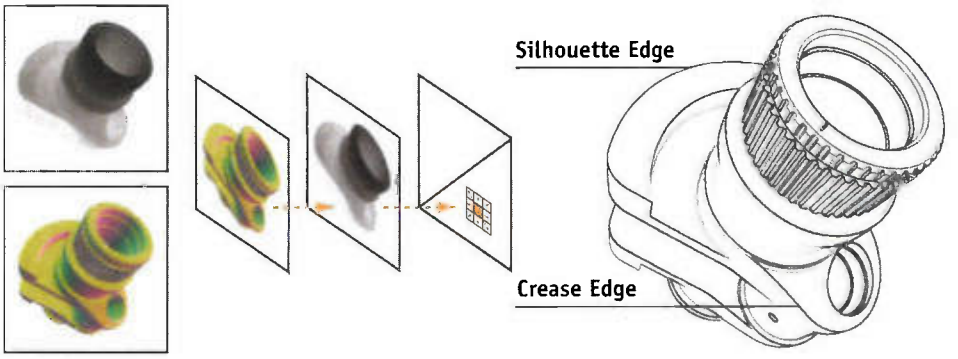


Figure 15-1. Steps in Creating an Edge Map
We texture a screen-aligned quad with the normal buffer and z-buffer and sample neighboring texture values to detect discontinuities, which are then represented as edge intensities. The resulting edge map contains visually important edges that are applicable to distinguish single components from each other in an overall composition.

Visually important edges include silhouette, border, and crease edges. We obtain these edges by detecting discontinuities in the normal buffer and z-buffer. For that purpose, we texture a screen-aligned quad that fills in the viewport of the canvas using these textures and calculate texture coordinates (s, t) for each fragment produced for the quad in such a way that they correspond to window coordinates. Sampling neighboring texels allows us to extract discontinuities that result in intensity values. The assembly of intensity values constitutes edges that we render directly into a single texture, called an *edge map*. Figure 15-2 depicts the normal and z-buffer and the resulting edge map.

15.1.3 Depth Sprite Rendering

Conceptually, depth sprites are 2D images that provide an additional z-value at each pixel for depth testing. For our purposes, we implement depth sprite rendering as follows:

1. We render a screen-aligned quad textured with a high-precision texture containing z-values.
2. We apply a specialized fragment shader that replaces fragment z-values of the quad (produced by the rasterizer) with z-values derived from accessing the texture. For optimizing fill rate, we additionally discard a fragment if its z-value equals 1—which denotes the depth of the far clipping plane. Otherwise, the fragment shader calculates RGBA color values of that fragment.
3. Rendering then proceeds with the ordinary depth test. If fragments pass the test, the frame buffer will be filled with the color and z-values of the depth sprite.

NVIDIA Ex1023
Page Q-277



Figure 15-2. The Z-Buffer, Normal Buffer, and Edge Map for Each Layer
The z-buffer (first row) and the normal buffer (second row) of each depth layer (column) form the basis for constructing the edge map (third row) for each layer.

As an example in 3D scenes, depth sprite rendering can correctly resolve the visibility of image-based renderings derived from 3D geometries by considering their z-values. In particular, we can combine blueprints and sketchy drawings with other 3D geometric models in arbitrary order.

15.2 Blueprint Rendering

For blueprint rendering, we extract both visible and nonvisible edges of 3D geometry. Visible edges are edges directly seen by the virtual camera; nonvisible edges are edges that are occluded by faces of 3D geometry; that is, they are not directly seen. To extract these edges, we combine the depth-peeling technique (Everitt 2001) and the edge enhancement technique. Once generated, we compose visible and nonvisible edges as blueprints in the frame buffer.

15.2.1 Depth Peeling

Depth peeling is a technique that operates on a per-fragment basis and extracts 2D layers of 3D geometry in depth-sorted order. Generally speaking, depth peeling successively "peels away" layers of unique depth complexity.

NVIDIA Ex1023
Page Q-278

In general, fragments passing an ordinary depth test define the minimal z-value at each pixel. But we cannot determine the fragment that comes second (or third, and so on) with respect to its depth complexity. Thus, we need an additional depth test to extract those fragments that form a layer of a given ordinal number (with respect to depth complexity) (Diefenbach 1996). We thus can extract the first n layers using n rendering passes.

We refer to a layer of unique depth complexity as a *depth layer* and a high-precision texture received from capturing the associated z-buffer contents as a *depth layer map*. Accordingly, we call the contents of the associated color buffer captured in an additional texture a *color layer map*. In particular, color layer maps can later be used in depth-sorted order to compose the final rendering, for example, for order-independent transparency (Everitt 2001).

The pseudocode in Listing 15-1 outlines our implementation of depth peeling. It operates on a set G of 3D geometries. We render G multiple times, whereby the rasterizer produces a set F of fragments. The loop terminates if no fragment gets rendered (*termination condition*); otherwise, we continue with the next depth layer. That is, if the number of rendering passes has reached the maximum depth complexity, the condition is satisfied.

Listing 15-1. Pseudocode for Combining Depth Peeling and Edge Enhancement for Blueprint Rendering

```
procedure depthPeeling(G ← 3DGeometry) begin
  i=0
  do
    F ← rasterize(G)
    /* Perform single depth test in the first rendering pass */
    if(i==0) begin
      for all fragment ∈ F begin
        bool test ← performDepthTest(fragment)
        if(test) begin
          fragment.depth → z-buffer
          fragment.color → color buffer
        end
        else reject fragment
      end
    end
    else begin
      /* Perform two depth test */
      for all fragment ∈ F begin
        if(fragment.depth > fragment.value_depthLayerMap(i-1)) begin
```


NVIDIA Ex1023
Page Q-279

Listing 15-1 *(continued)*. Pseudocode for Combining Depth Peeling and Edge Enhancement for Blueprint Rendering

```
          /* First test */
           bool test ← performDepthTest(fragment)
           if(test) begin
             /* Second test */
             fragment.depth → z-buffer
             fragment.color → color buffer
           end
           else reject fragment
         end
         else reject fragment
       end
     end
     depthLayerMap(i) ← capture(z-buffer)
     colorLayerMap(i) ← capture(color buffer)
     /* Edge intensities */
     edgeMap(i) ← edges(depthLayerMap(i),colorLayerMap(i))
     i++
   while(occlusionQuery() ≠ 0 )  /* Termination condition */
 end
```

Performing Two Depth Tests

In the first rendering pass (`i = 0`), we perform an ordinary depth test on each fragment. We capture the contents of the z-buffer and the color buffer in a depth layer map and a color layer map for further processing.

In consecutive rendering passes (`i > 0`), we perform an additional depth test on each fragment. For this test, we use a texture with the depth layer map of the previous rendering pass (`i - 1`). We then determine the texture coordinates of a fragment in such a way that they correspond to canvas coordinates of the targeted pixel position. In this way, a texture access can provide a fragment with the z-value stored at that pixel position in the z-buffer of the previous rendering pass.

Now, the two depth tests work as follows:

- If the current z-value of a fragment is greater than the texture value that results from accessing the depth layer map, the fragment proceeds and the second ordinary depth test is performed.
- Otherwise, if the test fails, the fragment is rejected.

NVIDIA Ex1023
Page Q-280

When we have processed all fragments, the contents of the z-buffer and the color buffer form the new depth layer map and color layer map. We can efficiently implement the additional depth test using a fragment program. Furthermore, we utilize the occlusion query extension (Kilgard 2004) to efficiently implement the termination condition.

15.2.2 Extracting Visible and Nonvisible Edges

We complement depth peeling with edge map construction for each depth layer. Because discontinuities in the normal buffer and z-buffer constitute visible edges, we must construct both for a rendering pass. We encode fragment normals as color values to generate the normal buffer as a color layer map. Then we can directly construct the edge map because the depth map already exists.

Furthermore, nonvisible edges become visible when depth layers are peeled away successively. Consequently, we can also extract nonvisible edges using the modified depth peeling technique (already shown in Listing 15-1).

As a result, our blueprint rendering technique preserves visible and nonvisible edges as edge maps for further processing. Figure 15-2 shows z-buffers, normal buffers, and resulting edge maps of successive depth layers.

Note that edges in edge maps of consecutive depth layers appear repeatedly because local discontinuities remain if we peel away faces of 3D geometry. Consider the following cases:

1. Two connected polygons share the same edge. One polygon occludes the other one. The discontinuity in the z-buffer that is produced along the shared edge will remain if we peel away the occluding polygon.
2. A polygon that partially occludes another polygon produces discontinuities in the z-buffer at the transition. If we peel away the occluding polygon and the nonoccluded portions, a discontinuity in the z-buffer will be produced at the same location.

Figure 15-3 illustrates both cases. However, the performance of edge map construction is virtually independent of the number of discontinuities.

15.2.3 Composing Blueprints

We compose blueprints using visible and nonvisible edges stored in edge maps in depth-sorted order. For each edge map, we proceed as follows:

NVIDIA Ex1023
Page Q-281

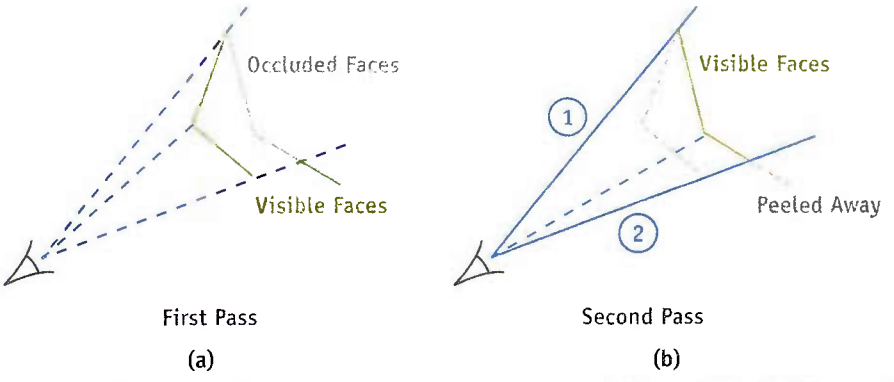


Figure 15-3. Two Possibilities When Peeling Away 3D Geometry
Rays are cast to discontinuities that are produced by the composition of polygons and that are visible from the camera position. (a) The first rendering pass. (b) The same composition with faces peeled away. The solid blue rays indicate edges that exist in both edge maps.

- We texture a screen-aligned quad that fills in the viewport with the edge map and its associated depth map.
- We further apply a fragment program that (1) performs depth sprite rendering using the appropriate depth layer map as input; (2) calculates the fragment's RGB value using the edge intensity value derived from accessing the edge map and, for instance, a bluish color; and (3) sets the fragment's alpha value to the edge intensity.
- We then use color blending by considering edge intensity values as blending factors for providing depth complexity cues while keeping edges enhanced.

We have found that in practice, it is sufficient to blend just the first few color layer maps to compose blueprints. The remaining layer maps have less visual impact on the overall composition because only a few (often isolated) pixels get colored. To alter the termination conditions for blueprint rendering and thus optimize rendering speed, we specify a desired minimal fraction of fragments (depending on the window resolution) to pass the depth test. In this way, we decrease the number of rendering passes while we maintain the visual quality of blueprints. To implement the trade-off between speed and quality, we configure the occlusion query extension appropriately. Figure 15-4a shows the resulting blueprint of the crank model.

15.2.4 Depth Masking

Depth masking is a technique that peels away a minimal number of depth layers until a specified fraction of a single, occluded component becomes visible. In fact, depth

NVIDIA Ex1023
Page Q-282

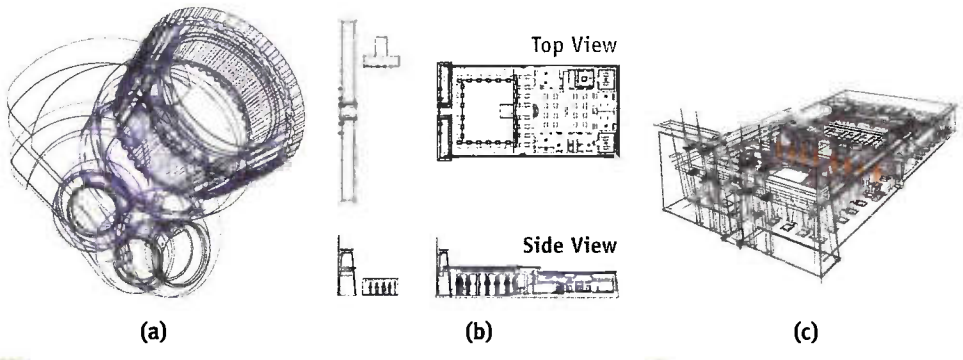


Figure 15-4. Various Applications of the Technique
(a) Blueprint rendering applied to the model of a crank to outline its design. (b) With blueprint rendering, we can generate plan views of the Temple of Ramses II automatically. (Model provided by ART+COM, Berlin. Used with permission.) (c) Depth masking applied to the temple and its statues.

masking provides a termination condition for blueprint rendering to dynamically adapt the number of rendering passes. For depth masking, we proceed as follows:

1. We derive a depth texture as the *depth mask* of the designated component in a preceding rendering pass.
2. In successive rendering passes, we render the depth mask as a depth sprite whenever we have peeled away a depth layer. If a specified fraction of the number of fragments passes the ordinary depth test (based on the z-buffer contents just produced), we terminate our technique. Otherwise, we must peel away more depth layers.
3. Finally, we simply integrate the designated component when composing blueprints.

The modifications to our technique are shown in pseudocode in Listing 15-2. Again, we use the occlusion query extension to implement the termination condition.

Listing 15-2. Pseudocode with Modified Termination Condition to Implement Depth Masking

```
procedure depthPeeling(G ← 3DGeometry,
                       C ← geometryOfOccludedComponent) begin
  depthMask ← depthTexture(C)
  quad ← createTexturedScreenAlignedQuad(depthMask)
  renderDepthSprite(quad)      /* Render quad as depth sprite */
  int Q = occlusionQuery()    /* Number of fragments of the components */
  . . .
```


NVIDIA Ex1023
Page Q-283

Listing 15-2 *(continued)*. Pseudocode with Modified Termination Condition to Implement Depth Masking

```
do
  . . .
  renderDepthSprite(quad)
  int R = occlusionQuery()  /* Number of visible fragments of the
                               component */
while(R<fraction(Q))        /* Termination condition */
end
```

15.2.5 Visualizing Architecture Using Blueprint Rendering

Blueprints can be used to outline the design of architecture. We apply blueprint rendering to generate plan views of architecture. Composing plan views using an orthographic camera is a straightforward task. In the visualization of the Temple of Ramses II in Abydos (Figure 15-4b), we can identify chambers, pillars, and statues systematically. Thus, blueprints increase the visual perception in these illustrations. The plan views are produced automatically.

A perspective view still provides better spatial orientation and conceptual insight in blueprints of architecture. Figure 15-4c visualizes the design of the entrance and the inner yard of the temple with its surrounding walls and statues. These are in front of the highlighted statues that guard the doorway to the rear part of the temple. With depth masking, we determine the number of depth layers that occlude the guarding statues and, therefore, must be peeled away. In this way, we can reduce the visual complexity of a blueprint if the structural complexity of the model is more than can be reasonably displayed.

15.3 Sketchy Rendering

Our sketchy drawing technique considers visually important edges and surface colors for sketching 3D geometry nonuniformly using uncertainty.

Our rendering technique proceeds as follows: As with blueprint rendering, we first generate intermediate rendering results that represent edges and surface colors of 3D geometry. Then we apply uncertainty values in image space to sketch these textures nonuniformly. Finally, we adjust depth information in such a way that the resulting sketchy drawing can be merged with general 3D scene contents.

NVIDIA Ex1023
Page Q-284

15.3.1 Edges and Color Patches

We again use the edge enhancement technique to store edges of 3D geometry as an edge map for later usage, as shown in Figure 15-5a. In addition, we render surface colors of 3D geometry to texture to produce striking color patches that appear flat, cover all surface details, and emulate a cartoon-like style. We call that texture a *shade map*; see Figure 15-5b.

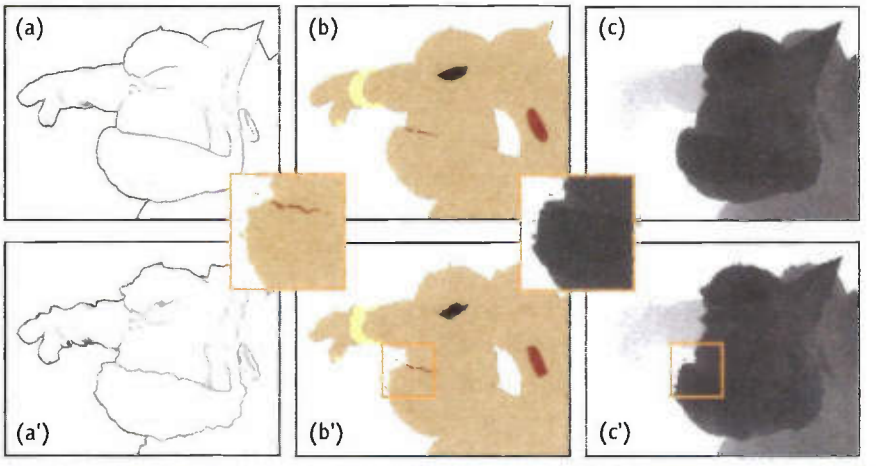


Figure 15-5. Applying Uncertainty Using Edge and Shade Maps
The edge map (a) and shade map (b) are two ingredients for sketchy drawings. Applying uncertainty results in perturbations of the edge map (a') and perturbations of the shade map (b'). Depth sprites use the depth map (c) to adjust z-values. For sketchy drawing, we reproduce the degrees of uncertainty applied to the edge and shade maps to perturb the depth map (c') for depth sprite rendering.

15.3.2 Applying Uncertainty

We apply uncertainty values to edges and surface colors in image space to simulate the effect of "sketching on a flat surface." For that purpose, we texture a screen-aligned quad (filling in the viewport of the canvas) using edge and shade maps as textures. Moreover, we apply an additional texture, whose texels represent uncertainty values. Because we want to achieve mostly continuous sketchy boundaries and frame-to-frame coherence, we opt for a noise texture whose texel values have been determined by the Perlin noise function (Perlin 1985); hence neighboring uncertainty values are correlated in image space. Once created (in a preprocessing step), the noise texture serves as an offset texture for accessing the edge and shade maps when rendering. That is, its texel

NVIDIA Ex1023
Page Q-285

values slightly perturb texture coordinates of each fragment of the quad that accesses the edge and shade maps.

In addition, we introduce a *degree of uncertainty* to control the amount of perturbation, for which we employ a user-defined 2×2 matrix. We multiply uncertainty values derived from the noise texture by that matrix to weight all these values uniformly and then use the resulting offset vector to translate the texture coordinates. Figure 15-6 illustrates the perturbation of the texture coordinates that access the shade map using the degree of uncertainty.

To enhance the sketchiness effect, we perturb texture coordinates for accessing the edge and shade maps differently. We thus apply two different 2×2 matrices, resulting in different degrees of uncertainty for each map. One degree of uncertainty shifts texture coordinates of the edge map, and one shifts texture coordinates of the shade map—that is, we shift them in opposite directions. Figure 15-5 shows the edge and shade maps after we have applied uncertainty.

We denote texture values that correspond to fragments of 3D geometry as *interior regions*; texture values that do not correspond to fragments of 3D geometry are called *exterior regions*. So in conclusion, by texturing the quad and perturbing texture coordinates using uncertainty, we can access interior regions of the edge and shade maps, whereas the initial texture coordinates would access exterior regions and vice versa (Figure 15-6). In this way, interior regions can be sketched beyond the boundary of 3D geometry, and exterior regions can penetrate interior regions. We can even produce spots beyond the boundary of the 3D geometry (Figure 15-5).

Finally, we combine texture values of the edge and the shade maps. We multiply the intensity values derived from perturbing the edge map with the color values derived from

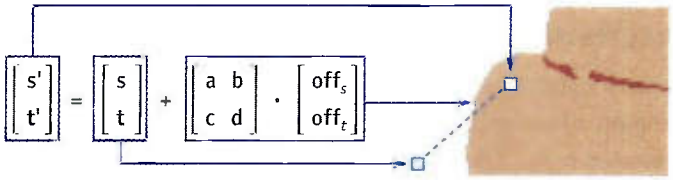


Figure 15-6. Applying Uncertainty in Image Space
We multiply the uncertainty value (off_s,off_t) derived from the noise texture by a 2×2 matrix (with weights a, b, c, and d) forming the degree of uncertainty that we apply to the texture coordinates (s, t) of a fragment to translate them in image space. In the figure, the perturbed texture coordinates (s', t') access a texture value of the interior region of the shade map even though the initial texture coordinates (s, t) would access the exterior region.

NVIDIA Ex1023
Page Q-286

perturbing the shade map. For the sketchy drawing in Figure 15-7a, we determined uncertainty values (off_s, off_t) by using the turbulence function (based on Perlin noise):

$off_s \leftarrow$ `turbulence(s, t);` and $off_t \leftarrow$ `turbulence(1 - s, 1 - t);`

15.3.3 Adjusting Depth

Up to now, we have merely textured a screen-aligned quad with a sketchy depiction. This method has significant shortcomings. When we render the quad textured with textures of 3D geometry, (1) z-values of the original geometry are not present in interior regions, and in particular, (2) no z-values of the original geometry are present in exterior regions when uncertainty has been applied. Thus the sketchy rendering cannot correctly interact with other objects in the scene.

To overcome these shortcomings, we apply depth sprite rendering by considering previous perturbations. That is, we additionally texture the quad with the high-precision depth map (already available) and access this texture twice using perturbed texture coordinates. As first perturbation, we apply the degree of uncertainty used for accessing the edge map; as second perturbation, we apply the degree of uncertainty used for accessing the shade map. The minimum value of both these texture values produces the final z-value of the fragment for depth testing. Figure 15-5c′ shows the result of both perturbations applied to the depth map. The interior region of the perturbed depth map matches the combination of the interior regions of both the perturbed edge map and the perturbed shade map. Even those spots produced for the shade map appear in the perturbation of the depth map.

Thus, modifying depth sprite rendering allows us to adjust z-values for the textured quad that represents a sketchy depiction of 3D geometry. So, the z-buffer remains in a correct state with respect to that geometry, and its sketchy representation can be composed with further (for example, nonsketchy) 3D geometry. The accompanying video on the book's CD illustrates this feature.

15.3.4 Variations of Sketchy Rendering

We present two variations of sketchy rendering, both of which also run in real time.

Roughened Profiles and Color Transitions

Although we sketch edges and surface colors nonuniformly, the profiles and the color transitions of a sketchy depiction are precise as if sketched with pencils on a flat surface. We roughen the profiles and color transitions to simulate different drawing tools and

NVIDIA Ex1023
Page Q-287

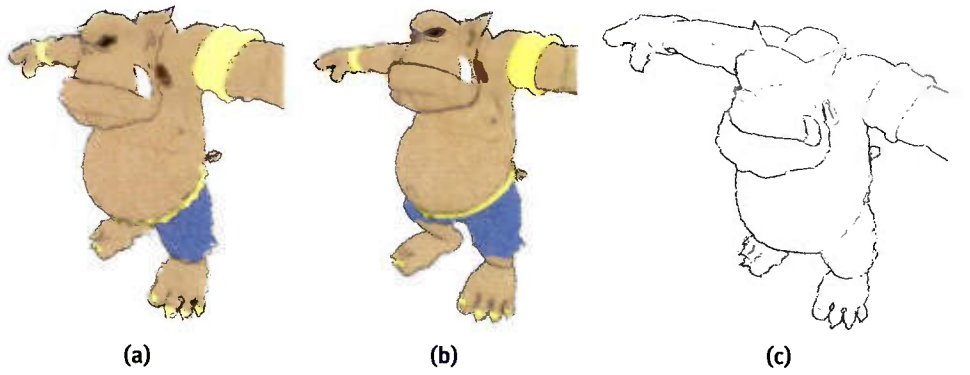


Figure 15-7. Different Styles of Sketchy Rendering
The combination of the perturbed edge, shade, and depth maps generates the final sketchy drawing in (a). Additionally, sketchy rendering can generate different styles of depiction, including (b) roughened profiles and color transitions and (c) repeated edge rendering.

media, such as chalk applied on a rough surface. We apply randomly chosen noise values; hence, adjacent texture values of the noise texture are uncorrelated. Consequently, the degrees of uncertainty that we apply to the texture coordinates of adjacent fragments are also uncorrelated. In this way, we produce sketchy depictions with softened and frayed edges and color transitions, as shown in Figure 15-7b. The roughness and granularity, in particular for edges, vary as though the pressure had varied as it does when drawing with chalk. This effect depends on the amount of uncertainty that we apply in image space.

Repeated Edges

A fundamental technique in hand drawings is to repeatedly draw edges to draft a layout or design. We sketch visually important edges only to simulate this technique. For that purpose, we exclude the shade map but apply the edge map multiple times using different degrees of uncertainty and possibly different edge colors. Edges will then overlap nonuniformly as if the edges of the 3D geometry have been sketched repeatedly, as in Figure 15-7c. We also have to adjust the depth information by accessing the depth map multiple times, using the corresponding degrees of uncertainty.

15.3.5 Controlling Uncertainty

Controlling uncertainty values, in general, enables us to configure the visual appearance of sketchy renderings. By providing uncertainty values based on a Perlin noise function

NVIDIA Ex1023
Page Q-288

for each pixel in image space, (1) we achieve frame-to-frame coherence, for instance, when interacting with the scene (because neighboring uncertainty values are correlated), and (2) we can access interior regions from beyond exterior regions and, vice versa, sketch beyond the boundary of 3D geometries. However, uncertainty values remain unchanged in image space and have no obvious correspondence with geometric properties of the original 3D geometry. Consequently, sketchy drawings tend to "swim" in image space (known as the *shower-door effect*) and we cannot predetermine their visual appearance.

To overcome these limitations, we seek to accomplish the following:

- Preserve geometric properties such as surface positions, normals, or curvature information to determine uncertainty values.
- Continue to provide uncertainty values in exterior regions, at least close to the 3D geometry.

Preserving Geometric Properties

To preserve geometric properties of 3D geometry to control uncertainty, we proceed as follows:

1. We render these geometric properties directly into a texture to generate an additional G-buffer.
2. Next we texture the screen-aligned quad with that additional texture, and then access geometric properties using texture coordinates (s, t).
3. Finally, we calculate uncertainty values based on a noise function, using the geometric properties as parameters.

Then we can use these uncertainty values to determine different degrees of uncertainty for perturbing texture coordinates resulting in (s', t'). Mathematically, we use the following function to determine the perturbed texture coordinates (s', t'):

$$f : (s, t) \rightarrow (s', t')$$
$$f(s, t) = p(s, t, g(s, t)),$$

where (s, t) represent texture coordinates of a fragment produced when rasterizing the screen-aligned quad, $g()$ provides the geometric properties available in the additional texture, and $p()$ determines the perturbation applied to (s, t) using $g()$ as input.

For sketchy rendering, we apply two functions $f(s, t)$ to handle perturbations for the edge ($f_{Edge}(s, t)$) and the shade ($f_{Shade}(s, t)$) maps differently.

NVIDIA Ex1023
Page Q-289

Enlarging the Geometry

We enlarge the original 3D geometry to generate geometric properties in its surroundings in image space. We do this by slightly shifting each vertex of the mesh along its vertex normal in object space. For this technique to work as expected, the surface must at least form a connected component and each of its shared vertices must have an interpolated normal.

By enlarging the 3D geometry in this way, we can render the geometric properties into a texture for calculating uncertainty values in interior regions as well as in the exterior regions (nearby the original 3D geometry). That way, interior regions can be sketched beyond the boundary of the 3D geometry and exterior regions can penetrate interior regions. We can then apply perturbations based on uncertainty values that do have an obvious correspondence to the underlying 3D geometry.

15.3.6 Reducing the Shower-Door Effect

We now illustrate, by way of example, how to control sketchiness to reduce the shower-door effect for sketchy rendering.

We render enlarged 3D geometry with its object-space positions as color values into a texture. To do so, we determine the object-space positions for each of the displaced vertices and provide them as texture coordinates to the rasterization process. Then the rasterizer produces interpolated object-space positions for each fragment. A specialized fragment shader then outputs them as high-precision color values to a high-precision texture. Thus, $g(s, t)$ preserves object-space positions.

Based on $g(s, t)$, we can determine texture coordinates $f(s, t)$ using $p()$. In our example, the function $p()$ calculates the perturbation by a user-defined 2×2 matrix and by a Perlin noise function encoded into a 3D texture. Then we access the 3D texture using $g(s, t)$ as texture coordinates. We multiply the resulting texture value by the 2×2 matrix to obtain a degree of uncertainty. The function $f(s, t)$ then applies the degree of uncertainty to perturb (s, t), resulting in (s', t').

Calculating $f_{Edge}(s, t)$ and $f_{Shade}(s, t)$ using different matrices results in the sketchy depiction in Figure 15-8. The accompanying video on the book's CD illustrates that the shower-door effect has been significantly reduced. The overview in Figure 15-8 illustrates the process flow of sketchy rendering by considering the underlying geometric properties.

NVIDIA Ex1023
Page Q-290

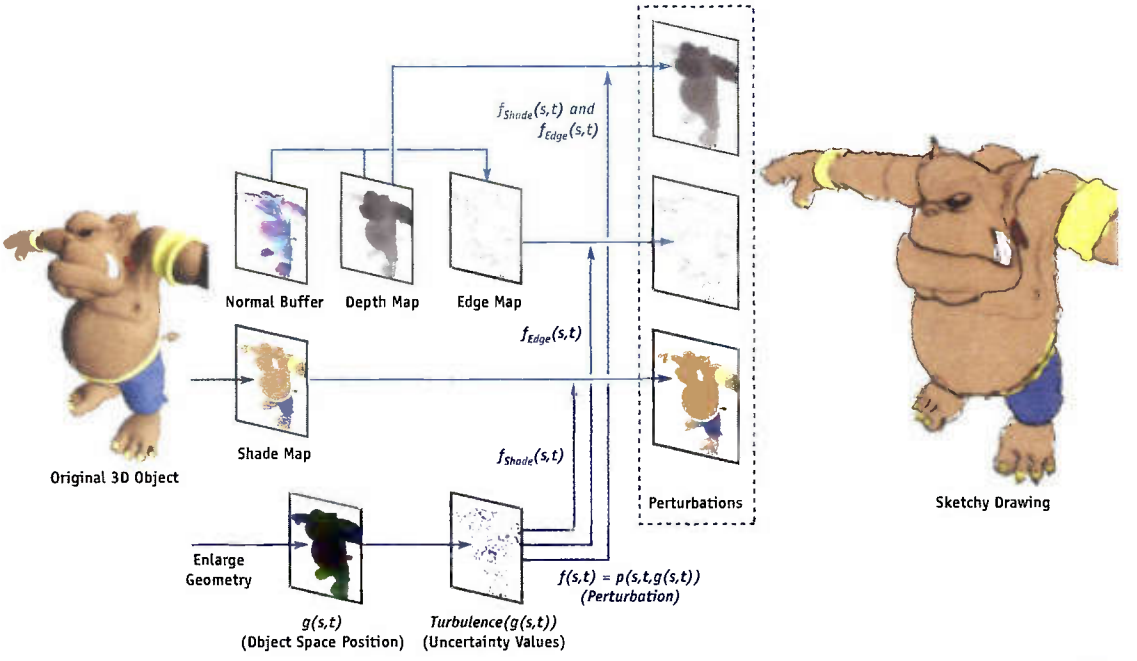


Figure 15-8. Buffers and Intermediate Rendering Results Involved in Sketchy Rendering

Although this example simply reduces the shower-door effect, it gives a clue as to how to control sketchy depictions using geometric properties. Using a higher-level shading language such as GLSL (Rost 2004) or Cg (Mark et al. 2003), one can further design and stylize sketchy drawings.

15.4 Conclusion

Blueprint rendering represents the first image-space rendering technique that renders visible and occluded edges of 3D geometries. In our future work, we aim at increasing visual perception of blueprints by combining edge maps based on techniques derived from volume rendering.

Sketchy drawing also represents the first image-space rendering technique that generates stylized edges of 3D geometries (Isenberg et al. 2003) for cartoon-like depictions. Because we produce and access the geometric properties and uncertainty values in exterior regions using intermediate rendering results, shaders written in higher-level shading languages alone are not enough to reproduce our technique for sketchy rendering. In our future work, we expect to reproduce artistically pleasing sketches.

NVIDIA Ex1023
Page Q-291

15.5 References

Diefenbach, P. J. 1996. "Pipeline Rendering: Interaction and Realism Through Hardware-Based Multi-Pass Rendering." Ph.D. thesis. University of Pennsylvania.

Everitt, C. 2001. "Interactive Order-Independent Transparency." NVIDIA technical report. Available online at **http://developer.nvidia.com/object/Interactive_Order_Transparency.html**

Isenberg, T., B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte. 2003. "A Developer's Guide to Silhouette Algorithms for Polygonal Models." *IEEE Computer Graphics and Applications* 23(4), pp. 28–37.

Kilgard, M., ed. 2004. "NVIDIA OpenGL Extension Specifications." NVIDIA Corporation. Available online at **http://www.nvidia.com/dev_content/nvopenglspecs/nvOpenGLspecs.pdf**

Mark, W. R., R. S. Glanville, K. Akeley, and M. J. Kilgard. 2003. "Cg: A System for Programming Graphics Hardware in a C-like Language." *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)* 22(3), pp. 896–907.

Nienhaus, M., and J. Döllner. 2003. "Edge-Enhancement: An Algorithm for Real-Time Non-Photorealistic Rendering." *Journal of WSCG '03*, pp. 346–353.

Perlin, K. 1985. "An Image Synthesizer." In *Computer Graphics (Proceedings of SIGGRAPH 85)* 19(3), pp. 287–296.

Rost, R. J. 2004. *OpenGL Shading Language.* Addison-Wesley.

Saito, T., and T. Takahashi. 1990. "Comprehensible Rendering of 3-D Shapes." In *Computer Graphics (Proceedings of SIGGRAPH 90)* 24(4), August 1990, pp. 197–206.

The authors want to thank ART+COM, Berlin, for providing the model of the Temple of Ramses II; Amy and Bruce Gooch for the crank model; and Konstantin Baumann, Johannes Bohnet, Henrik Buchholz, Oliver Kersting, Florian Kirsch, and Stefan Maass for their contributions to the Virtual Rendering System.

NVIDIA Ex1023
Page Q-292



Chapter 16

Accurate Atmospheric Scattering

Sean O'Neil

16.1 Introduction

Generating realistic atmospheric scattering for computer graphics has always been a difficult problem, but it is very important for rendering realistic outdoor environments. The equations that describe atmospheric scattering are so complex that entire books have been dedicated to the subject. Computer graphics models generally use simplified equations, and very few of them run at interactive frame rates. This chapter explains how to implement a real-time atmospheric scattering algorithm entirely on the GPU using the methods described in Nishita et al. 1993. Figure 16-1 shows screenshots from the scattering demo included on this book's CD.

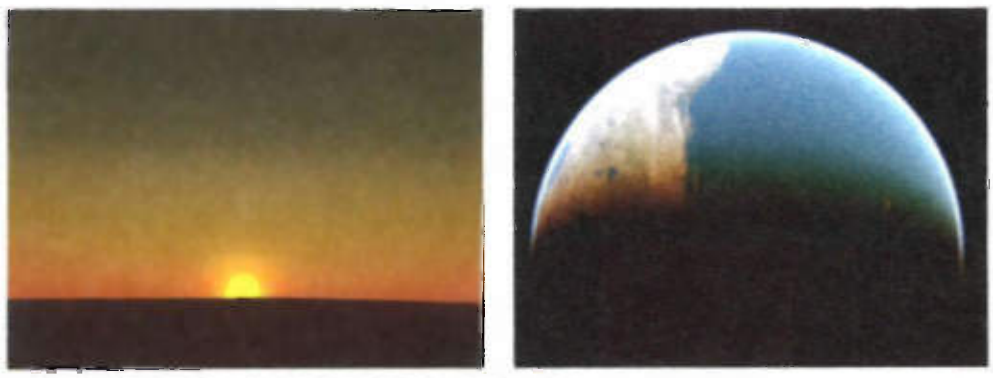


Figure 16-1. Screenshots from the Scattering Demo

NVIDIA Ex1023
Page Q-293

Many atmospheric scattering models assume that the camera is always on or very close to the ground. This makes it easier to assume that the atmosphere has a constant density at all altitudes, which simplifies the scattering equations in Nishita et al. 1993 tremendously. Refer to Hoffman and Preetham 2002 for an explanation of how to implement these simplified equations in a GPU shader. This implementation produces an attractive scattering effect that is very fast on DirectX 8.0 shaders. Unfortunately, it doesn't always produce very accurate results, and it doesn't work well for a flight or space simulator, in which the camera can be located in space or very high above the ground.

This chapter explains how to implement the full equations from Nishita et al. 1993 in a GPU shader that runs at interactive frame rates. These equations model the atmosphere more accurately, with density falling off exponentially as altitude increases. O'Neil 2004 describes a similar algorithm that ran on the CPU, but that algorithm was too CPU-intensive. It was based on a precalculated 2D lookup table with four floating-point channels. Calculating the color for each vertex required several lookups into the table, with extra calculations around each lookup. At the time that article was written, no GPU could support such operations in a shader in one pass.

In this chapter, we eliminate the lookup table without sacrificing image quality, allowing us to implement the entire algorithm in a GPU shader. These shaders are small and fast enough to run in real time on most GPUs that support DirectX Shader Model 2.0.

16.2 Solving the Scattering Equations

The scattering equations have nested integrals that are impossible to solve analytically; fortunately, it's easy to numerically compute the value of an integral with techniques such as the trapezoid rule. Approximating an integral in this manner boils down to a weighted sum calculated in a loop. Imagine a line segment on a graph: break up the segment into n sample segments and evaluate the integrand at the center point of each sample segment. Multiply each result by the length of the sample segment and add them all up. Taking more samples makes the result more accurate, but it also makes the integral more expensive to calculate.

In our case, the line segment is a ray from the camera through the atmosphere to a vertex. This vertex can be part of the terrain, part of the sky dome, part of a cloud, or even part of an object in space such as the moon. If the ray passes through the atmosphere to get to the vertex, scattering needs to be calculated. Every ray should have two points defined that mark where the ray starts passing through the atmosphere and

NVIDIA Ex1023
Page Q-294

where it stops passing through the atmosphere. We'll call these points A and B, and they are shown in Figure 16-2. When the camera is inside the atmosphere, A is the camera's position. When the vertex is inside the atmosphere, B is the vertex's position. When either point is in space, we perform a sphere-intersection check to find out where the ray intersects the outer atmosphere, and then we make the intersection point A or B.

Now we have a line segment defined from point A to point B, and we want to approximate the integral that describes the atmospheric scattering across its length. For now let's take five sample positions and name their points P_1 through P_5. Each point P_1 through P_5 represents a point in the atmosphere at which light scatters; light comes into the atmosphere from the sun, scatters at that point, and is reflected toward the camera. Consider the point P_5, for example. Sunlight goes directly from the sun to P_5 in a straight line. Along that line the atmosphere scatters some of the light away from P_5. At P_5, some of this light is scattered directly toward the camera. As the light from P_5 travels to the camera, some of it gets scattered away again.

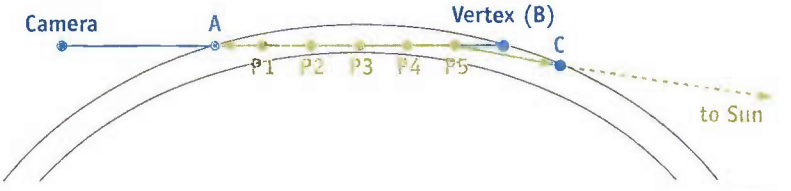


Figure 16-2. The Geometry of Atmospheric Scattering
The blue line represents the ray from the camera to the vertex. The green lines represent the rays from P_5 to the sun and to the camera.

16.2.1 Rayleigh Scattering vs. Mie Scattering

Another important detail is related to how the light scattering at the point P is modeled. Different particles in the atmosphere scatter light in different ways. The two most common forms of scattering in the atmosphere are *Rayleigh scattering* and *Mie scattering*. Rayleigh scattering is caused by small molecules in the air, and it scatters light more heavily at the shorter wavelengths (blue first, then green, and then red). The sky is blue because the blue light bounces all over the place, and ultimately reaches your eyes from every direction. The sun's light turns yellow/orange/red at sunset because as light travels far through the atmosphere, almost all of the blue and much of the green light is scattered away before it reaches you, leaving just the reddish colors.

NVIDIA Ex1023
Page Q-295

Mie scattering is caused by larger particles in the air called aerosols (such as dust and pollution), and it tends to scatter all wavelengths of light equally. On a hazy day, Mie scattering causes the sky to look a bit gray and causes the sun to have a large white halo around it. Mie scattering can also be used to simulate light scattered from small particles of water and ice in the air, to produce effects like rainbows, but that is beyond the scope of this chapter. (Refer to Brewer 2004 for more information.)

16.2.2 The Phase Function

$$F(\theta, g) = \frac{3 \times (1 - g^2)}{2 \times (2 + g^2)} \times \frac{1 + \cos^2 \theta}{(1 + g^2 - 2 \times g \times \cos \theta)^{\frac{3}{2}}}$$

The phase function describes how much light is scattered toward the direction of the camera based on the angle θ (the angle between the two green rays in Figure 16-2) and a constant g that affects the symmetry of the scattering. There are many different versions of the phase function. This one is an adaptation of the Henyey-Greenstein function used in Nishita et al. 1993.

Rayleigh scattering can be approximated by setting g to 0, which greatly simplifies the equation and makes it symmetrical for positive and negative angles. Negative values of g scatter more light in a forward direction, and positive values of g scatter more light back toward the light source. For Mie aerosol scattering, g is usually set between -0.75 and -0.999. Never set g to 1 or -1, as it makes the equation reduce to 0.

16.2.3 The Out-Scattering Equation

$$t(P_a P_b, \lambda) = 4\pi \times K(\lambda) \times \int_{P_a}^{P_b} \exp\left(\frac{-h}{H_0}\right) ds$$

The out-scattering equation is the inner integral mentioned earlier. The integral part determines the "optical depth," or the average atmospheric density across the ray from point P_a to point P_b multiplied by the length of the ray. This is also called the "optical length" or "optical thickness." Think of it as a weighting factor based on how many air particles are in the path of the light along the ray. The rest of the equation is made up of constants, and they determine how much of the light those particles scatter away from the ray.

NVIDIA Ex1023
Page Q-296

To compute the value of this integral, the ray from P_a to P_b will be broken up into segments and the exponential term will be evaluated at each sample point. The variable h is the height of the sample point. In my implementation, the height is scaled so that 0 represents sea level and 1 is at the top of the atmosphere. In theory, the atmosphere has no fixed top, but for practical purposes, we have to choose some height at which to render the sky dome. H_0 is the scale height, which is the height at which the atmosphere's average density is found. My implementation uses 0.25, so the average density is found 25 percent of the way up from the ground to the sky dome.

The constant λ is the wavelength (or color) of light and $K(\lambda)$ is the scattering constant, which is dependent on the density of the atmosphere at sea level. Rayleigh and Mie scattering each have their own scattering constants, including the scale height (H_0). Rayleigh and Mie scattering also differ in how they depend on wavelength. The Rayleigh scattering constant is usually divided by λ^4. In most computer graphics models, the Mie scattering constant is not dependent on wavelength, but at least one implementation divides it by $\lambda^{0.84}$. Wherever this equation depends on wavelength, it must be solved separately for each of the three color channels and separately for each type of scattering.

16.2.4 The In-Scattering Equation

$$I_v(\lambda) = I_s(\lambda) \times K(\lambda) \times F(\theta, g) \times \int_{P_a}^{P_b} \left(\exp\left(\frac{-h}{H_0}\right) \times \exp\left(-t(PP_c, \lambda) - t(PP_a, \lambda)\right) \right) ds$$

The in-scattering equation describes how much light is added to a ray through the atmosphere due to light scattering from the sun. For each point P along the ray from P_a to P_b, PP_c is the ray from the point to the sun and PP_a is the ray from the sample point to the camera. The out-scattering function determines how much light is scattered away along the two green rays in Figure 16-2. The remaining light is scaled by the phase function, the scattering constant, and the intensity of the sunlight, $I_s(\lambda)$. The sunlight intensity does not have to be dependent on wavelength, but this is where you would apply the color if you wanted to create an alien world revolving around a purple star.

16.2.5 The Surface-Scattering Equation

$$I'_v(\lambda) = I_v(\lambda) + I_e(\lambda) \times \exp\left(-t(P_aP_b, \lambda)\right)$$

To scatter light reflected from a surface, such as the surface of a planet, you must take into account the fact that some of the reflected light will be scattered away on its way to

NVIDIA Ex1023
Page Q-297

the camera. In addition, extra light is scattered in from the atmosphere. $I_e(\lambda)$ is the amount of light emitted or reflected from a surface, and it is attenuated by an out-scattering factor. The sky is not a surface that can reflect or emit light, so only $I_v(\lambda)$ is needed to render the sky. Determining how much light is reflected or emitted by a surface is application-specific, but for reflected sunlight, you need to account for the out-scattering that takes place before the sunlight strikes the surface (that is, $I_s(\lambda) \times \exp(-t(P_c P_b, \lambda))$), and use that as the color of the light when determining how much light the surface reflects.

16.3 Making It Real-Time

Let's find out how poorly these equations will perform if they're implemented as explained in the preceding section, with five sample points for the in-scattering equation and five sample points for each of the integrals to compute the out-scattering equations. This gives $5 \times (5 + 5)$ samples at which to evaluate the functions for each vertex. We also have two types of scattering, Rayleigh and Mie, and we have to calculate each one for the different wavelengths of each of the three color channels (RGB). So now we're up to approximately $2 \times 3 \times 5 \times (5 + 5)$, or 300 computations per vertex. To make matters worse, the visual quality suffers noticeably when using only five samples for each integral. O'Neil 2004 used 50 samples for the inner integrals and five for the outer integral, which pushes the number of calculations up to 3,000 per vertex!

We don't need to go any further to know that this will not run very fast. Nishita et al. 1993 used a precalculated 2D lookup table to cut the number of calculations in half, but it still won't run in real time. Their lookup table took advantage of the fact that the sun is so far away that its rays can be considered parallel. (This idea is reflected in the in-scattering equation, in Section 16.2.4.) This makes it possible to calculate a lookup table that contains the amount of out-scattering for rays going from the sun to any point in the atmosphere. This table replaces one of the out-scattering integrals with a lookup table whose variables are altitude and angle to the sun. Because the rays to the camera are not parallel, the out-scattering integral for camera rays still had to be solved at runtime.

In O'Neil 2004, I proposed a better 2D lookup table that allows us to avoid both out-scattering integrals. The first dimension, x, takes a sample point at a specific altitude above the planet, with 0.0 being on the ground and 1.0 being at the top of the atmosphere. The second dimension, y, represents a vertical angle, with 0.0 being straight up and 1.0 being straight down. At each (x, y) pair in the table, a ray is fired from a point at

NVIDIA Ex1023
Page Q-298

altitude x to the top of the atmosphere along angle y. The lookup table had four channels, two reserved for Rayleigh scattering and two reserved for Mie scattering. One channel for each simply contained the atmospheric density at that altitude, or $\exp(-h/H_0)$. The other channel contained the optical depth of the ray just described. Because the lookup table was precomputed, I chose to use 50 samples to approximate the optical depth integral, resulting in very good accuracy.

As with the lookup table proposed in Nishita et al. 1993, we can get the optical depth for the ray to the sun from any sample point in the atmosphere. All we need is the height of the sample point (x) and the angle from vertical to the sun (y), and we look up (x, y) in the table. This eliminates the need to calculate one of the out-scattering integrals. In addition, the optical depth for the ray to the camera can be figured out in the same way, right? Well, almost. It works the same way when the camera is in space, but not when the camera is in the atmosphere. That's because the sample rays used in the lookup table go from some point at height x all the way to the top of the atmosphere. They don't stop at some point in the middle of the atmosphere, as they would need to when the camera is inside the atmosphere.

Fortunately, the solution to this is very simple. First we do a lookup from sample point P to the camera to get the optical depth of the ray passing through the camera to the top of the atmosphere. Then we do a second lookup for the same ray, but starting at the camera instead of starting at P. This will give us the optical depth for the part of the ray that we don't want, and we can subtract it from the result of the first lookup. Examine the rays starting from the ground vertex (B_1) in Figure 16-3 for a graphical representation of this.

There's only one problem left. When a vertex is above the camera in the atmosphere, the ray from sample point P through the camera can pass through the planet itself. The height variable is not expected to go so far negative, and it can cause the numbers in the lookup table to go extremely high, losing precision and sometimes even encountering a floating-point overflow. The way to avoid this problem is to reverse the direction of the rays when the vertex is above the camera. Examine the rays passing through the sky vertex (B_2) in Figure 16-3 for a graphical representation of this.

So now we've reduced 3,000 calculations per vertex to something closer to $2 \times 3 \times 5 \times (1 + 1)$, or 60 (assuming five samples for the out-scattering integral for the eye ray). Implemented in software on an Athlon 2500+ with an inefficient brute-force rendering method, I was able to get this method to run between 50 and 100 frames per second.

NVIDIA Ex1023
Page Q-299



Figure 16-3. Problems with the Improved Lookup Table
This figure illustrates two issues with the lookup table when the camera is in the atmosphere. The lookup table can only return the optical depth for the blue and orange rays indicated. To get the optical depth from either sample point to the camera, the optical depth of each blue ray must be subtracted from the optical depth of its respective orange ray.

16.4 Squeezing It into a Shader

At this point, I felt that this algorithm had been squeezed about as much as it could, but I knew that I had to squeeze it even smaller to fit it into a shader. I didn't want this algorithm to require Shader Model 3.0, so having lookup tables in textures used by the vertex shaders wasn't possible. I decided to take a different approach and started to mathematically analyze the results of the lookup table. Even though I knew I couldn't come up with a way to simplify the integral equations, I had hoped that I might be able to simulate the results closely enough with a completely different equation.

16.4.1 Eliminating One Dimension

I started by plotting the results of the lookup table on a graph. I plotted height from 0.0 to 1.0 along the x axis and the lookup table result (optical depth) along the y axis. For various angles sampled from 0 to 1, I plotted a separate line on the graph. I noticed right away that each line dropped off exponentially as x went from 0 to 1, but the scale of each line on the graph varied dramatically. This makes sense, because as the angle of the ray goes from straight up to straight down, the length of the ray increases dramatically.

To make it easier to compare the shapes of the curves of each line, I decided to normalize them. I took the optical depth value at $x = 0$ (or height = 0) for each line and divided all of the values on that line by that value. This scaled all lines on the graph to start at $(x = 0, y = 1)$ and work their way down toward $(x = 1, y = 0)$. To my surprise, almost all of the normalized lines fell right on top of each other on the graph! The curve was almost exactly the same for every one, and that curve turned out to be $\exp(-4x)$.

NVIDIA Ex1023
Page Q-300

This makes some sense, because the optical depth equation is the integral of $\exp(-h/H_0)$. I chose H_0 to be 0.25, so $\exp(-4h)$ is a common factor. It is still a bit puzzling, however, as the h inside the integral is not the same as the height x outside the integral, which is only the height at the start of the ray. The h value inside the integral does not vary linearly, and it has more to do with how it passes through a spherical space than with the starting height. There is some variation in the lines, and the variation increases as the angle increases. The variation gets worse exponentially as the angle increases over 90 degrees. Because we don't care about angles that are much larger than 90 degrees (because the ray passes through the planet), $\exp(-4x)$ works very well for eliminating the x axis of the lookup table.

16.4.2 Eliminating the Other Dimension

Now that the x dimension (height) of the lookup table is being handled by $\exp(-4x)$, we need to eliminate the y dimension (angle). The only part of the optical depth that is not handled by $\exp(-4x)$ is the scale used to normalize the lines on the graph explained previously, which is the value of the optical depth at $x = 0$. So now we create a new graph by plotting the angle from 0 to 1 on the x axis and the scale of each angle on the y axis. For lack of a better name, I call this the *scale function*.

The first time I looked at the scale function, I noticed that it started at 0.25 (the scale height) and increased on some sort of accelerating curve. Thinking that it might be exponential, I divided the scales by the scale height (to make the graph start at 1) and took the natural logarithm of the result. The result was another accelerating curve that I didn't recognize. I tried a number of curves, but nothing fit well on all parts of the curve. I ended up using graphical analysis software to find a "best fit" equation for the curve, and it came back with a polynomial equation that was not pretty but fit the values well.

One significant drawback to this implementation is that the scale function is dependent on the scale height and the ratio between the atmosphere's thickness and the planet's radius. If either value changes, you need to calculate a new scale function. In the demo included on this book's CD, the atmosphere's thickness (the distance from the ground to the top of the atmosphere) is 2.5 percent of the planet's radius, and the scale height is 25 percent of the atmosphere's thickness. The radius of the planet doesn't matter as long as those two values stay the same.

NVIDIA Ex1023
Page Q-301

16.5 Implementing the Scattering Shaders

Now that the problem has been solved mathematically, let's look at how the demo was implemented. The C++ code for the demo is fairly simple. The `gluSphere()` function is called to render both the ground and the sky dome. The front faces are reversed for the sky dome so that the inside of its sphere is rendered. It uses a simple rectangular Earth texture to make it possible to see how the ground scattering affects colors on the ground, and it uses a simple glow texture billboard to render the moon. No distinct sun is rendered, but the Mie scattering creates a glow in the sky dome that looks like the sun (only when seen through the atmosphere).

I have provided shader implementations in both Cg and GLSL on the book's CD. The ground, the sky, and objects in space each have two scattering shaders, one for when the camera is in space and one for when the camera is in the atmosphere (this avoids conditional branching in the shaders). The ground shaders can be used for the terrain, as well as for objects that are beneath the camera. The sky shaders can be used for the sky dome, as well as for objects that are above the camera. The space shaders can be used for any objects outside the atmosphere, such as the moon.

The naming convention for the shaders is "*render_object*From*camera_position*". So the SkyFromSpace shader is used to render the sky dome when the camera is in space. There is also a common shader that contains some common constants and helper functions used throughout the shaders. Let's use SkyFromSpace as an example.

16.5.1 The Vertex Shader

As you can see in Listing 16-1, SkyFromSpace.vert is a fairly complex vertex shader, but hopefully it's easy enough to follow with the comments in the code and the explanations provided here. `Kr` is the Rayleigh scattering constant, `Km` is the Mie scattering constant, and `ESun` is the brightness of the sun. Rayleigh scatters different wavelengths of light at different rates, and the ratio is `1/pow(wavelength, 4)`. Referring back to Figure 16-2, `v3Start` is point *A* from the previous examples and `v3Start + fFar * v3Ray` is point *B*. The variable `v3SamplePoint` goes from P_1 to P_n with each iteration of the loop.

The variable `fStartOffset` is actually the value of the lookup table from point *A* going toward the camera. Why would we need to calculate this when it's at the outer edge of the atmosphere? Because the density is not truly zero at the outer edge. The density falls off exponentially and it is close to zero, but if we do not calculate this value and use it as an offset, there may be a visible "jump" in color when the camera enters the atmosphere.

NVIDIA Ex1023
Page Q-302

You may have noticed that the phase function is missing from this shader. The phase function depends on the angle toward the light source, and it suffers from tessellation artifacts if it is calculated per vertex. To avoid these artifacts, the phase function is implemented in the fragment shader.

Listing 16-1. SkyFromSpace.vert, Which Renders the Sky Dome When the Camera Is in Space

```
#include "Common.cg"

vertout main(float4 gl_Vertex : POSITION,
  uniform float4x4 gl_ModelViewProjectionMatrix,
  uniform float3 v3CameraPos,      // The camera's current position
  uniform float3 v3LightDir,       // Direction vector to the light source
  uniform float3 v3InvWavelength, // 1 / pow(wavelength, 4) for RGB
  uniform float fCameraHeight,     // The camera's current height
  uniform float fCameraHeight2,    // fCameraHeight^2
  uniform float fOuterRadius,      // The outer (atmosphere) radius
  uniform float fOuterRadius2,     // fOuterRadius^2
  uniform float fInnerRadius,      // The inner (planetary) radius
  uniform float fInnerRadius2,     // fInnerRadius^2
  uniform float fKrESun,           // Kr * ESun
  uniform float fKmESun,           // Km * ESun
  uniform float fKr4PI,            // Kr * 4 * PI
  uniform float fKm4PI,            // Km * 4 * PI
  uniform float fScale,            // 1 / (fOuterRadius - fInnerRadius)
  uniform float fScaleOverScaleDepth) // fScale / fScaleDepth
{
  // Get the ray from the camera to the vertex and its length (which
  // is the far point of the ray passing through the atmosphere)
  float3 v3Pos = gl_Vertex.xyz;
  float3 v3Ray = v3Pos - v3CameraPos;
  float fFar = length(v3Ray);
  v3Ray /= fFar;

  // Calculate the closest intersection of the ray with
  // the outer atmosphere (point A in Figure 16-3)
  float fNear = getNearIntersection(v3CameraPos, v3Ray, fCameraHeight2,
                                    fOuterRadius2);

  // Calculate the ray's start and end positions in the atmosphere,
  // then calculate its scattering offset
  float3 v3Start = v3CameraPos + v3Ray * fNear;
  fFar -= fNear;
```


NVIDIA Ex1023
Page Q-303

Listing 16-1 *(continued)*. SkyFromSpace.vert, Which Renders the Sky Dome When the Camera Is in Space

```
    float fStartAngle = dot(v3Ray, v3Start) / fOuterRadius;
    float fStartDepth = exp(-fInvScaleDepth);
    float fStartOffset = fStartDepth * scale(fStartAngle);

    // Initialize the scattering loop variables
    float fSampleLength = fFar / fSamples;
    float fScaledLength = fSampleLength * fScale;
    float3 v3SampleRay = v3Ray * fSampleLength;
    float3 v3SamplePoint = v3Start + v3SampleRay * 0.5;

    // Now loop through the sample points
    float3 v3FrontColor = float3(0.0, 0.0, 0.0);
    for(int i=0; i<nSamples; i++) {
      float fHeight = length(v3SamplePoint);
      float fDepth = exp(fScaleOverScaleDepth * (fInnerRadius - fHeight));
      float fLightAngle = dot(v3LightDir, v3SamplePoint) / fHeight;
      float fCameraAngle = dot(v3Ray, v3SamplePoint) / fHeight;
      float fScatter = (fStartOffset + fDepth * (scale(fLightAngle) -
                                                 scale(fCameraAngle)));
      float3 v3Attenuate = exp(-fScatter *
                               (v3InvWavelength * fKr4PI + fKm4PI));
      v3FrontColor += v3Attenuate * (fDepth * fScaledLength);
      v3SamplePoint += v3SampleRay;
    }

    // Finally, scale the Mie and Rayleigh colors
    vertout OUT;
    OUT.pos = mul(gl_ModelViewProjectionMatrix, gl_Vertex);
    OUT.c0.rgb = v3FrontColor * (v3InvWavelength * fKrESun);
    OUT.c1.rgb = v3FrontColor * fKmESun;
    OUT.t0 = v3CameraPos - v3Pos;
    return OUT;
}
```

16.5.2 The Fragment Shader

This shader should be fairly self-explanatory. One thing that is not shown in Listing 16-2 is that I commented out the math in the `getRayleighPhase()` function. The Rayleigh phase function causes the blue sky to be the brightest at 0 degrees and 180 degrees and the darkest at 90 degrees. However, I feel it makes the sky look too dark

NVIDIA Ex1023
Page Q-304

around 90 degrees. In theory, this problem arises because we are not implementing multiple scattering. Single scattering only calculates how much light is scattered coming directly from the sun. Light that is scattered out of the path of a ray just vanishes. Multiple scattering attempts to figure out where that light went, and often it is redirected back toward the camera from another angle. It is a lot like radiosity lighting, and it is not considered feasible to calculate in real time. Nishita et al. 1993 uses an ambient factor to brighten up the darker areas. I feel that it looks better to just leave out the Rayleigh phase function entirely. Feel free to play with it and see what you like best.

Listing 16-2. SkyFromSpace.frag, Which Renders the Sky Dome When the Camera Is in Space

```
#include "Common.cg"

float4 main(float4 c0 : COLOR0,
  float4 c1 : COLOR1,
  float3 v3Direction : TEXCOORD0,
  uniform float3 v3LightDirection,
  uniform float g,
  uniform float g2) : COLOR
{
  float fCos = dot(v3LightDirection, v3Direction) / length(v3Direction);
  float fCos2 = fCos * fCos;
  float4 color = getRayleighPhase(fCos2) * c0 +
                 getMiePhase(fCos, fCos2, g, g2) * c1;
  color.a = color.b;
  return color;
}
```

16.6 Adding High-Dynamic-Range Rendering

Atmospheric scattering doesn't look very good without high-dynamic-range (HDR) rendering. This is because these equations can very easily generate images that are too bright or too dark. See Figure 16-4. It looks much better when you render the image to a floating-point buffer and scale the colors to fall within the range 0..1 using an exponential curve with an adjustable exposure constant.

The exposure equation used in this demo is very simple: $1.0 - \exp(-fExposure \times color)$. The exposure constant works like the aperture of a camera or the pupils of your eyes. When there is too much light, the exposure constant is reduced to let in less light. When there is not enough light, the exposure constant is increased to let in more light.

NVIDIA Ex1023
Page Q-305

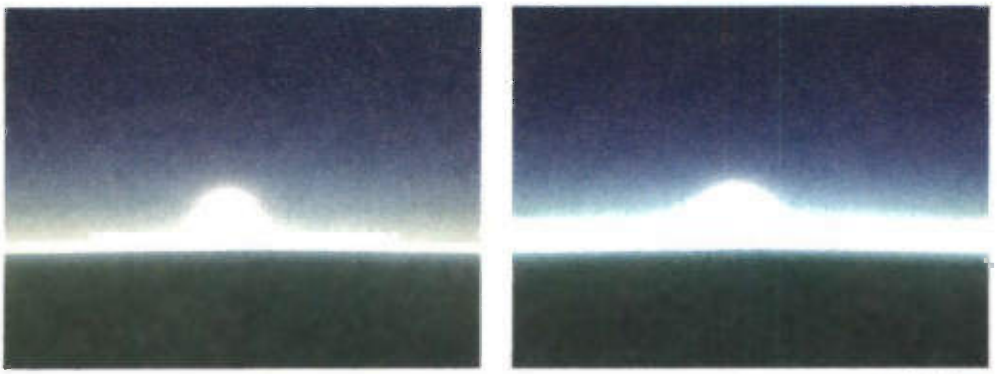


Figure 16-4. The Importance of High Dynamic Range
The same scene rendered with HDR (left) and without HDR (right). Notice the extreme bright and dark regions in the image without HDR.

Without reasonable limits set on the exposure constant, daylight can look like moonlight and vice versa. Regardless, the relative brightness of each color is always preserved.

The HDR rendering implemented in this demo is also very simple. An OpenGL pbuffer is created with a floating-point pixel format and render-to-texture enabled. The same rendering code as before is used, but everything is rendered to the pbuffer's rendering context. Afterward, the main rendering context is selected and a single quad is rendered to fill the screen. The floating-point buffer is selected as a texture, and a simple exponential scaling shader is used to scale the colors to the normal 0–255 range. The exposure constant is set manually and can be changed at runtime in the demo, along with several of the scattering constants used.

Fortunately, modern GPUs support floating-point render targets. Some critical features such as alpha blending aren't available until you move up to GeForce 6 Series GPUs, so blending the sky dome with objects rendered in space won't work for older GPUs.

16.7 Conclusion

Although it took a fair number of simplifications to be able to interactively render scenes with the model described, the demo looks pretty sharp, especially with HDR rendering. Tweaking the parameters to the system can be a source of much entertainment; I had never imagined that the sky at the horizon would be orange at high noon if our atmos-

NVIDIA Ex1023
Page Q-306

phere were a little thicker, or that sunsets would look that much more incredible. Red and orange skies produce sunsets of different shades of blue, and purple skies produce green sunsets. Once I even managed to find settings that produced a rainbow sunset.

In any case, the work is not done here. There are a number of improvements that could be made to this algorithm:

- Create an atmospheric-density scale function that is not hard-coded to a specific scale height and the ratio between the atmosphere thickness and the planetary radius.
- Split up the Rayleigh and Mie scattering so that they can have different scale depths.
- Provide a better way to simulate multiple scattering and start using the Rayleigh phase function again.
- Allow light from other sources, such as the moon, to be scattered, as well as sunlight. This would be necessary to create a moonlight effect with a halo surrounding the moon.
- Atmospheric scattering also makes distant objects look larger and blurrier than they are. The moon would look much smaller to us if it were not viewed through an atmosphere. It may be possible to achieve this in the HDR pass with some sort of blurring filter that is based on optical depth.
- Change the sample points taken along the ray from linear to exponential based on altitude. Nishita et al. 1993 explains why this is beneficial.
- Add other atmospheric effects, such as clouds, precipitation, lightning, and rainbows. See Harris 2001, Brewer 2004, and Dobashi et al. 2001.

16.8 References

Brewer, Clint. 2004. "Rainbows and Fogbows: Adding Natural Phenomena." NVIDIA Corporation. SDK white paper.
http://download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/src/HLSL_RainbowFogbow/docs/RainbowFogbow.pdf

Dobashi, Y., T. Yamamoto, and T. Nishita. 2001. "Efficient Rendering of Lightning Taking into Account Scattering Effects due to Clouds and Atmospheric Particles."
http://nis-lab.is.s.u-tokyo.ac.jp/~nis/cdrom/pg/pg01_lightning.pdf

NVIDIA Ex1023
Page Q-307

Harris, Mark J., and Anselmo Lastra. 2001. "Real-Time Cloud Rendering." *Eurographics 2001* 20(3), pp. 76–84.
http://www.markmark.net/PDFs/RTClouds_HarrisEG2001.pdf

Hoffman, N., and A. J. Preetham. 2002. "Rendering Outdoor Scattering in Real Time." ATI Corporation.
http://www.ati.com/developer/dx9/ATI-LightScattering.pdf

Nishita, T., T. Sirai, K. Tadamura, and E. Nakamae. 1993. "Display of the Earth Taking into Account Atmospheric Scattering." In *Proceedings of SIGGRAPH 93,* pp. 175–182.
http://nis-lab.is.s.u-tokyo.ac.jp/~nis/cdrom/sig93_nis.pdf

O'Neil, Sean. 2004. "Real-Time Atmospheric Scattering." GameDev.net.
http://www.gamedev.net/columns/hardcore/atmscattering/

NVIDIA Ex1023
Page Q-308



Chapter 17

Efficient Soft-Edged Shadows Using Pixel Shader Branching

Yury Uralsky
NVIDIA Corporation

Rendering realistic shadows has always been a hard problem in computer graphics. Soft shadows present an even greater challenge to developers, even when real-time performance is not required. Yet shadows are important visual clues that help us to establish spatial relationships between objects in a scene. In the past, artists understood the importance of shadows and used them to their advantage to emphasize certain objects and add depth to their pictures.

Real-world shadows always appear to have a certain amount of blurriness across their edges, and never appear strictly hard. Even on a bright, sunny day, when shadows are the sharpest, this is a noticeable effect. Even the sun, being a huge distance away from the Earth, can't be considered a point light source. Additionally, atmospheric scattering diffuses incoming light, blurring shadows from sunlight even more.

Recent advances in graphics hardware functionality and performance finally enable us to achieve real-time speeds when rendering soft shadows. This chapter discusses a method for rendering a high-quality approximation to real soft shadows at interactive frame rates. The method is based on adaptively taking multiple shadow-map samples in the fragment shader using a technique called *percentage-closer filtering* (PCF). With the branching capabilities of fragment shaders, this adaptive sampling approach provides increased visual quality while still maintaining high performance, compared to always taking a fixed number of samples.

Introduction

NVIDIA Ex1023
Page Q-309

17.1 Current Shadowing Techniques

The most well-known and robust techniques for rendering shadows are stencil shadow volumes and shadow maps. Both methods have their advantages and disadvantages.

Stencil shadow volumes work by classifying points on the scene surfaces with respect to frusta that encompass space where lighting is blocked by occluders. Sides of these frusta are formed by planes containing light-source position and edges in an object silhouette, as observed from the light source's point of view. The sides of the shadow volumes are rendered into a special off-screen buffer, "tagging" pixels in shadow. After that, normal scene rendering can color each pixel according to its tag value.

This method is simple and elegant, and it requires only minimal hardware support. It works well with both directional and point light sources, and it can be integrated pretty easily into any polygon-based rendering system such that every surface will be shadowed properly.

On the other hand, stencil shadow volumes depend heavily on how complex the geometry of a shadowing object is, and they burn a lot of fill rate. Most current implementations of the stencil shadow volumes algorithm require quite a bit of work on the CPU side, though that situation is likely to change in the future. Another downside of stencil shadow volumes is that the method is inherently multipass, which further restricts the polygon count of scenes with which it can be used.

Shadow maps are essentially z-buffers rendered as viewed from the light source's perspective. The entire scene is rendered into the shadow map in a separate pass. When the scene itself is rendered, the position of each pixel is transformed to light space, and then its z-value is compared to the value stored in the shadow map. The result of this comparison is then used to decide whether the current pixel is in shadow or not. Shadow mapping is an image-space algorithm, and it suffers from aliasing problems due to finite shadow-map resolution and resampling issues. Imagine we are looking at a large triangle positioned edge-on with respect to a light source such that the shadow stretches over it. As you can guess, its projection will cover only a handful of texels in the shadow map, failing to reproduce the shadow boundary with decent resolution. Another disadvantage of shadow maps is that they do not work very well for omnidirectional light sources, requiring multiple rendering passes to generate several shadow maps and cover the entire sphere of directions.

NVIDIA Ex1023
Page Q-310

When their disadvantages can be avoided or minimized, shadow maps are great. Shadow maps are fully orthogonal to the rest of the rendering pipeline: everything that can be rendered and that can produce a z-value can be shadowed. Shadow maps scale very well as the triangle count increases, and they can even work with objects that have partially transparent textures. These properties have made this algorithm the method of choice for offline rendering.

Neither of these algorithms is able to produce shadows with soft edges "out of the box." Extensions to these algorithms exist that allow rendering of soft-edged shadows, but they rely on sampling the surface of the area light source and performing multiple shadow-resolution passes for each sample position, accumulating the results. This increases the cost of the shadows dramatically, and it produces visible color bands in the penumbrae of the shadows if the number of passes is not high enough.

However, it is possible to efficiently approximate soft shadows by modifying the original shadow-mapping algorithm. The percentage-closer filtering technique described by Reeves et al. (1987) takes a large number of shadow-map samples around the current point being shaded and filters their results to produce convincing soft shadows.

17.2 Soft Shadows with a Single Shadow Map

With PCF, instead of sampling the light source and rendering shadow maps at each sample position, we can get away with a *single* shadow map, sampling it several times in the area around the projected point in the shadow map, and then combining the results. This method does not produce physically correct soft shadows, but the visual effect is similar to what we see in reality.

17.2.1 Blurring Hard-Edged Shadows

Effectively, we are going to blur our hard-edged shadow so that it appears to have a softer edge. The straightforward way to implement this is to perform shadow-map comparisons several times using offset texture coordinates and then average the results. Intuitively, the amount of this offset determines how blurry the final shadow is.

Note that we cannot blur the actual contents of the shadow map, because it contains depth values. Instead, we have to blur the *results* of the shadow-map lookup when rendering a scene, which requires us to perform several lookups, each with a position offset slightly from the pixel's true position.

NVIDIA Ex1023
Page Q-311

This approach resembles the traditional method for generating soft shadows by rendering multiple shadow maps. We just "rotate the problem 90 degrees" and replace expensive multiple shadow-map-generating passes with much less expensive multiple lookups into a single shadow map.

The GLSL code in Listing 17-1 demonstrates this idea.

Unfortunately, the naive approach only works well when the penumbra region is very small. Otherwise, banding artifacts become painfully obvious, as shown in Figure 17-1. When the number of shadow-map samples is small compared to the penumbra size, our "soft" shadow will look like several hard-edged shadows, superimposed over one another. To get rid of these banding artifacts, we would have to increase the number of shadow-map samples to a point where the algorithm becomes impractical due to performance problems.

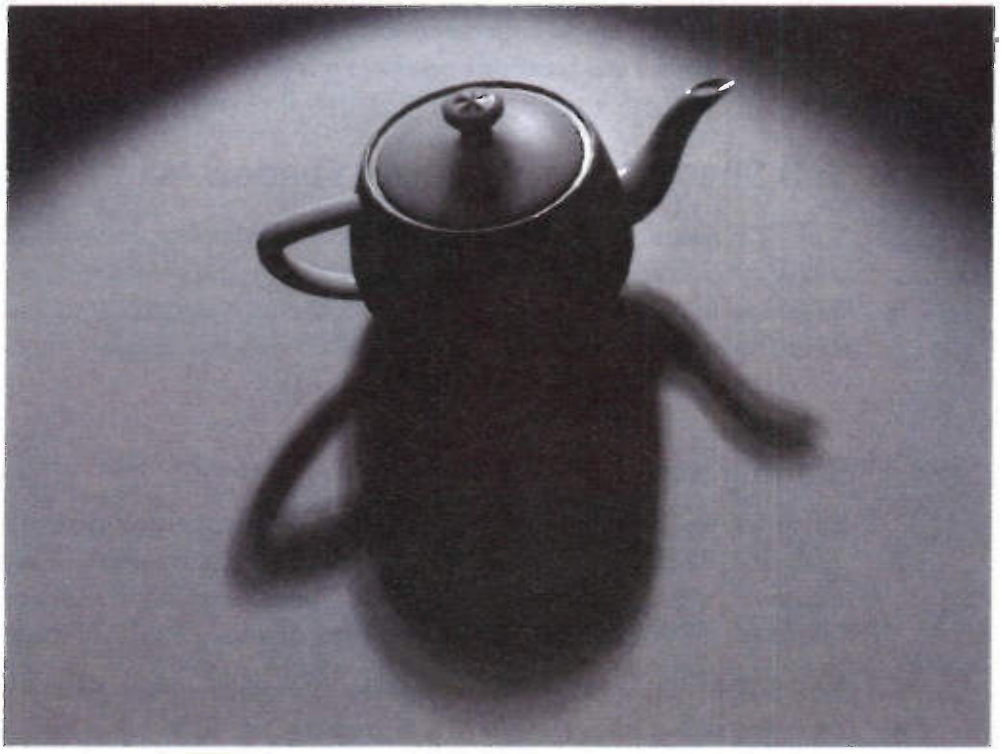


Figure 17-1. A Naive Approach to Blurring Shadows
The image was rendered using 32 shadow-map samples. Notice the heavy banding in penumbra regions.

NVIDIA Ex1023
Page Q-312

Listing 17-1. Implementing a Basic Algorithm for PCF

```
#define SAMPLES_COUNT 32
#define INV_SAMPLES_COUNT (1.0f / SAMPLES_COUNT)

uniform sampler2D decal;  // decal texture
uniform sampler2D spot;   // projected spotlight image
uniform sampler2DShadow shadowMap;  // shadow map

uniform float fwidth;
uniform vec2 offsets[SAMPLES_COUNT];

// these are passed down from vertex shader
varying vec4 shadowMapPos;
varying vec3 normal;
varying vec2 texCoord;
varying vec3 lightVec;
varying vec3 view;

void main(void)
{
  float shadow = 0;
  float fsize = shadowMapPos.w * fwidth;
  vec4 smCoord = shadowMapPos;

  for (int i = 0; i<SAMPLES_COUNT; i++) {
    smCoord.xy = offsets[i] * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
  }

  vec3 N = normalize(normal);
  vec3 L = normalize(lightVec);
  vec3 V = normalize(view);

  vec3 R = reflect(-V, N);

  // calculate diffuse dot product
  float NdotL = max(dot(N, L), 0);

  // modulate lighting with the computed shadow value
  vec3 color = texture2D(decal, texCoord).xyz;

  gl_FragColor.xyz = (color * NdotL + pow(max(dot(R, L), 0), 64)) *
                     shadow * texture2DProj(spot, shadowMapPos) +
                     color * 0.1;
}
```


NVIDIA Ex1023
Page Q-313

Let's find out how to implement this blurring in a more effective way without using too many shadow-map samples. To do this, we apply Monte Carlo methods, which are based on a probabilistic computation in which a number of *randomly placed* samples are taken of a function and averaged. In our context, that means we need to sample our shadow map *randomly*. This random placement of samples turns out to be a wise decision. The key point here is that the sequence of shadow-map sample locations is slightly different each time we calculate the shadowing value, so that the approximation error is different at different pixels. This effectively replaces banding with high-frequency noise, an artifact that the human visual system is trained to filter out very well. Note that the error is still there; it is just much better hidden from a human's eye.

If we carefully construct the sequence of offsets in a way that it has only a certain amount of randomness, we can get better results than if we use completely random offsets; this approach is called *stratified* or *jittered* sampling. To construct this distribution, we divide our domain into a number of subregions with equal areas, also known as *strata*, and randomly choose a sample location within each one. This subdivision provides even coverage of the domain, whereas each sample is still placed randomly within its cell.

If we assume that our sampling domain is a disk, stratification can be tricky. One possible solution is to stratify a square region and then warp it to a disk shape, preserving the area of the subregions. Generating a square grid of samples and then jittering it is straightforward, and for the area-preserving square-to-disk transformation, we use the following formulas:

$$x = \sqrt{v} \times \cos(2\pi u),$$
$$y = \sqrt{v} \times \sin(2\pi u),$$

where u, $v \in [0..1]$ and represent the jittered sample location within the square domain, and x, y are the corresponding sample position within the disk domain. Figure 17-2 illustrates the process. This way, the u dimension maps to angular position and the v dimension maps to radial distance from the disk center. The square roots compensate for the change in the cells' area as we move from the disk center outward, reducing the radial size of the cells.

17.2.2 Improving Efficiency

To achieve good shadow quality, we still need to take quite a few samples. We may need to take as many as 64 samples to have good-looking shadows with smooth penumbrae.

NVIDIA Ex1023
Page Q-314

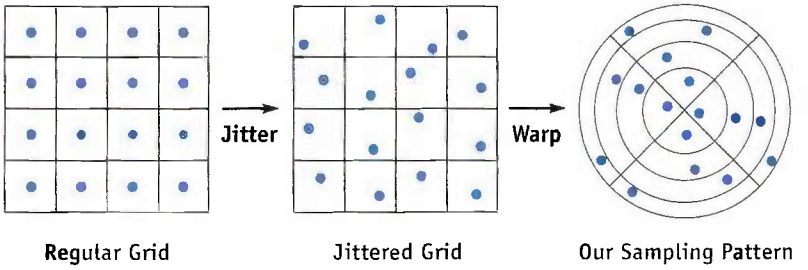


Figure 17-2. Jittering a Regular Grid and Warping to a Disk

Think about it: we need to sample the shadow map 64 times *per pixel*. This places significant pressure on texture memory bandwidth and shader processing.

Fortunately, most real-world shadows don't have a large penumbra region. The penumbra region is the soft part of the shadow, and it is where the most samples are needed. The other pixels are either completely in the shadow or completely out of the shadow, and for those pixels, taking one sample would give the same result as taking thousands of them.

Branching Like a Tree

Of course, we want to skip unnecessary computations for as many pixels as possible. But how do we know which pixels are in penumbrae and which are not? As you can guess, implementing this would require us to make decisions at the fragment level.

GeForce 6 Series GPUs finally enable true dynamic branching in fragment programs. This essentially means that now large blocks of code that do not need to be executed can be skipped completely, saving substantial computation.

Ideally, branch instructions would never have any overhead, and using them to skip code would only increase performance. Unfortunately, this is not the case. Modern GPUs have highly parallel, deeply pipelined architectures, calculating many pixels in concert. Pipelining allows GPUs to achieve high levels of performance, and parallelization is also natural for computer graphics, because many pixels perform similar computations. The consequence of using pipelining and parallel execution is that at any given time, a GPU processes several thousand pixels, but all of them run *the same* computations.

What does this have to do with branching, you say? Well, dynamic branching essentially enables pixels to choose their own execution paths. If two pixels follow different branches in their fragment program, you can no longer process those pixels together. That way, branching *hurts* parallelism to a certain extent.

NVIDIA Ex1023
Page Q-315

For this reason, you want to make sure your branching is as *coherent* as possible. Coherence in this context means that pixels in some local neighborhood should always follow the same branch—that way, all of them can be processed together. The less coherent branching is, the less benefit you'll get from skipping code with branching.

Predict and Forecast

Now let's try to predict how much benefit we can expect from branching. Basically, we need to detect whether a pixel is in the penumbra and decide how many shadow-map samples it needs. If we could come up with an inexpensive "estimator" function that would tell us whether we need more samples for the current pixel, we could use it as our branch condition.

It is impossible to find out something about the shadow map at a given pixel without actually sampling the shadow map. It also seems obvious that one sample is not enough to determine if the current pixel is only partly in shadow. So, we need a way to determine that with a small number of samples.

Therefore, we take a small number of shadow-map samples and compare them. If they all happen to have the same value (that is, all of them are either ones or zeroes), then it is very likely that the rest of the samples within the same region will have the same value. This works because most shadows are "well behaved": changes between lit and shadowed areas are pretty infrequent compared to penumbra size. The smaller the number of samples we take to try to figure this out, the bigger the benefit we gain from branching.

The next question is how to place our estimation samples to make the penumbra prediction as precise as possible. If these samples have fixed locations within the sampling domain, once again we will suffer from banding. But we know how to deal with banding: jitter the samples again!

As it turns out, the best we can do is distribute the jittered samples in a circle near the edge of the disk, where they are most likely to catch the transition from light to shadow, as shown in Figure 17-3.

Of course, this placement of samples will fail to catch penumbrae for some pixels. But thanks to jittering, the next pixel will have a slightly different placement of estimation samples and will probably get it correct. That way, we again dissipate prediction error in high-frequency noise across adjacent pixels.

NVIDIA Ex1023
Page Q-316

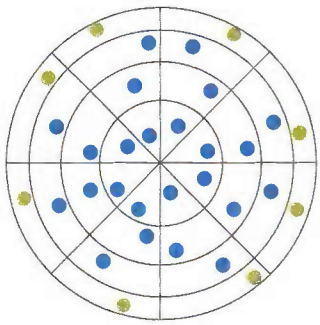


Figure 17-3. Estimation Samples (Shown in Green)

17.2.3 Implementation Details

For the shadow map itself, we use a depth texture and a "compare" texture application mode that are standard in OpenGL 1.4. Note that using floating-point formats for shadow maps is usually overkill and adds the unnecessary burden of having to manually perform comparisons in the fragment shader. As an additional benefit, NVIDIA hardware supports free four-sample bilinear percentage-closer filtering of depth textures, which helps to smooth out the penumbrae even more.

The bulk of our algorithm deals with jittered sampling. Because generating jittered offsets is a nontrivial task, it is best to precompute them and store them in a texture (jitter map).

To avoid the color-banding issue, pixels that are near each other need to use different sets of jittering offsets. It is not a problem if the offsets repeat due to tiling of the jitter map, as long as the noise frequency is high enough to keep the effect from being noticeable.

The most natural way to implement this is to use a 3D texture to store the jittered offsets. We store unique sequences of jittered offsets in "columns" of texels along the *r* texture dimension, and we tile the texture in screen space along the *s* and *t* dimensions. The texture extent in the *s* and *t* dimensions determines the size of the region where sequences are different, and its size in the *r* dimension depends on the maximum number of samples we intend to take. Conveniently for us, GLSL provides a built-in variable, `gl_FragCoord`, that contains current pixel position in screen space. This is exactly what we need to address this 3D texture in the *s* and *t* dimensions.

Because our offsets are two-dimensional and texels can have up to four channels, we can employ another optimization to reduce storage requirements for this 3D texture.

NVIDIA Ex1023
Page Q-317

We store a pair of offsets per texel, with the first offset occupying the r and g components and the second one sitting in the b and a components of the texel. This way, the "depth" of our texture in texels is half the number of shadow-map samples we take per pixel. As an additional bonus, we save some amount of texture memory bandwidth, because we have to look up the texture only once for every two shadow-map samples.

Our offsets are stored in normalized form: if we distribute our samples within a disk with a radius of 1, then the center-relative offsets are always in the range $[-1..1]$. In a fragment program, we scale each offset by a blurriness factor to control how soft our shadows appear. Offsets don't need to be high precision, so we can get away with the signed four-component, 8-bit-per-component format for this texture.

Because our estimation samples need to be jittered too, we can use some of the offsets stored in the 3D texture for them. These should be located on the outer rings of the sampling disk. We fill our 3D texture in such a way that estimation sample offsets start at $r = 0$ within each column, with the rest of the offsets following. This simplifies addressing in the fragment program a bit.

To better understand the construction process for the jitter map, look at the function `create_jitter_lookup()` in the source code included on the accompanying CD.

Performance Notes

Shadow maps have another well-known problem related to sampling and precision. This problem is commonly called "shadow acne" and appears as shadow "leakage" on the lit side of the object, similar to z-fighting. It is usually avoided by rendering the object's back faces to the shadow map instead of its front faces. This moves the problem to the triangles facing away from the light source, which is not noticeable because those triangles are not lit by that light source and have the same intensity as shadowed surfaces.

This trick has certain performance implications with our soft shadow algorithm. Though not visible, the "acne" will have a significant impact on the branching coherence. To remedy this, we modulate our penumbra estimation result with the function `(dot(N, L) <= 0 ? 0 : 1)`. That is, if a pixel happens to belong to the shadowed side of the object, the function will resolve to 0, effectively invalidating potentially incorrect penumbra estimation and forcing it to return a "completely in shadow" result.

Another possible improvement is to use midpoint shadow maps instead of regular shadow maps. Midpoint shadow maps work by storing an averaged depth between front faces and back faces, as viewed from the light source. When shadowing objects are closed and have reasonable thickness, this reduces shadow-aliasing artifacts as well, at the expense of more complex shadow-map construction.

NVIDIA Ex1023
Page Q-318

The GLSL fragment shader code in Listing 17-2 illustrates the adaptive sampling approach described in this chapter.

Listing 17-2. Implementing Adaptive Sampling for PCF of Shadow Maps

```
#define SAMPLES_COUNT 64
#define SAMPLES_COUNT_DIV_2 32
#define INV_SAMPLES_COUNT (1.0f / SAMPLES_COUNT)

uniform sampler2D decal;    // decal texture
uniform sampler3D jitter;   // jitter map
uniform sampler2D spot;     // projected spotlight image
uniform sampler2DShadow shadowMap;  // shadow map

uniform float fwidth;
uniform vec2 jxyscale;

// these are passed down from vertex shader
varying vec4 shadowMapPos;
varying vec3 normal;
varying vec2 texCoord;
varying vec3 lightVec;
varying vec3 view;

void main(void)
{
  float shadow = 0;
  float fsize = shadowMapPos.w * fwidth;
  vec3 jcoord = vec3(gl_FragCoord.xy * jxyscale, 0);
  vec4 smCoord = shadowMapPos;

  // take cheap "test" samples first
  for (int i = 0; i<4; i++) {
    vec4 offset = texture3D(jitter, jcoord);
    jcoord.z += 1.0f / SAMPLES_COUNT_DIV_2;

    smCoord.xy = offset.xy * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) / 8;

    smCoord.xy = offset.zw * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) / 8;
  }
```


NVIDIA Ex1023
Page Q-319

Listing 17-2 *(continued)*. Implementing Adaptive Sampling for PCF of Shadow Maps

```
    vec3 N = normalize(normal);
    vec3 L = normalize(lightVec);
    vec3 V = normalize(view);

    vec3 R = reflect(-V, N);

    // calculate diffuse dot product
    float NdotL = max(dot(N, L), 0);

    // if all the test samples are either zeroes or ones, or diffuse dot
    // product is zero, we skip expensive shadow-map filtering
    if ((shadow - 1) * shadow * NdotL != 0) {
      // most likely, we are in the penumbra
      shadow *= 1.0f / 8;  // adjust running total

      // refine our shadow estimate
      for (int i = 0; i<SAMPLES_COUNT_DIV_2 - 4; i++) {
        vec4 offset = texture3D(jitter, jcoord);
        jcoord.z += 1.0f / SAMPLES_COUNT_DIV_2;

        smCoord.xy = offset.xy * fsize + shadowMapPos;
        shadow += texture2DProj(shadowMap, smCoord)* INV_SAMPLES_COUNT;

        smCoord.xy = offset.zw * fsize + shadowMapPos;
        shadow += texture2DProj(shadowMap, smCoord)* INV_SAMPLES_COUNT;
      }
    }

    // all done - modulate lighting with the computed shadow value
    vec3 color = texture2D(decal, texCoord).xyz;

    gl_FragColor.xyz = (color * NdotL + pow(max(dot(R, L), 0), 64)) *
                       shadow * texture2DProj(spot, shadowMapPos) +
                       color * 0.1;

}
```


NVIDIA Ex1023
Page Q-320

17.3 Conclusion

We have described an efficient method for rendering soft-edged shadows in real time based on percentage-closer filtering in a fragment shader. Figure 17-4 shows sample results. This method is fully orthogonal with respect to other rendering techniques; for example, the stencil buffer can still be used for additional effects. Because most of the work is performed on the GPU, this method will demonstrate higher performance as new, more powerful GPU architectures become available.

Note that the approach described here may not work very well with unusual shadow-map projections such as perspective shadow maps. In such cases, more work is necessary to adjust the size of the penumbra on a per-pixel basis.

If used carefully, branching can provide significant performance benefits by allowing the GPU to skip unnecessary computations on a per-fragment basis. On a GeForce



Figure 17-4. Realistic Soft-Edged Shadows in Real Time

NVIDIA Ex1023
Page Q-321

6800 GT GPU, our test implementation of adaptive shadow-map sampling showed more than twice the improvement in frame rate compared to the regular shadow-map sampling. The key point for achieving high performance with branching is to keep it as coherent as possible. This will become even more important for future hardware architectures, from which we can expect even more parallelism and deeper pipelining.

17.4 References

Akenine-Möller, Tomas, and Ulf Assarsson. 2002. "Approximate Soft Shadows on Arbitrary Surfaces Using Penumbra Wedges." In *Thirteenth Eurographics Workshop on Rendering*, pp. 297–306.

Assarsson, Ulf, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. 2003. "An Optimized Soft Shadow Volume Algorithm with Real-Time Performance." In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003*, pp. 33–40.

Brabec, Stefan, and Hans-Peter Seidel. 2002. "Single Sample Soft Shadows Using Depth Maps." In *Proceedings of Graphics Interface 2002*, pp. 219–228.

Cook, Robert L. 1986. "Stochastic Sampling in Computer Graphics." *ACM Transactions on Graphics* 5(1), pp. 51–72.

Kirsch, Florian, and Jurgen Dollner. 2003. "Real-Time Soft Shadows Using a Single Light Sample." *WSCG* 11(1), pp. 255–262.

Reeves, William T., David H. Salesin, and Robert L. Cook. 1987. "Rendering Antialiased Shadows with Depth Maps." *Computer Graphics* 21(4), pp. 283–291.

NVIDIA Ex1023
Page Q-322



Chapter 18

Using Vertex Texture Displacement for Realistic Water Rendering

Yuri Kryachko
1C:Maddox Games

Water surfaces are common in computer graphics, especially in games. They are a critical element that can significantly improve the level of realism in a scene. But depicting them realistically is a hard problem, because of the high visual complexity present in the motion of water surfaces, as well as in the way light interacts with water. This chapter describes techniques developed for rendering realistic depictions of the ocean for the game *Pacific Fighters*.

Modern graphics hardware provides a number of useful features with DirectX Shader Model 3.0 that can be used to aid the rendering of water surfaces. This chapter will discuss how one of these features, vertex texturing, can be used to increase the realism of rendered water surfaces. Figure 18-1 shows some sample results. In addition, we also use branching in order to improve the performance of our vertex programs.

18.1 Water Models

For water animation and rendering, a number of methods have been developed. The most remarkable and realistic-looking ones are those based on fluid dynamics and Fast Fourier Transforms (FFTs)(such as Tessendorf 2001). These methods provide very realistic results, but unfortunately they require substantial amounts of computation, making them inappropriate for interactive applications.

NVIDIA Ex1023
Page Q-323

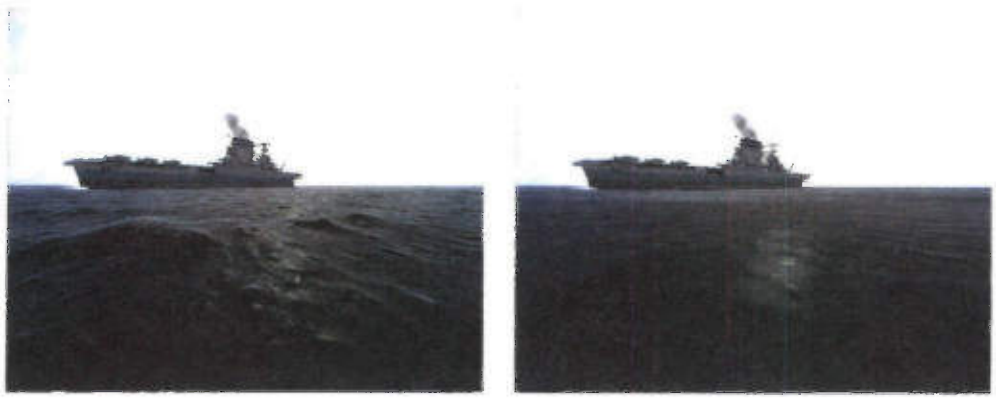


Figure 18-1. The Benefits of Displacement Mapping
Water surface rendered (left) with displacement mapping and (right) without displacement mapping.

At the other extreme, most games currently use very simple water models, most of which employ normal maps to create visual details. Unfortunately, these approaches cannot provide enough realism and do not faithfully reproduce waves on the surface.

We seek a technique that combines the speed of simple normal-mapped water-rendering methods with the visual quality of FFT-like approaches.

18.2 Implementation

Our implementation builds upon rendering algorithms that employ normal maps for lighting calculations. Because normal maps faithfully reproduce fine detail in high-frequency waves, we use them for our lighting calculations. However, in addition, we perturb the water mesh geometrically with lower-frequency waves with large amplitude.

18.2.1 Water Surface Model

Our model of a water surface is based on the superposition of several height maps, tiled in both space and time. Each texture represents one "harmonic" or "octave" of the spectrum, and the textures are added together as in Fourier synthesis. These textures are called *height maps* because each value represents the elevation of the corresponding point above the horizontal plane.

Height maps are great for artists: creating them is as simple as painting a grayscale image. See Figure 18-2. With height maps, artists can easily control the parameters of

NVIDIA Ex1023
Page Q-324



Figure 18-2. A Height Map Used for Water Displacement

water animation down to individual waves by just painting their shapes. Height maps also work well as vertex textures: using them to displace vertex positions vertically is trivial.

By combining several height maps with different spatial and time scales, we can achieve complex and visually intricate animations:

$$H(x, y, t) = \sum_{i=0}^{N} h\left(A_i^x x + B_i^x,\ A_i^y y + B_i^y,\ A_i^t t + B_i^t\right).$$

The coefficients A and B and the number of terms under the sum are chosen heuristically to achieve the most aesthetically pleasing results while minimizing repeating-pattern artifacts. In *Pacific Fighters*, we sum four height maps for lighting calculations, and two of them with the largest scale are used for displacement mapping. This is sufficient for simulating moving ocean surfaces at scales from 10 cm up to 40 km.

18.2.2 Implementation Details

All of the computations we need to perform can be classified into two groups: geometric displacement computations and lighting computations. Because our water surface is finely tessellated, it is reasonable to perform lighting calculations at the fragment program level, offloading the displacement mapping to the vertex stage. Also, performing lighting calculations at the vertex stage can create visual artifacts, especially in the distance.

NVIDIA Ex1023
Page Q-325

At the time of writing, the only hardware capable of doing vertex texturing were GeForce 6 Series GPUs and the latest NVIDIA Quadro FX GPUs. The vertex texture implementation on this hardware has certain restrictions; in particular, vertex textures must be 32-bit-per-component textures, floating point, and they can't use any filtering mode except nearest filtering. Nevertheless, they proved to be very useful for the techniques described in this chapter.

18.2.3 Sampling Height Maps

Our implementation samples the height maps per vertex and computes the resulting displacement value in the vertex program. For sampling, we use a radial grid, centered at the camera position. This grid is tessellated in such a way that it provides more detail closer to the viewer, as shown in Figure 18-3.

The following equations show how the vertex positions for the radial grid are computed.

$$\begin{aligned} r &= a_0 + a_1 i^4 \\ x_{i,j} &= r\cos\left(2\pi j/M\right) \\ y_{i,j} &= r\sin\left(2\pi j/M\right), \end{aligned}$$

where $i = [0..N-1], j = [0..M-1]$. We choose a_0, a_1 so that

$$\begin{aligned} r_0 &= a_0 = 10 \text{ cm} \\ r_{N-1} &= a_0 + a_1\left(N-1\right)^4 = 40 \text{ km}. \end{aligned}$$

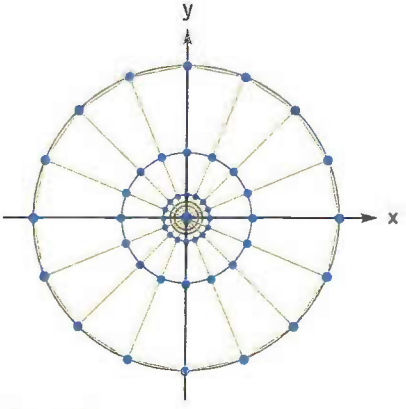


Figure 18-3. Radial Grid for Sampling Vertex Textures

NVIDIA Ex1023
Page Q-326

With this approach, we naturally get distance-based tessellation, which provides a simple level-of-detail (LOD) scheme. Other approaches, such as the ROAM or SOAR terrain-rendering algorithms, could be used here, but they require a significant amount of work on the CPU side, which would eliminate all the benefits of using vertex textures. See Chapter 2 of this book, "Terrain Rendering Using GPU-Based Geometry Clipmaps," for another approach to rendering height fields on the GPU with adaptive tessellation.

Listing 18-1 shows the simple vertex shader that implements sampling from a single height map with a radial grid.

Listing 18-1. Vertex Shader for Sampling from a Height Map Using the Radial Grid Geometry

```
float4 main(float4 position : POSITION,
  uniform sampler2D tex0,
  uniform float4x4 ModelViewProj,
  uniform float4 DMParameters, // displacement map parameters
  uniform float4 VOfs) : POSITION
{
  // Read vertex packed as (cos(), sin(), j)
  float4 INP = position;

  // Transform to radial grid vertex
  INP.xy = INP.xy * (pow(INP.z, 4) * VOfs.z);

  // Find displacement map texture coordinates
  // VOfs.xy, DMParameters.x - Height texture offset and scale
  float2 t = (INP.xy + VOfs.xy) * DMParameters.x;

  // Fetch displacement value from texture (lod 0)
  float vDisp =  tex2D(tex0, t).x;

  // Scale fetched value from 0..1:
  // DMParameters.y - water level
  // DMParameters.z - wavy amplitude
  INP.z = DMParameters.y + (vDisp - 0.5) * DMParameters.z;

  // Displace current position with water height
  // and project it
  return mul(ModelViewProj, INP);
}
```


NVIDIA Ex1023
Page Q-327

18.2.4 Quality Improvements and Optimizations

Packing Heights for Bilinear Filtering

Vertex texture fetch can be quite expensive. On GeForce 6 Series hardware, a single vertex texture fetch can introduce noticeable latency in the vertex program. So we want to minimize the number of texture fetches inside the vertex program. On the other hand, it is very desirable to perform some kind of filtering on the texture values; otherwise, visual quality will be significantly degraded.

Traditional well-known filtering methods are bilinear and trilinear filtering. Bilinear filtering computes the weighted average of four texels closest to the coordinates of the texture fetch. Trilinear filtering averages the results of bilinear lookups in adjacent mip levels, weighting each by the corresponding LOD fraction.

Because the current generation of graphics hardware doesn't support any form of filtering of vertex texture values, we have to emulate filtering in the shader with explicit math instructions. When implemented naively, even the simplest bilinear filter would require four texture lookups to calculate a single filtered value. A trilinear filter would require twice as many texture lookups.

To reduce the number of texture fetches necessary for filtering, we build our texture in a special way, so that each texel contains all the data necessary for a single bilinear texture lookup. This is possible because our height maps are essentially one-component textures, and we can pack four height values into a single texel of a four-component texture:

$$
\begin{aligned}
A_x^{i,j} &= H^{i,j}, \\
A_y^{i,j} &= H^{i+1,j}, \\
A_z^{i,j} &= H^{i,j+1}, \\
A_w^{i,j} &= H^{i+1,j+1}, \\
A_{filtered} &= F\left(A_{x,y,z,w}^{i,j}\right),
\end{aligned}
$$

where $i = 0..N-1, j = 0..\mathrm{M}-1$. H is our height map value, F is a filtering function, and A is the packed output texture.

Listing 18-2 implements bilinear texture lookup into vertex texture, packed as shown previously.

NVIDIA Ex1023
Page Q-328

Listing 18-2. Efficient Bilinear Texture Interpolation in Vertex Shaders
Based on fetching the appropriate four height components with a single texture fetch.

```
float tex2D_bilinear4x(uniform sampler2D tex,
  float4 t,
  float2 Scales)
{
  float size  = Scales.x;
  float scale = Scales.y;

  float4 tAB0 = tex2Dbias(tex, t);

  float2 f = frac(t.xy * size);
  float2 tAB  = lerp(tAB0.xz, tAB0.yw, f.x);
  return lerp(tAB.x, tAB.y, f.y);
}
```

We can extend this approach for trilinear filtering. Because trilinear filtering requires a fractional LOD value, we can use the distance from the camera as a good approximation of LOD. The code in Listing 18-3 implements trilinear texture lookup into the packed vertex texture.

Listing 18-3. Extending the Bilinear Filtering Technique to Trilinear Filtering

```
float tex2D_trilinear(uniform sampler2D tex,
  float4 t,
  float2 Scales)
{
  float fr = frac(t.z);
  t.z -= fr;  // floor(t.zw);
  float Res;
  if (fr < 0.30)
    Res =  tex2D_bilinear4x(tex, t.xyzz, Scales);
  else if (fr > 0.70)
    Res =  tex2D_bilinear4x(tex, t.xyzz + float4(0, 0, 1, 1),
                            Scales * float2(0.5, 2));
  else {
    Res = tex2D_bilinear4x(tex, t.xyzz, Scales);
    float Res1 = tex2D_bilinear4x(tex, t.xyzz + float4(0, 0, 1, 1),
                                  Scales * float2(0.5, 2));
    fr = saturate((fr - 0.30) * (1 / (0.70 - 0.30)));
    Res = Res1 * fr + Res * (1 - fr);
  }
  return Res;
}
```


NVIDIA Ex1023
Page Q-329

Note that we've further optimized trilinear texture fetch by performing two texture lookups only in the region where the influence of both mip levels is significant. In other regions, we "snap" the LOD value to the closest mip level, thus saving on texture bandwidth.

Avoiding Unnecessary Work with Branching

Even with optimized texture filtering, the number of texture fetches during water rendering can still be high, which significantly affects performance. We could reduce the total number of rendered vertices, but that would lower overall visual detail and increase aliasing.

Because we render our water with large batches of geometry, some of the triangles end up being completely off-screen. Note that even for such triangles, the vertex program is still executed, wasting precious computational resources. We can save a significant amount of per-vertex work if we skip computation for triangles falling outside the camera frustum.

The vertex program operates at one vertex at a time and has no access to topological information, so we can make decisions only on the per-vertex level, but not on the per-triangle level. This can create artifacts if some of the vertices within a triangle skip vertex texturing and others don't. We have found that in practice our triangles and our vertex texture displacements are small enough for this artifact not to be detectable.

The following pseudocode illustrates this idea:

```
float4 ClipPos = mul(ModelViewProj, INP);
float3 b0 = abs(ClipPos.xyz) < (ClipPos.www * C0 + C1);

if (all(b0)) {
  // Vertex belongs to visible triangle,
  // Perform texture sampling and displace vertex accordingly
}
```

In the preceding code, we use the clip-space vertex position to determine if the current vertex lies within the frustum, and then we perform the expensive computations only when necessary.

The values `C0` and `C1` are special "fudge" constants that control how much triangles need to extend beyond the camera frustum to trigger clipping. That way, we avoid artifacts caused by skipping texturing for out-of-frustum vertices whose triangles are still visible. Effectively, we are making our "clipping" frustum slightly wider, allowing for a certain amount of "guard-band" space along screen edges. Because our water plane is tessellated finely enough and the vertex texture displacements are reasonable, this simple method works well in practice.

NVIDIA Ex1023
Page Q-330

Using Render-to-Texture

We can also improve the speed of our approach by first combining our height map textures into a single floating-point texture in a separate pass. It then becomes unnecessary to perform multiple expensive filtering operations in the vertex shader. Additionally, we can now use a more compact texture format, 16-bit floating point, for storage of the original height maps. We could also store a sequence of animated height maps as slices of a 3D texture, which would make animation smoother.

With this optimization, our rendering loop becomes two passes:

1. Combine the height maps using a special pixel shader by rendering a single quadrilateral into an fp32-texture. Texels in this texture map to the vertices of the radial mesh.
2. Use the generated height map as a vertex texture to displace the radial mesh vertices, as described previously.

Back Sides of Waves

Because our lighting computations are performed in the pixel shader under the assumption that the water surface is flat, this approximation can cause visual artifacts in certain cases.

In the case depicted in Figure 18-4, we see the back side of the wave, even though it is directed outward from the viewer due to geometrical displacement and shouldn't be visible in reality. This results in disturbing bright areas at the tops of the waves.

To minimize these artifacts, we adjust our normal vectors used for the lighting calculation by "tilting" them toward the viewer a bit, so that they correspond to the front faces of the wave more closely. You can find source code for this technique on the accompanying CD. Figure 18-5 shows a scene produced using the methods described in this chapter.

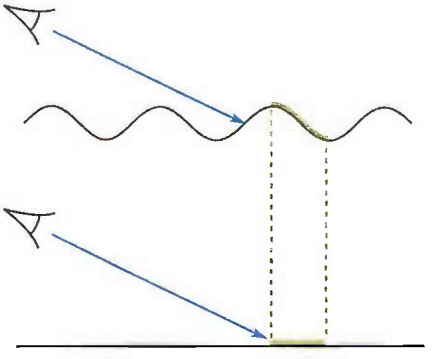


Figure 18-4. A Source of Rendering Artifacts
The back side of a wave (green) may be shaded even though it shouldn't be visible. Adjusting the normal vector used in the lighting calculation can significantly reduce this error.

NVIDIA Ex1023
Page Q-331



Figure 18-5. Extremely Realistic Water Rendered with Techniques Presented Here

18.2.5 Rendering Local Perturbations

Sometimes it is desirable to render local choppiness caused by buoyant objects or by objects falling into the water. This is especially important for games, where it is necessary to generate explosions, ship trails, or the like. Because it is hard to integrate physically correct methods in our height-map-based model of the water surface, we discuss simpler methods, based on heuristics.

Analytical Deformation Model

The simplest way to achieve local choppiness is to disturb displaced vertex positions analytically, by combining them with the computed vertex position in the vertex shader. For explosions, we can use the following formula:

$$I(r) = \frac{I_0 \sin(kr + \omega t) e^{-bt}}{r^2},$$

NVIDIA Ex1023
Page Q-332

where r is the distance from the explosion center in the water plane and b is a decimation constant. The values of I_0, ω, and k are chosen according to a given explosion and its parameters.

For rendering, we can use the same radial grid as the one used for regular water rendering, but centered at the explosion location.

Dynamic Displacement Mapping

Another option is to render all of the locally generated displacements directly into the vertex texture, essentially implementing a general-purpose programming on the GPU (GPGPU) type of approach. That way, we generate a vertex texture in the first pass and then use it in a subsequent pass for the actual water rendering. As an additional benefit, we can offload some work from the vertex shaders by filtering the base height map and summing "octaves" in the pixel shader.

To calculate displacements, we can either employ the above-mentioned analytical model or try using a cellular-automata approach, by evolving local displacements from frame to frame. Wind effects can also be taken into account by blurring the texture along the appropriate direction.

However, to cover 1 km of water surface with 50 cm resolution, it would be necessary to use a texture about 2048×2048 in size, which would create additional pressure on texture memory and shader execution speed. Also, quick transitions of the viewpoint would be problematic.

Nevertheless, we encourage the reader to experiment with these approaches.

Foam Generation

When choppiness is strong enough, we can generate foam to further increase realism. The simplest way to do this is to blend in a precreated foam texture at vertices displaced above a certain height H_0. Transparency of the foam texture is calculated according to the following formula:

$$Foam.a = \text{saturate}\left(\frac{H - H_0}{H_{\max} - H_0}\right),$$

where $H_{\max}$ is the height at which the foam is at maximum, H_0 is the base height, and H is the current height.

NVIDIA Ex1023
Page Q-333

The foam texture can be animated to show the evolution of foam creation and dissipation. The animation sequence can be either created manually by artists or generated programmatically.

18.3 Conclusion

By combining the flexibility of vertex texture fetch and the performance-saving features of dynamic branching, we were able to develop a practical method for rendering realistic-looking water surfaces at interactive speeds. The approach described here was successfully applied in the title *Pacific Fighters*, and it allowed us to create a realistic water surface across a significant range of distances—from 10 cm up to 40 km. This is acceptable for modern flight simulators. We were able to eliminate tiling artifacts across the whole visible region of the water surface.

Future hardware is likely to enable even more robust implementations of the technique, in particular eliminating the need to manually perform filtering of the texture values, as well as providing even more vertex shader performance.

The quality of our approach can be increased even further by employing advanced shading techniques, such as parallax mapping, to provide fine-grained details and bumps on the water surface. See Chapter 8, "Per-Pixel Displacement Mapping with Distance Functions," for one approach like this.

Finally, lighting calculations could greatly benefit from high-dynamic-range techniques, because highly reflective water surfaces can exhibit huge brightness variations.

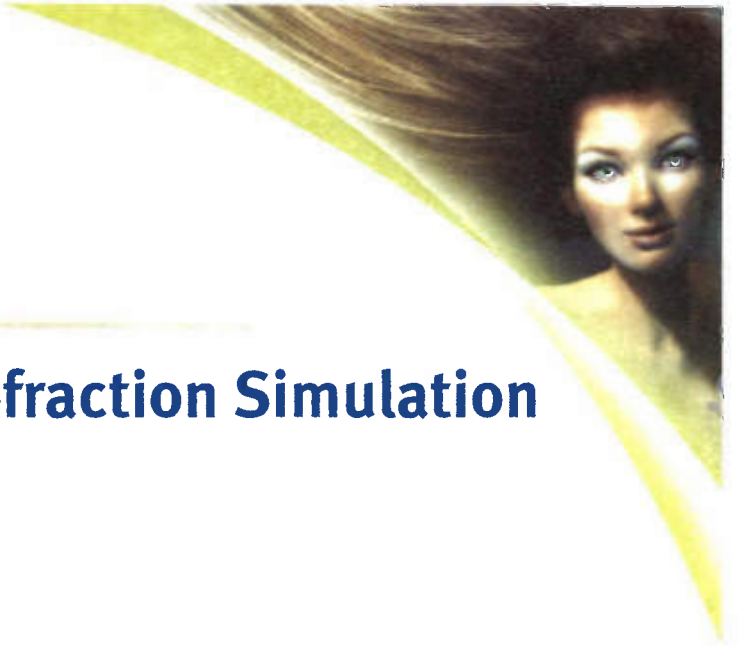
18.4 References

Fournier, Alain, and William T. Reeves. 1986. "A Simple Model of Ocean Waves." In *Computer Graphics (Proceedings of SIGGRAPH 86)*, pp. 75-84.

Kryachko, Yuri. 2004. "Modelling Sea and River Water with Pixel Shader 2.0." Presentation. Available online at **http://www.kriconf.ru/2004/rec/KRI-2004.Programming_20.ppt**

Tessendorf, Jerry. 2001. "Simulating Ocean Water." In "Simulating Nature: Realistic and Interactive Techniques," *SIGGRAPH 2001* course notes, course 47.

NVIDIA Ex1023
Page Q-334



Chapter 19

Generic Refraction Simulation

Tiago Sousa
Crytek

Refraction, the bending of light as it passes from a medium with one index of refraction to another (for example, from air to water, from air to glass, and so on), is challenging to achieve efficiently in real-time computer graphics. There are many well-known techniques for simulating light reflection (such as planar reflection maps and cubic environment maps), and these techniques work well in most situations. However, there aren't as many widely known and effective techniques for simulating refraction. This chapter describes an approach to refraction based on perturbing the texture coordinates used in a texture lookup of an image of the nonrefractive objects in the scene. This technique is very efficient and works well in many cases. The method presented here is an expansion of the techniques used in *Far Cry* for rendering water, heat haze, and the sniper-scope lens, among other effects.

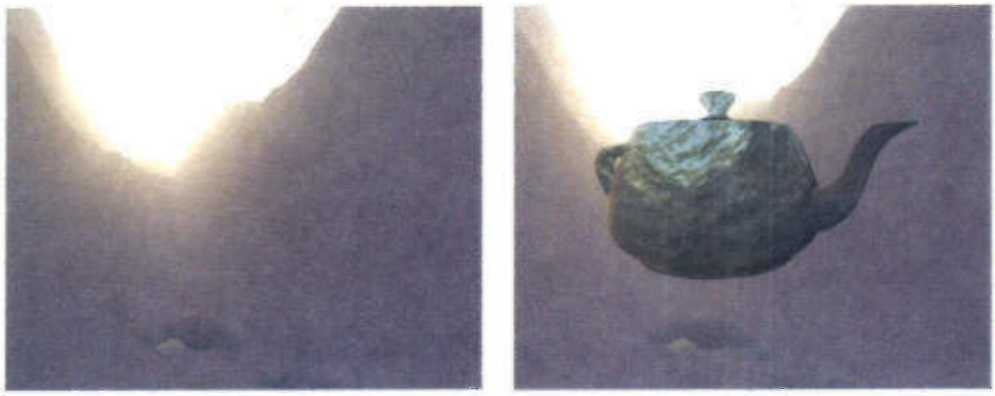
There are several ways to simulate refraction: some are based on precomputing an environment map and then using it at runtime; others are based on computing the environment map on the fly. Drawbacks of these techniques are high texture memory usage and the performance penalty, especially if there are many refractive surfaces in the scene requiring different environment maps.

Another problem with current water refraction simulation techniques is that they require two rendering passes: one to generate the refraction map with geometry above the water plane clipped, and then another pass to render the water surface. This approach can also have poor performance, especially in complex rendering situations.

Introduction

NVIDIA Ex1023
Page Q-335

This chapter discusses a simple technique to overcome these problems. We start by introducing the basic technique, which is based on using the current back buffer as a refraction map and then adding displacement to the texture coordinates to simulate the refractive look. This basic approach can lead to artifacts, however, so we then discuss how to mask out geometry from the refraction map. Finally, we demonstrate some general techniques for rendering realistic water and glass using this refraction simulation technique. Figure 19-1 shows a simple example of refraction in a scene.



(a) (b)

Figure 19-1. An Example of Refraction in a Scene
*(a) Background scene (**S**). (b) Final composition with refractive meshes.*

19.1 Basic Technique

The first step of the basic refraction technique is to render the scene geometry into a texture, skipping all refractive meshes. This texture can be used to determine which objects are visible behind the refractive objects that will be rendered in a subsequent pass. We denote this texture as S.

The second step is to render the refractive meshes, looking up values from the texture S with a perturbation applied to simulate the refractive look. The perturbation can be achieved using a normal map N, where the normal-map red and green (XY) components are used and scaled by some small value to add a displacement into the projected texture coordinates. This approach is straightforward to implement in a shader: (1) fetch the texture N, (2) use the XY components scaled by a small value (such as 0.05), and (3) add this displacement value into the projected texture coordinates for S. Listing 19-1 shows a shader that demonstrates this approach; Figure 19-2 illustrates the three steps.

NVIDIA Ex1023
Page Q-336

Listing 19-1. Shader for Basic Refraction Technique

```
half4 main(float2 bumpUV : TEXCOORD0,
  float4 screenPos : TEXCOORD1,
  uniform sampler2D tex0,
  uniform sampler2D tex1,
  uniform float4 vScale) : COLOR
{
  // fetch bump texture, unpack from [0..1] to [-1..1]
  half4 bumpTex=2.0 * tex2D(tex0, bumpUV.xy) - 1.0;

  // displace texture coordinates
  half2 newUV = (screenPos.xy/screenPos.w) + bumpTex.xy * vScale.xy;

  // fetch refraction map
  return tex2D(tex1, newUV);
}
```



Figure 19-2. A Visualization of the Rendering Steps
*(a) Normal map (**N**). (b) Projected background texture (**S**). (c) Projected background texture with perturbation applied.*

19.2 Refraction Mask

The basic technique presented in the previous section will work reasonably well in a variety of situations, but it can be prone to artifacts if a bump with a large enough scale is used for perturbation when a mesh is in front of a refractive mesh. Figure 19-3 illustrates the problem: the brown sphere is inaccurately rendered as in the refraction in the teapot, even though it is in front of the teapot, not behind it.

These artifacts are visible because the texture S has all the scene geometry rendered into it, including objects in front of the refractive mesh, and we are indiscriminately applying

NVIDIA Ex1023
Page Q-337



Figure 19-3. Artifacts Caused by the Technique

perturbation on every pixel. This leads to refraction "leakage" between objects in the scene. A straightforward solution is to decrease the amount of perturbation applied, until artifacts are reduced to an acceptable visual quality level. However, it is difficult to find a scale for the perturbation that works well in all circumstances, and this solution also has the negative side effect of capping how bumpy your refractive surfaces can be.

A better solution instead is to make sure that we actually don't add perturbation into the wrong pixels. To do so, we lay down a mask in the alpha channel of the S texture for all refractive meshes and use it to ensure that the perturbed coordinates are used only if they end up being inside a refractive object's area on the screen. See Figure 19-4 for an example.

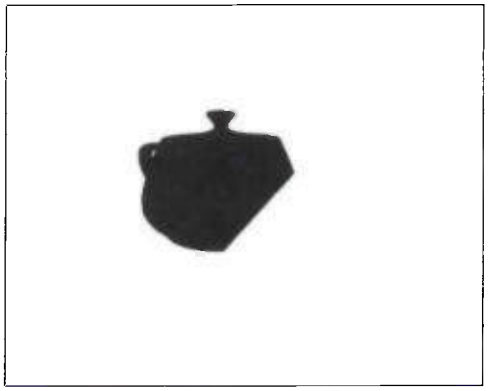


Figure 19-4. The Alpha Channel in the Frame Buffer
The refractive mesh is black.

NVIDIA Ex1023
Page Q-338

For this approach to work, we need to change the rendering a bit: first we make sure that the alpha channel of texture S is cleared to white, and then we render refractive meshes as black only into texture S's alpha channel. When rendering the refractive meshes, we make use of the extra information stored in S's alpha channel, to discriminate which pixels will be processed. This is done by checking if alpha is white; if so, we do not add perturbation in that case. Only if alpha is black do we add perturbation. Thus, if perturbation would have included a pixel outside of the refractive object, we instead just use the original pixel, giving whatever geometry was directly behind the refractive object. Although this result is inaccurate—refraction may actually lead to objects that aren't directly behind the object becoming visible—it works well in practice and is quite efficient. Listing 19-2 shows the shader that implements this approach, and Figure 19-5 shows the results.

Listing 19-2. Improved Shader That Uses the Refraction Mask to Avoid Including Pixels from Objects in Front of the Refractive Object

```
half4 main(float2 bumpUV : TEXCOORD0,
  float4 screenPos : TEXCOORD1
  uniform sampler2D tex0,
  uniform sampler2D tex1,
  uniform float4 vScale) : COLOR
{
  // fetch bump texture
  half4 bumpTex=2.0 * tex2D(tex0, bumpUV.xy) - 1.0;

  // compute projected texture coordinates
  half2 vProj = (screenPos.xy/screenPos.w);

  // fetch refraction map
  half4 vRefrA = tex2D(tex1, vProj.xy + bumpTex.xy * vScale.xy);
  half4 vRefrB = tex2D(tex1, vProj.xy);

  return vRefrB * vRefrA.w + vRefrA * (1 - vRefrA.w);
}
```

It's simple in this way to remove the artifacts, making it possible to use this refraction simulation technique on every mesh type. Although almost not noticeable, some artifacts may be visible, because we're replacing occluder pixel colors with the background color.

NVIDIA Ex1023
Page Q-339



Figure 19-5. Artifacts Removed by Using the Refraction Mask

19.3 Examples

Many interesting effects can be achieved with the presented technique. For example, the normal map used may be an animated texture, or a transformation may be applied to its coordinates. In this section, we cover a few practical examples.

19.3.1 Water Simulation

Simulating refractive water is one of the applications in which the presented technique is particularly effective, as current methods use an extra pass to generate the refractive map, by rendering the scene again and clipping geometry above the water plane. With this technique, we can render water in just one pass, because the only extra work is to render the water plane into texture S's alpha channel for the refraction mask.

For *Far Cry*, we used an animated bump texture, which worked out really well. In later experiments, however, we achieved even better results by using multiple bump-map layers for animating water waves, then blending between reflection and refraction through a per-pixel Fresnel term. Figure 19-6 shows the rendering steps for water.

The water simulation rendering is done by rendering the scene as described in previous sections, with the water plane rendered into the back-buffer alpha for the refraction mask. Next we need to generate a water reflection map, by rendering the scene reflected through the water plane into the reflection map and clipping the geometry below the water surface. Finally, there's no need to do an extra rendering pass for the refraction map, because we can use texture S and its refraction mask on the alpha channel.

NVIDIA Ex1023
Page Q-340



Figure 19-6. The Rendering Steps for Water
(a) Reflection. (b) Refraction. (c) Final composition.

Once the main texture input has been generated, we must render the water itself. In this example, we use four bump layers. The texture coordinates of the bump layers are scaled in the vertex shader by an increasing value for each, so that we have a nice mix of low-frequency and high-frequency details on the water waves. A translation is also added to the texture coordinates to simulate the motion of waves in the water. Listings 19-3 and 19-4 show the implementation.

Listing 19-3. Fresnel Approximation Computation for Water Rendering

```
half Fresnel(half NdotL, half fresnelBias, half fresnelPow)
{
  half facing = (1.0 - NdotL);
  return max(fresnelBias +
             (1.0 - fresnelBias) * pow(facing, fresnelPow), 0.0);
}
```

Listing 19-4. The Fragment Program for Refractive/Reflective Water

```
half4 main(float3 Eye : TEXCOORD0,
  float4 Wave0 : TEXCOORD1,
  float2 Wave1 : TEXCOORD2,
  float2 Wave2 : TEXCOORD3,
  float2 Wave3 : TEXCOORD4,
  float4 ScreenPos : TEXCOORD5,
  uniform sampler2D tex0,
  uniform sampler2D tex1,
  uniform sampler2D tex2) : COLOR
{
  half3 vEye = normalize(Eye);
```


NVIDIA Ex1023
Page Q-341

Listing 19-4 *(continued)*. The Fragment Program for Refractive/Reflective Water

```
    // Get bump layers
    half3 vBumpTexA = tex2D(tex0, Wave0.xy).xyz;
    half3 vBumpTexB = tex2D(tex0, Wave1.xy).xyz;
    half3 vBumpTexC = tex2D(tex0, Wave2.xy).xyz;
    half3 vBumpTexD = tex2D(tex0, Wave3.xy).xyz;

    // Average bump layers
    half3 vBumpTex=normalize(2.0 * (vBumpTexA.xyz + vBumpTexB.xyz +
                                    vBumpTexC.xyz + vBumpTexD.xyz) - 4.0);

    // Apply individual bump scale for refraction and reflection
    half3 vRefrBump = vBumpTex.xyz * half3(0.075, 0.075, 1.0);
    half3 vReflBump = vBumpTex.xyz * half3(0.02, 0.02, 1.0);

    // Compute projected coordinates
    half2 vProj = (ScreenPos.xy/ScreenPos.w);
    half4 vReflection = tex2D(tex2, vProj.xy + vReflBump.xy);
    half4 vRefrA = tex2D(tex1, vProj.xy + vRefrBump.xy);
    half4 vRefrB = tex2D(tex1, vProj.xy);

    // Mask occluders from refraction map
    half4 vRefraction = vRefrB * vRefrA.w + vRefrA * (1 - vRefrA.w);

    // Compute Fresnel term
    half NdotL = max(dot(vEye, vReflBump), 0);
    half facing = (1.0 - NdotL);
    half fresnel = Fresnel(NdotL, 0.2, 5.0);

    // Use distance to lerp between refraction and deep water color
    half fDistScale = saturate(10.0/Wave0.w);
    half3 WaterDeepColor = (vRefraction.xyz * fDistScale +
                            (1 - fDistScale) * half3(0, 0.15, 0.115));

    // Lerp between water color and deep water color
    half3 WaterColor = half3(0, 0.15, 0.115);
    half3 waterColor = (WaterColor * facing +
                        WaterDeepColor * (1.0 - facing));

    half3 cReflect = fresnel * vReflection;

    // final water = reflection_color * fresnel + water_color
    return half4(cReflect + waterColor, 1);
}
```


NVIDIA Ex1023
Page Q-342

19.3.2 Glass Simulation

Glass simulation is usually done using cube maps for both reflection and refraction, and then the results are blended using the Fresnel term. Some techniques go further and make refraction wavelength-dependent, to simulate chromatic aberration. In this example, we use only the texture S and a 2D reflection map to simulate the refraction and reflections on the glass. But this approach could be extended to include the previously mentioned techniques.

First we render the scene as usual to generate the S texture (with the refraction mask in alpha). The glass surface is then rendered by using environmental bump mapping blended with the refraction map using the per-pixel Fresnel term. The glass color comes from the diffuse texture modulated by the refraction. Figure 19-7 shows the three steps of this process.



Figure 19-7. The Rendering Steps for Stained Glass
(a) Refraction. (b) Environmental bump mapping. (c) Final composition.

By changing the bump texture, diffuse texture, and environment map, it's simple to simulate the appearance of different types of glass without having to change the fragment program itself. Figure 19-8 shows two examples, and the shader appears in Listing 19-5.

Listing 19-5. Shader for Refractive/Reflective Glass Simulation

```
half4 main(float2 BaseUV : TEXCOORD0,
  float4 ScreenPos : TEXCOORD1,
  float3 Eye : TEXCOORD2,
  uniform sampler2D tex0,
  uniform sampler2D tex1,
  uniform sampler2D tex2,
  uniform sampler2D tex3) : COLOR
{
  half3 vEye = normalize(Eye.xyz);

  // Get bump and apply scale, then get diffuse
  half4 vBumpTex = 2.0 * tex2D(tex1, BaseUV.xy) - 1.0;
```


NVIDIA Ex1023
Page Q-343



(a) (b)

Figure 19-8. Different Glass Types
(a) Stained glass. (b) Partition glass.

Listing 19-5 *(continued)*. Shader for Refractive/Reflective Glass Simulation

```
    half3 vBump = normalize(vBumpTex.xyz * half3(0.2, 0.2, 1.0));
    half4 vDiffuse = tex2D(tex0, BaseUV.xy);

    // Compute reflection vector
    half LdotN = dot(vBump.xyz, vEye.xyz);
    half3 vReflect = 2.0 * LdotN * vBump.xyz - vEye;

    // Reflection vector coordinates used for environmental mapping
    half4 vEnvMap = tex2D(tex3, (vReflect.xy + 1.0) * 0.5);

    // Compute projected coordinates and add perturbation
    half2 vProj = (ScreenPos.xy/ScreenPos.w);
    half4 vRefrA = tex2D(tex2, vProj.xy + vBump.xy);
    half4 vRefrB = tex2D(tex2, vProj.xy);

    // Mask occluders from refraction map
    half4 vFinal = vRefrB * vRefrA.w + vRefrA * (1 - vRefrA.w);

    // Compute Fresnel term
    half fresnel = Fresnel(LdotN, 0.4, 5.0);

    // Lerp between 1 and diffuse for glass transparency
    vDiffuse.xyz = saturate(0.1 + vDiffuse.xyz * 0.9);

    // Final output blends reflection and refraction using Fresnel term
    return vDiffuse * vFinal * (1 - fresnel) + vEnvMap * fresnel;
}
```


NVIDIA Ex1023
Page Q-344

19.4 Conclusion

In this chapter we have presented a technique to simulate refraction. The technique, though not physically based, produces results with good visual quality and is extremely efficient. The examples presented are just a few of the diverse effects that can be achieved.

One limitation of this technique is that, when applied to different-colored refractive surfaces, it will yield incorrect results where the surfaces overlap. As long as the refractive surfaces have a similar color, the visuals will look correct. One possible solution would be to sort refractive meshes from back to front and update the refraction map every time a refractive mesh is rendered. Alternatively, a less accurate solution would be to sort the refractive meshes from back to front and use alpha blending while rendering them.

The technique is also applicable to a range of target hardware: even though it's not demonstrated in this chapter, the technique can easily be expanded so that it works on lower-end hardware using pixel shader versions 1.1 to 1.4.

19.5 References

Akenine-Möller, Tomas, and Eric Haines. 2002. "Planar Reflections." In *Real-Time Rendering*, 2nd ed., pp. 239–243. A K Peters.

Everitt, Cass. 2001."Projective Texture Mapping." NVIDIA Corporation. Available online at **http://developer.nvidia.com/object/Projective_Texture_Mapping.html**

Kilgard, Mark J. 2001."Chromatic Aberration." NVIDIA Corporation. Available online at **http://developer.nvidia.com/object/chromatic_aberration.html**

Oliveira, Gustavo. 2000. "Refractive Texture Mapping, Part One." On Gamasutra Web site. Available online at **http://www.gamasutra.com/features/20001110/oliveira_01.htm**

Vlachos, Alex, John Isidoro, and Chris Oat. 2002. "Rippling Reflective and Refractive Water." In *ShaderX*, edited by Wolfgang Engel. Wordware.

Wloka, Matthias. 2002. "Fresnel Reflection Technical Report." NVIDIA Corporation.

A special thanks to Martin Mittring and Carsten Wenzel for ideas and discussion about artifact minimization and to Márcio Martins for his model loader used in the chapter demos.

NVIDIA Ex1023
Page Q-345



PART III
HIGH-QUALITY RENDERING

Robot image courtesy of Frantic Films

Every author of an interactive application is faced with a scale that must be balanced: deliver high-quality images, but deliver them at interactive speeds. While Moore's Law and the cleverness of GPU architects provide graphics programmers more raw horsepower with each generation of devices, fresh new techniques are also invented to get the most from the chips we have available today.

"High quality" is a slippery term, and one whose definition can change abruptly from year to year, with last year's innovative techniques being considered mundane by the next generation. What sorts of images could we achieve with the GPU's power if we were not constrained by real-time considerations? Horsepower never goes out of style—today's slow algorithm will eventually become tomorrow's fast one. Even in today's real-time applications, new GPUs are already capable of delivering image quality that would have been inaccessible only a year or two ago.

In **Chapter 20, "Fast Third-Order Texture Filtering,"** authors **Christian Sigg** of ETH Zurich and **Markus Hadwiger** of VRVis Research Center attack one of the fundamental challenges for all GPU-accelerated imaging: image filtering. They offer an efficient GPU-centric solution to complex filtering by leveraging the linear filtering already provided by GPUs. Their method delivers valid derivatives as well, with applications beyond simple 2D filtering into volume rendering and reconstruction of implicit surfaces.

Dan Wexler and **Eric Enderton** of NVIDIA, two of the authors of NVIDIA's GPU-accelerated final-film renderer Gelato, also confront both 2D image filtering and 3D imaging. In **Chapter 21, "High-Quality Antialiased Rasterization,"** they describe their method for filtering large, tiled images. Their method, as used in Gelato, is well suited to highly complex rendering that is performed in a large number of tiled pieces—and at resolutions that may be far higher than the rendering capacity of a single GPU pass.

NVIDIA Ex1023
Page Q-347

In **Chapter 22, "Fast Prefiltered Lines," Eric Chan** and **Frédo Durand** of the Massachusetts Institute of Technology present an accelerated method for improving the quality of an ubiquitous graphics entity that many users don't think about: simple lines. Chan and Durand draw lines using a CPU-prefiltered convolution that permits smooth line drawing from lookup table textures in the GPU, performed at interactive speeds. High-quality line drawing has application for game, CAD, and medical imaging, not just for 2D drawing but also for rendering such infamously problematic narrow 3D objects as telephone and power lines.

Hair-thin primitives come to the fore again with **Chapter 23, "Hair Animation and Rendering in the Nalu Demo,"** by **Hubert Nguyen** and **William Donnelly** of NVIDIA. They describe the breakthrough methods they employed for the animation and GPU rendering of "Nalu," a real-time demo whose innovative use of lighting and shadow techniques challenges the look of some of the best non-real-time character rendering from films and TV.

Rendering 3D images isn't the only application of GPUs in motion-picture production. In **Chapter 24, "Using Lookup Tables to Accelerate Color Transformations," Jeremy Selan** of Sony Pictures Imageworks describes how complex color manipulations of 2D images—sourced from 3D renderings or live-action plates—can be performed in real time by GPUs for the purposes of compositing and real-time preview on a variety of output media.

In a similar vein, **Pete Warden** of Apple Computer describes in **Chapter 25, "GPU Image Processing in Apple's Motion,"** a program that faced many of the same challenges and provided powerful real-time solutions on the desktops of professional and amateur video users. Warden's experience with GPUs in a heterogeneous, media-rich environment may help lead the way to a graphics-intensive rethinking of the computer usage experience as a whole.

Simon Green of NVIDIA revisits and expands upon a topic from *GPU Gems*, revealing his methods in **Chapter 26, "Implementing Improved Perlin Noise."** This GPU-friendly algorithm has application in both 2D and 3D imaging and delivers higher quality than the simple texture noise currently in vogue among game programmers.

NVIDIA Ex1023
Page Q-348

Justin Novosad of discreet takes yet another view of image filtering in **Chapter 27, "Advanced High-Quality Filtering."** He lays out a broad overview of the general problems of image reconstruction, and then he adds innovative ideas about image deblurring and even how to reduce live-action motion blur.

Finally, Climax Entertainment's **Iain Cantlay** brings high-quality issues to enlightened game programmers in **Chapter 28, "Mipmap-Level Measurement,"** providing ideas and tools to help programmers manage the complex balances between image quality, memory consumption, and rendering speed—not just for individual still images but also how they play out dynamically in an active, in-game environment.

Today's audiences—whether they are game players, movie watchers, or computer users—demand ever more rich, high-quality imagery, a trend that has been accelerating since at least the mid-1800s. It's in the interests of every GPU programmer to stay aware of these soon-to-be-standard methods for ensuring the best experiences in all media for today and tomorrow.

Kevin Bjorke, NVIDIA

NVIDIA Ex1023
Page Q-349