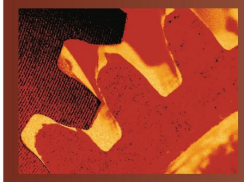
How GPUs Work

David Luebke, NVIDIA Research
Greg Humphreys, University of Virginia



GPUs have moved away from the traditional fixed-function 3D graphics pipeline toward a flexible general-purpose computational engine.

In the early 1990s, ubiquitous interactive 3D graphics was still the stuff of science fiction. By the end of the decade, nearly every new computer contained a graphics processing unit (GPU) dedicated to providing a high-performance, visually rich, interactive 3D experience.

This dramatic shift was the inevitable consequence of consumer demand for videogames, advances in manufacturing technology, and the exploitation of the inherent parallelism in the feed-forward graphics pipeline. Today, the raw computational power of a GPU dwarfs that of the most powerful CPU, and the gap is steadily widening.

Furthermore, GPUs have moved away from the traditional fixed-function 3D graphics pipeline toward a flexible general-purpose computational engine. Today, GPUs can implement many parallel algorithms directly using graphics hardware. Well-suited algorithms that leverage all the underlying computational horsepower often achieve tremendous speedups. Truly, the GPU is the first widely deployed commodity desktop parallel computer.

THE GRAPHICS PIPELINE

The task of any 3D graphics system is to synthesize an image from a description of a scene—60 times per second for real-time graphics such as videogames. This scene contains the geometric primitives to be viewed as well as descriptions of the lights illuminating the scene, the way that each object reflects light, and the viewer's position and orientation.

GPU designers traditionally have expressed this image-synthesis process as a hardware pipeline of specialized stages. Here, we provide a high-level overview of the classic graphics pipeline; our goal is to highlight those aspects of the real-time rendering calculation that allow graphics application developers to exploit modern GPUs as general-purpose parallel computation engines.

Pipeline input

Most real-time graphics systems assume that everything is made of triangles, and they first carve up any more complex shapes, such as quadrilaterals or curved surface patches, into triangles. The developer uses a computer graphics library (such as OpenGL or Direct3D) to provide each triangle to the graphics pipeline one vertex at a time; the GPU assembles vertices into triangles as needed.

Model transformations

A GPU can specify each logical object in a scene in its own locally defined coordinate system, which is convenient for objects that are naturally defined hierarchically. This convenience comes at a price: before rendering, the GPU must first transform all objects into a common coordinate system. To ensure that triangles aren't warped or twisted into curved shapes, this transformation is limited to simple affine operations such as rotations, translations, scalings, and the like.

As the "Homogeneous Coordinates" sidebar explains, by representing each vertex in homogeneous coordinates, the graphics system can perform the entire hierarchy of transformations simultaneously with a single matrix-vector multiply. The need for efficient hardware to perform floating-point vector arithmetic for millions of vertices each second has helped drive the GPU parallel-computing revolution.

The output of this stage of the pipeline is a stream of triangles, all expressed in a common 3D coordinate system in which the viewer is located at the origin, and the direction of view is aligned with the z-axis.

Lighting

Once each triangle is in a global coordinate system, the GPU can compute its color based on the lights in the scene. As an example, we describe the calculations for a single-point light source (imagine a very small lightbulb). The GPU handles multiple lights by summing the contributions of each individual light. The traditional graphics pipeline supports the Phong lighting equation (B-T. Phong, "Illumination for Computer-Generated Images," *Comm. ACM*, June 1975, pp. 311-317), a phenomenological appearance model that approximates the look of plastic. These materials combine a dull diffuse base with a shiny specular high-

ATI 2078
LG v. ATI
IPR2015-00326
AMD1318_0140542
ATI Ex. 2127
IPR2023-00922
Page 1 of 5

light. The Phong lighting equation gives the output color $C = K_d L_i (N \cdot L) + K_s L_i (R \cdot V)^s$.

Table 1 defines each term in the equation. The mathematics here isn't as important as the computation's structure; to evaluate this equation efficiently, GPUs must again operate directly on vectors. In this case, we repeatedly evaluate the dot product of two vectors, performing a four-component multiply-and-add operation.

Camera simulation

The graphics pipeline next projects each colored 3D triangle onto the virtual camera's film plane. Like the model transformations, the GPU does this using matrix-vector multiplication, again leveraging efficient vector operations in hardware. This stage's output is a stream of triangles in screen coordinates, ready to be turned into pixels.

Rasterization

Each visible screen-space triangle overlaps some pixels on the display; determining these pixels is called rasterization. GPU designers have incorporated many rasterizatiom algorithms over the years, which all exploit one crucial observation: Each pixel can be treated independently from all other pixels. Therefore, the machine can handle all pixels in parallel—indeed, some exotic machines have had a processor for each pixel. This inherent independence has led GPU designers to build increasingly parallel sets of pipelines.

Texturing

The actual color of each pixel can be taken directly from the lighting calculations, but for added realism, images called textures are often draped over the geometry to give the illusion of detail. GPUs store these textures in high-speed memory, which each pixel calculation must access to determine or modify that pixel's color.

In practice, the GPU might require multiple texture accesses per pixel to mitigate visual artifacts that can result when textures appear either smaller or larger on screen than their native resolution. Because the access pattern to texture memory is typically very regular (nearby pixels tend to access nearby texture image locations), specialized cache designs help hide the latency of memory accesses.

Hidden surfaces

In most scenes, some objects obscure other objects. If each pixel were simply written to display memory, the most recently submitted triangle would appear to be in front. Thus, correct hidden surface removal would require sorting all triangles from back to front for each view, an expensive operation that isn't even always possible for all scenes.

All modern GPUs provide a depth buffer, a region of memory that stores the distance from each pixel to the viewer. Before writing to the display, the GPU compares a pixel's distance to the distance of the pixel that's already present, and it updates the display memory only if the new pixel is closer.

THE GRAPHICS PIPELINE, EVOLVED

GPUs have evolved from a hardwired implementation of the graphics pipeline to a programmable computational substrate that can support it. Fixed-function units for transforming vertices and texturing pixels have been subsumed by a unified grid of processors, or *shaders*, that can perform these tasks and much more. This evolution has taken place over several generations by gradually replacing individual pipeline stages with increasingly programmable units. For example, the NVIDIA GeForce 3, launched in February 2001, introduced programmable vertex shaders. These shaders provide units that the programmer can use for performing matrix-vector multiplication, exponentiation, and square root calculations, as

Homogeneous Coordinates

Points in three dimensions are typically represented as a triple (x,y,z). In computer graphics, however, it's frequently useful to add a fourth coordinate, w, to the point representation. To convert a point to this new representation, we set $w = 1$. To recover the original point, we apply the transformation $(x,y,z,w) \rightarrow (x/w, y/w, z/w)$.

Although at first glance this might seem like needless complexity, it has several significant advantages. As a simple example, we can use the otherwise undefined point $(x,y,z,0)$ to represent the direction vector (x,y,z). With this unified representation for points and vectors in place, we can also perform several useful transformations such as simple matrix-vector multiplies that would otherwise be impossible. For example, the multiplication

$$\begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

can accomplish translation by an amount $\mathrm{D}x$, $\mathrm{D}y$, $\mathrm{D}z$.

Furthermore, these matrices can encode useful nonlinear transformations such as perspective foreshortening.

Table 1. Phong lighting equation terms.

Term	Meaning
K_d	Diffuse color
L_i	Light color
N	Surface normal
L	Vector to light
K_s	Specular color
R	Reflected light vector
V	Vector to camera
s	"Shininess"

AMD1318_0140543
ATI Ex. 2127
IPR2023-00922
Page 2 of 5



Figure 1. Programmable shading. The introduction of programmable shading in 2001 led to several visual effects not previously possible, such as this simulation of refractive chromatic dispersion for a "soap bubble" effect.



Figure 2. Unprecedented visual realism. Modern GPUs can use programmable shading to achieve near-cinematic realism, as this interactive demonstration shows, featuring actress Adrianne Curry on an NVIDIA GeForce 8800 GTX.

well as a short default program that uses these units to perform vertex transformation and lighting.

GeForce 3 also introduced limited reconfigurability into pixel processing, exposing the texturing hardware's functionality as a set of *register combiners* that could achieve novel visual effects such as the "soap-bubble" look demonstrated in Figure 1. Subsequent GPUs introduced increased flexibility, adding support for longer programs, more registers, and control-flow primitives such as branches, loops, and subroutines.

The ATI Radeon 9700 (July 2002) and NVIDIA GeForce FX (January 2003) replaced the often awkward register combiners with fully programmable pixel shaders. NVIDIA's latest chip, the GeForce 8800 (November 2006), adds programmability to the primitive assembly stage, allowing developers to control how they construct triangles from transformed vertices. As Figure 2 shows, modern GPUs achieve stunning visual realism.

Increases in precision have accompanied increases in programmability. The traditional graphics pipeline provided only 8-bit integers per color channel, allowing values ranging from 0 to 255. The ATI Radeon 9700 increased the representable range of color to 24-bit floating point, and NVIDIA's GeForce FX followed with both 16-bit and 32-bit floating point. Both vendors have announced plans to support 64-bit double-precision floating point in upcoming chips.

To keep up with the relentless demand for graphics performance, GPUs have aggressively embraced parallel design. GPUs have long used four-wide vector registers much like Intel's Streaming SIMD Extensions (SSE) instruction sets now provide on Intel CPUs. The number of such four-wide processors executing in parallel has increased as well, from only four on GeForce FX to 16 on GeForce 6800 (April 2004) to 24 on GeForce 7800 (May 2005). The GeForce 8800 actually includes 128 scalar shader processors that also run on a special *shader clock* at 2.5 times the clock rate (relative to pixel output) of former chips, so the computational performance might be considered equivalent to 128 ¥ 2.5/4 = 80 four-wide pixel shaders.

UNIFIED SHADERS

The latest step in the evolution from hardwired pipeline to flexible computational fabric is the introduction of

AMD1318_0140544
ATI Ex. 2127
IPR2023-00922
Page 3 of 5

unified shaders. Unified shaders were first realized in the ATI Xenos chip for the Xbox 360 game console, and NVIDIA introduced them to PCs with the GeForce 8800 chip.

Instead of separate custom processors for vertex shaders, geometry shaders, and pixel shaders, a unified shader architecture provides one large grid of data-parallel floating-point processors general enough to run all these shader workloads. As Figure 3 shows, vertices, triangles, and pixels recirculate through the grid rather than flowing through a pipeline with stages of fixed width.

This configuration leads to better overall utilization because demand for the various shaders varies greatly between applications, and indeed even within a single frame of one application. For example, a videogame might begin an image by using large triangles to draw the sky and distant terrain. This quickly saturates the pixel shaders in a traditional pipeline, while leaving the vertex shaders mostly idle. One millisecond later, the game might use highly detailed geometry to draw intricate characters and objects. This behavior will swamp the vertex shaders and leave the pixel shaders mostly idle.

These dramatic oscillations in resource demands in a single image present a load-balancing nightmare for the game designer and can also vary unpredictably as the players' viewpoint and actions change. A unified shader architecture, on the other hand, can allocate a varying percentage of its pool of processors to each shader type.

For this example, a GeForce 8800 might use 90 percent of its 128 processors as pixel shaders and 10 percent as vertex shaders while drawing the sky, then reverse that ratio when drawing a distant character's geometry. The net result is a flexible parallel architecture that improves GPU utilization and provides much greater flexibility for game designers.

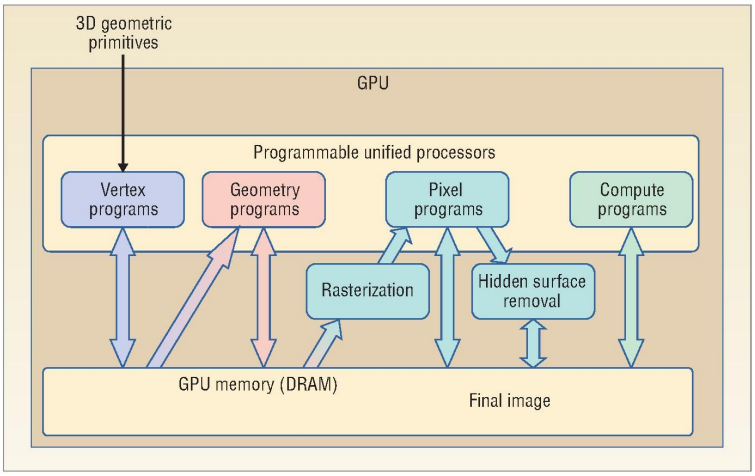


Figure 3. Graphics pipeline evolution. The NVIDIA GeForce 8800 GPU replaces the traditional graphics pipeline with a unified shader architecture in which vertices, triangles, and pixels recirculate through a set of programmable processors. The flexibility and computational power of these processors invites their use for general-purpose computing tasks.

GPGPU

The highly parallel workload of real-time computer graphics demands extremely high arithmetic throughput and streaming memory bandwidth but tolerates considerable latency in an individual computation since final images are only displayed every 16 milliseconds. These workload characteristics have shaped the underlying GPU architecture: Whereas CPUs are optimized for low latency, GPUs are optimized for high throughput.

The raw computational horsepower of GPUs is staggering: A single GeForce 8800 chip achieves a sustained 330 billion floating-point operations per second (Gflops) on simple benchmarks (http://graphics.stanford.edu/projects/gpubench). The ever-increasing power, programmability, and precision of GPUs have motivated a great deal of research on general-purpose computation on graphics hardware—GPGPU for short. GPGPU researchers and developers use the GPU as a computational coprocessor rather than as an image-synthesis device.

The GPU's specialized architecture isn't well suited to every algorithm. Many applications are inherently serial and are characterized by incoherent and unpredictable memory access. Nonetheless, many important problems require significant computational resources, mapping well to the GPU's many-core arithmetic intensity, or they require streaming through large quantities of data, mapping well to the GPU's streaming memory subsystem.

Porting a judiciously chosen algorithm to the GPU often produces speedups of five to 20 times over mature, optimized CPU codes running on state-of-the-art CPUs, and speedups of more than 100 times have been reported for some algorithms that map especially well.

Notable GPGPU success stories include Stanford University's Folding@home project, which uses spare cycles that users around the world donate to study protein folding (http://folding.stanford.edu). A new GPU-accelerated Folding@home client contributed 28,000 Gflops in the month after its October 2006 release—more than 18 percent of the total Gflops that CPU clients contributed running on Microsoft Windows since October 2000.

In another GPGPU success story, researchers at the University of North Carolina and Microsoft used GPU-based code to win the 2006 Indy PennySort category of the TeraSort competition, a sorting benchmark testing price/performance for database

AMD1318_0140545
ATI Ex. 2127
IPR2023-00922
Page 4 of 5

operations (http://gamma.cs.unc.edu/GPUTERASORT). Closer to home for the GPU business, the HavokFX product uses GPGPU techniques to accelerate tenfold the physics calculations used to add realistic behavior to objects in computer games (www.havok.com).

Modern GPUs could be seen as the first generation of commodity data-parallel processors. Their tremendous computational capacity and rapid growth curve, far outstripping traditional CPUs, highlight the advantages of domain-specialized data-parallel computing.

We can expect increased programmability and generality from future GPU architectures, but not without limit; neither vendors nor users want to sacrifice the specialized architecture that made GPUs successful in the first place. Today, GPU developers need new high-level programming models for massively multithreaded parallel computation, a problem soon to impact multicore CPU vendors as well.

Can GPU vendors, graphics developers, and the GPGPU research community build on their success with commodity parallel computing to transcend their computer graphics roots and develop the computational idioms, techniques, and frameworks for the desktop parallel computing environment of the future?

David Luebke is a research scientist at NVIDIA Research. Contact him at dluebke@nvidia.com.

Greg Humphreys is a faculty member in the Computer Science Department at the University of Virginia. Contact him at humper@cs.virginia.edu.

Computer welcomes your submissions to this bimonthly column. For additional information, or to suggest topics that you would like to see explained, contact column editor Alf Weaver at weaver@cs.virginia.edu.

IEEE

Innovative Technology for Computer Professionals

Computer

Welcomes Your Contribution

Computer magazine looks ahead to future technologies

IEEE computer society

- **Computer**, the flagship publication of the IEEE Computer Society, publishes peer-reviewed technical content that covers all aspects of computer science, computer engineering, technology, and applications.
- Articles selected for publication in **Computer** are edited to enhance readability for the nearly 100,000 computing professionals who receive this monthly magazine.
- Readers depend on **Computer** to provide current, unbiased, thoroughly researched information on the newest directions in computing technology.

To submit a manuscript for peer review, see *Computer*'s author guidelines:

www.computer.org/computer/author.htm

AMD1318_0140546

ATI Ex. 2127

IPR2023-00922

Page 5 of 5