



Texture Caches

MICHAEL DOGETT
Lund University

..... The texture cache is an essential component of modern GPUs and plays an important role in achieving real-time performance when generating realistic images. The texture cache is a read-only cache that stores image data used for putting images onto triangles, a process called *texture mapping*. Figure 1 shows a typical real-time graphics scene rendered with and without texture mapping, which adds color details to the triangle models that make up the 3D scene. The texture cache has a high hit rate because there's heavy reuse between neighboring pixels and is usually located close to the shader processor so texture data is available with high throughput and at low read latencies.

GPUs are large complex systems, and the texture cache is only a small component. Blythe presents an overview of GPUs and their evolution.¹ Blythe points out that texture mapping was an early requirement for games and one of the first features to move from high-end research systems to PC and console gaming. The GPU has evolved in a competitive market that has seen most companies disappear from the many that existed in the late 1990s. The last major GPU vendor, Nvidia, has expanded beyond just GPU design and also builds integrated GPU and CPU SOCs for the mobile market.

GPU architecture isn't visible to the programmer and is hidden under the

programming APIs (such as OpenGL and Direct3D) and so architecture, including texture caches, has changed frequently. Figure 2 shows a modern GPU's generalized top-level architecture. The texture cache, L1 cache, and L2 cache make up the GPU's memory hierarchy.

To get more detail about GPU architecture is challenging. One of the most detailed lower-level descriptions of how the graphics algorithms and units inside a GPU interact with memory, including the texture cache, is the description of the Neon GPU architecture by McCormack et al.² But the technology used is dated, so some concepts are no longer relevant. For more modern GPUs, Wong et al.

use extensive benchmarking to determine the CUDA-visible characteristics of Nvidia's GT200 architecture.³ They determine processing-element and memory-hierarchy characteristics.

On the topic of texture cache design, there's also little literature. Hakura and Gupta's seminal paper gives the best overview of the principles of a texture cache, including the effects of data locality, texture blocking, and tiled rasterization, all topics that I will revisit in this column.⁴ To understand the requirements of a texture cache, we first must look at the units in a GPU that impact the cache functionality—that is, its client, texture mapping, and the unit that determines pixel ordering, rasterization.

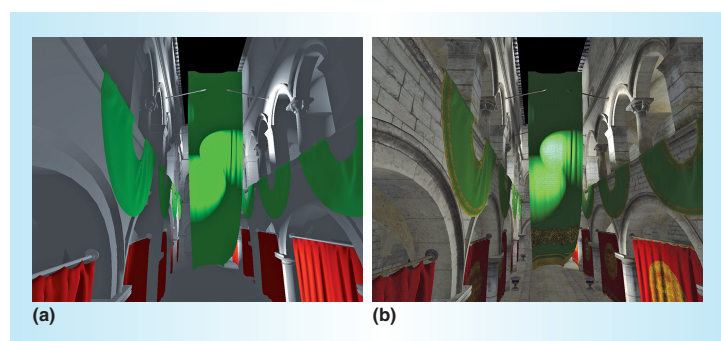


Figure 1. The sponza atrium palace model with colored triangles and lighting (a) and with texture mapping (b). In (b), the details of the bricks and patterns on the curtains can be seen. Also, the spotlight highlights changes in the lighting on the center curtain by using normal perturbations stored in a normal map.

Texture mapping

Texture mapping is the process of taking image data and placing it onto a surface. The image data is typically 2D, but can also be 1D and 3D. At each pixel on the screen, the corresponding value has to be found in the image data, commonly known as the texture map. The individual elements in the texture map are called *texels* (from “texture elements”) to differentiate them from the screen pixels. Each vertex of a triangle contains a 2D position on the texture map. These positions are called texture coordinates and are usually denoted by u, v . Figure 3 shows a pixel in a triangle using its u, v coordinate to look up the corresponding texel in the texture map, which is in s, t coordinates. The texture map texels are in their frame of reference s, t , which ranges from 0 to 1 along the s and t axes. The u and v coordinates typically range from 0 up to the texture’s width and height, but can also go beyond to allow effects such as texture wrapping. The texture coordinates are calculated by interpolating values at each vertex, and then the coordinates are used to look up the color applied to the current pixel. Looking up a single color on the basis of the integer coordinates is called *nearest filtering*, and results in aliasing. To avoid this aliasing, filtering of neighboring pixels is used. The first type of filtering is bilinear, in which linear interpolation of neighboring pixels in the X and Y dimensions is computed. This smooths out the texture mapping, but doesn’t handle minification of the texture, where many texels should be averaged into the current pixel.

To handle minification, texture mapping uses a technique called *mipmapping*.⁵ Mipmapping prefilters the texture map into a series of maps that are half the dimensions in X and Y. The sequence of maps can be stacked in a pyramid from the largest map at the bottom to the smallest map at the top. Figure 4 shows a mipmap stack with the highest resolution image at the bottom (level 0), and subsequent levels going up, each

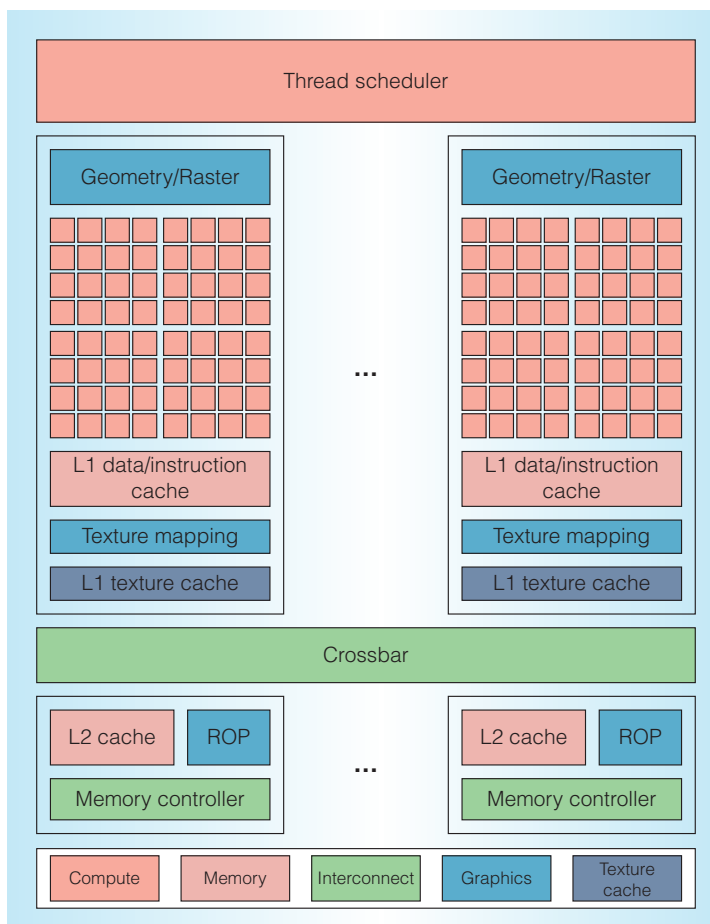


Figure 2. A simplified diagram of a modern GPU’s top-level architecture. The texture cache is located just above the crossbar and at the bottom of the shader. The thread scheduler collects workloads for the different stages of the graphics pipeline and loads them onto a variable number of graphics cores. The graphics cores have some fixed-function graphics logic in the geometry/raster unit, which contains geometry processing and rasterization. Underneath is the shader, which has control logic (not shown) and a grid of arithmetic logic units (ALUs) that process small programs in a single-instruction, multiple-data (SIMD) fashion. The shader uses the L1 data and instruction caches, texture mapping unit, and its associated texture cache as the programs are executed. The output of the shader and the connection of the L1s to the L2s goes via a crossbar, which is connected to a variable number of memory units containing the L2 cache, the ROP (fixed-function graphics), and the memory controller. The memory controller is connected to the external memory, where the framebuffer is stored. Most neighboring blocks are connected to each other, so interconnections between blocks are not shown.

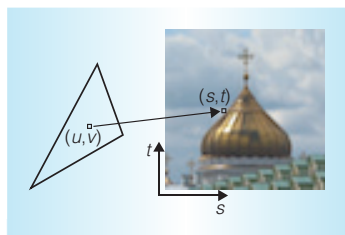


Figure 3. Texture mapping involves finding the corresponding color in a 2D image map in s, t coordinates using the texture coordinates u, v at the current pixel in a triangle.

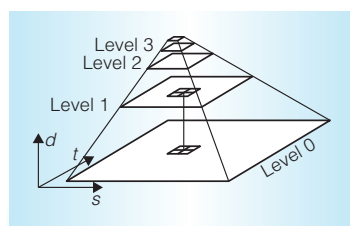


Figure 4. A sequence of images making a mipmap pyramid. Level 0 is the full resolution image, and each subsequent level is a half-size averaged version of the previous level. Texture filtering first calculates the level of detail, d , which selects two levels of the pyramid and then performs trilinear filtering between the two layers.

half the previous level's resolution. Trilinear mipmapping works by finding two neighboring maps in the pyramid and performing bilinear filtering on each map and then a linear interpolation between the two values, resulting in trilinear filtering. This can be seen in Figure 4, where levels 0 and 1 are selected and a neighborhood of 2×2 pixels are found for bilinear filtering on each level. To select the two neighboring maps, the level of detail (LOD) must be calculated by taking the difference in texture coordinates between neighboring pixels in a 2×2 grid. More complex filtering such as anisotropic filtering is used in texture mapping,⁶ but it's typically done using a

sequence of bilinear filtering operations making its requirements on the texture cache similar to bilinear filtering.

Rasterization

Rasterization is the process of breaking up triangles into screen pixels. Triangles are the basic primitive used by GPUs, because by limiting the rasterization process to using triangles, GPUs can optimize the algorithm for implementation in custom hardware. The simplest order in which to generate pixels from a triangle is to move down the triangle's edges and scan across each horizontal line from one edge to the other. Although this is straightforward, it's not the most optimal, and so the order is changed to improve performance.

GPU rasterization also groups pixels into 2×2 blocks, commonly referred to as quads. Each pixel stores its u, v coordinates, and the texture gradients, $\delta u/\delta x, \delta v/\delta x, \delta u/\delta y, \delta v/\delta y$, are calculated by taking the difference of two neighboring pixels' texture coordinates. These gradients are then used to calculate the LOD as specified in the OpenGL specification. If a triangle doesn't cover the entire quad, the GPU still pads out the quad with null pixels, which occupy space in the GPU compute engines. This leads to single pixel triangles being very inefficient for GPUs, but Fatahalian et al. have proposed solutions to packing these pixels together when they're part of a triangle strip/mesh.⁷

Texture map memory

The order of texels accessed in a texture map can have any orientation with respect to the rasterization order. If a texture map's texels are stored in memory in a simple linear row major ordering, and the texture is mapped horizontally, then cache lines will store long horizontal lines resulting in a high hit rate. But if the texture is mapped at a 90-degree rotation, then every pixel will miss and require a new cacheline to be loaded. To avoid this, orientation dependency textures are stored in memory in a

tilled, or blocked, fashion.⁴ Each texture is broken into a series of $n \times n$ tiles, and all the tile's texels are stored before moving onto the next tile.

Hakura and Gupta found that higher cache hit rates are achieved when the tile size is equal to cache line size for cache sizes of 128 Kbytes and 256 Kbytes and tile sizes of 8×8 and 16×16 .⁴ Another important aspect of memory usage is the concept of working set size. The working set size to render one frame is proportional to the resolution of the image, according to the principle of texture thrift.⁸ Hakura and Gupta found that if the working set fits into the cache, miss rates are further reduced.⁴

Texture caches are read only because the typical usage of a texture is for it to be placed onto a triangle. But computing the values that go into a texture map using the GPU—called *render to texture*—has become commonplace. So, even though a texture cache is typically read-only and the memory buffer in the GPU's main memory is tagged as read-only, it can also be changed to writeable. If a texture map's state is changed to writeable, all on-chip memory references to that texture must be invalidated, and the GPU can write to the texture. A simpler brute force method, which flushes all caches in the GPU, can also be used. This cache invalidation is required to ensure that the GPU's many specialized caches, such as the texture cache, are maintained in a coherent state.

Tiled rasterization

Similarly to tiling for texture maps in memory, tiling also improves the access order of texture data when used for rasterization. A simple rasterization process could generate pixels in a row major order, starting at the left-hand side of a triangle and generating pixels across the row to the right-hand side. For very large triangles, a texture cache could fill up as pixels are generated across a horizontal span and on returning to the next line, find that the texels from the

previous line have been evicted from the cache. This horizontal rasterization results in texture fetches that vary in terms of spatial locality across the width of the triangle. To improve this spatial locality, GPUs use tiled rasterization instead. Tiled rasterization divides the screen into equally sized rectangles, typically powers of two, and completes the rasterization within each tile before moving to the next tile. The order of texture access can be further improved by using hierarchical orders such as the Hilbert curve⁹ or Z-order.

Texture cache architecture

The texture cache's role has changed with the introduction of read/write L2 caches in GPUs such as Nvidia's Fermi¹⁰ and AMD's GCN.¹¹ Hakura and Gupta presented single-level, per-pipeline units and considered cache sizes of 4 Kbytes, 32 Kbytes, and 128 Kbytes.⁴ In terms of cache size, the modern GPU texture cache L1 is approaching a similar size—for example, it's 12 Kbytes in the Nvidia Fermi and 16 Kbytes for the AMD GCN. The texture cache changed into a two-level cache, as seen in the Nvidia G80 architecture.¹² The G80 had an L2 cache distributed to each DRAM channel. The L2 then connects to the texture L1 units via a large crossbar (see Figure 2). The per-memory controller L2 is now a common GPU memory architecture design. By placing one unit of the L2 at the memory controller, the GPU can be easily scaled in terms of memory controllers, independent of the number of L1s and, hence, shader units. When designing the memory hierarchy, making this L2 cache a linear memory mapped cache is better than customizing it for texture. This allows the L2 cache to have more than just texture as a client and has made for a more straightforward transition to the modern read/write L2.

Starting from the top-level GPU architecture in Figure 2, we see that the texture cache is located between the texture filtering unit and the crossbar. Figure 5 shows a texture cache's internal

architecture. Boundaries between blocks in a GPU are arbitrary, so some of the operations could instead be placed in a texture filtering unit, but I show them here to get a better understanding of what could be required of a texture cache.

The texture cache is designed to connect well with the texture unit. The texture unit requires four texels in parallel to perform one bilinear operation, so most texture caches are designed to feed these values in parallel to maintain full rate for bilinear filtering. Most GPUs have multiple texture-filtering units running in parallel, which the texture cache must supply with texels. To do this, the texture cache uses multiple read ports, or else replicates the data in the cache.

Texture filtering and texture cache units have to support every format the graphics APIs offer. Which formats are supported for custom hardware texture filtering can vary greatly depending on a GPU vendor's best estimate of what requirements are important for current and future games. The alternative is to support texture filtering in the shader at much lower performance. If a format is only supported by the texture cache, it could be required to do the final formatting, because the value doesn't go through the texture filtering unit. Texture formats can include bit widths from 1 to 32 bits and number formats including integer and floating point. If these formats are supported, final formatting can be done by the texture cache instead of the texture filtering, and the texture cache can include logic to shift bits and convert texels into the correct formats for the shader.

Texture compression is an important component of ensuring texturing performance by reducing texture bandwidth. Because decompression of the texture happens somewhere between reading the compressed texture from memory and giving texels ready for texture filtering to the texture pipe, texture compression can sometimes be found within the texture cache design.

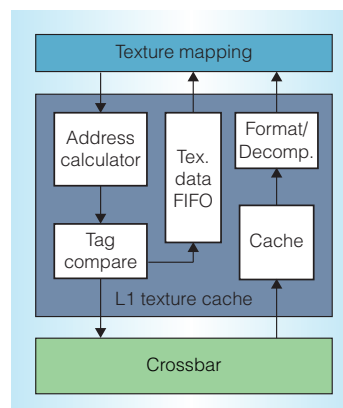


Figure 5. Overview of a texture cache architecture. The texture mapping unit provides texture coordinates for which a memory address is calculated. The address is sent to the tag compare to determine if the data is in the cache. If the data isn't in the cache, a request is sent via the crossbar to the L2 cache. Any state associated with the original request is sent into a first-in, first-out buffer (FIFO) to return to the texture mapping unit with the texel data. Once the data arrives in the cache, or is already available in the cache, it is returned to the texture mapping unit. If the data is compressed, it is decompressed and any necessary formatting is done.

Texture decompression is usually done after the cache because storing compressed data in the cache means more texture data gets cached and the cache is much more effective. For example, the most popular texture formats, S3TC¹³ (DXTC in Direct3D) and ETC (used in OpenGL ES), have a 6:1 compression ratio for RGB textures. Another important aspect of texture compression is that the logic must be small. This is necessary because wherever the logic is located, it will be replicated many times, making any added complexity very expensive. This has led to a slow adoption of new texture compression formats, with new ones only recently being added in Microsoft's Direct 3D 11.

Performance

Texture cache design is heavily dictated by performance. It's important that a texture cache can sustain the full external memory bandwidth with texture data to the texture unit. An important design target for the texture unit is to be able to perform one bilinear filtering operation per clock cycle, which requires four texels. This means that the texture cache must be able to deliver 4×32 bits per clock to the texture unit. For an L1 cache with four texture unit clients, it's four times that. Given a target hit rate determined from a cross section of current games, both an input bandwidth to the L1 cache and a corresponding output bandwidth from the crossbar can be determined. This output must then be sustained by the L2 cache as well.

When a cache miss occurs and a new cache line must be read in from off-chip memory, a large latency is incurred. Igehy et al. propose to hide this latency by precomputing the addresses required and prefetching the cachelines.¹⁴ Any associated texture filtering state that is needed when the texel data is returned to the texture filter is stored in a large first-in, first-out (FIFO) buffer while it waits for texel data to be returned from the cache. This means that texture accesses that hit in the cache must wait in the FIFO buffer behind others that miss. The FIFO buffer is used because the graphics pipeline ensures that triangle submission order is maintained, so allowing some texture fetches to complete early wouldn't have a large impact on performance. Storing the texture requests in a FIFO buffer hides the several hundred clock-cycle latency from the time a memory request is made in the shader pipe until data is returned and the shader can resume. These long latencies are hidden by the GPU running many thousands of threads,¹² which enable the GPU to switch to other work immediately and ensure that the GPU's computational units are kept active while some threads

wait for memory to load. These threads run on the shader with all register space allocated in shader memories, so that single-cycle thread switching is possible, ensuring no cycles are lost waiting for memory requests.

Although the basic principles of cache performance haven't changed,⁴ many of the parameters have changed dramatically. For example, they worked with a total texture size of between 1 and 56 Mbytes, compared to modern PC games, which could easily use half of the available memory on PC cards, which range up to 3 Gbytes. Most major games are cross-platform titles, and thus are limited by the consoles' memory, which is around 512 Mbytes. A game could easily use half of that memory for texture. This leads to higher-resolution textures in modern games compared to what Hakura and Gupta worked with. At the extreme end, technology such as id Tech 5's virtual texturing can support $128,000 \times 128,000$, although this technology only uses a small amount of GPU memory for swapping small texture tiles in and out. This increase in texture storage leads to an increase in textures' number, size, and resolution. Textures on mobile platforms have similar limits with high-end smart phones also having 512 Mbytes of memory. But mobile platforms are more impacted by power usage, and if a lot of texture memory is used, the power usage can be increased. In the future, texture memory's usage will follow the technology; as memory gets bigger, so will texture usage. For the mobile platforms, a steady growth can be expected. But for the consoles, a new generation is expected soon, and will bring with it larger memory space for textures, most likely enabling console games to have more texture than mobile games. This disparity in available texture memory, which can occur between different PC graphics cards, can be handled by using higher-resolution textures when more memory is available.

The amount of repeated texture has decreased over recent years in games. The test scenes Hakura and Gupta used had very large triangles in terms of pixel area (41, 294, 1149, and 186 pixels). Most modern games have pixel areas per triangle of less than 10 pixels, driven in part by modern tessellation engines such as in Nvidia's Fermi.¹⁰ The current limit to game triangle size is the GPU's quad, which means that going below four pixels severely impacts GPU shader utilization.

Associativity also impacts cache performance. Hakura and Gupta showed that two-way set associativity worked well to reduce cache-line conflicts between mip-map layers, but modern applications often use many textures per pixel, requiring higher levels of associativity. For example, Wong et al. find the associativities of the GT200 to be 20-way for the L1 cache and eight-way for the L2 cache.³

Modeling the texture cache and the GPU's memory hierarchy can help to better understand its performance. Although this is difficult, early work shows that this can be done for applications such as real-time ray tracing.¹⁵

General-purpose GPU

In recent years, using GPUs for general-purpose parallel computing has become commonplace,¹⁶ and is typically referred to as GPGPU. Texture mapping is part of the dedicated hardware designed for graphics rendering, but it can also be accessed in GPGPU languages such as OpenCL by using the image and sampler objects. In modern read/write L2 cache GPU architectures, the texture cache doesn't offer a great advantage over the shader load/store L1 cache, except that it's an additional cache that's otherwise unused. The texture filtering custom hardware can be put to good use in GPGPU applications, if a good match for the filtering operations can be found.

GPGPU is pushing GPU architecture into a more general direction, as evidenced by the introduction of the read/write L2 cache. Other features unused by graphics are also improving in GPUs,

such as atomics, 64-bit floating-point performance, and support for error correcting codes (ECC). These features, while desirable for GPGPU, take up chip area for graphics, making the GPU more expensive. A solution to this problem is to make two differently configured GPUs from the same architecture—one configured for graphics and one for GPGPU.

The consistent economic conditions that existed in the first decade of the 21st century and drove the GPU from a simple graphics display adaptor into a massively parallel device are not expected to continue in the second decade. With CPU and GPU integration becoming more mainstream, and with an increase in portable devices' graphics performance, the dominant graphics device will be an integrated one. At the high end, the GPU is driven by a small market for high-end graphics and GPGPU. Today, neither market is large enough to fund the research expense required for high-end GPUs. The expense still requires the mid-range graphics market. So it has been risky to produce chips just for GPGPU, but this is starting to change with the continuing growth of the GPGPU market. The console market, on the other hand, takes a GPU design and keeps it constant over many years, typically around five years. This allows custom GPUs to be designed for a particular console without the need for all the GPGPU features. If the GPGPU market can sustain the development of a high-end device without graphics, hardware vendors could build these devices without texture caches. If texturing performance could be made to be acceptably close to dedicated hardware, then it's possible that texture hardware and the associated texture cache could be removed from GPUs. But graphics performance is still an important factor for GPU sales and can't be neglected while two strong competitors exist. So, even though different markets require different memory architectures, GPUs will continue to be built for both markets, and a texture cache

will continue to be necessary for high performance.

Acknowledgments

Thanks to Jocelyn Houle and Tomas Akenine-Möller for proofreading. I acknowledge support from the Intel Visual Computing Institute, Saarbrücken, Germany and the ELLIIT Excellence Center at Linköping-Lund in Information Technology. The Sponza Atrium Palace model is courtesy of Crytek.

References

1. D. Blythe, "Rise of the Graphics Processor," *Proc. IEEE*, vol. 96, no. 5, 2008, pp. 761-778.
2. J. McCormack et al., "Neon: A Single-Chip 3D Workstation Graphics Accelerator," *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware*, ACM, 1998, pp. 123-132.
3. H. Wong et al., "Demystifying GPU Microarchitecture through Microbenchmarking," *IEEE Int'l Symp. Performance Analysis of Systems and Software*, IEEE CS, 2010, pp. 235-246.
4. Z.S. Hakura and A. Gupta, "The Design and Analysis of a Cache Architecture for Texture Mapping," *Proc. 24th Ann. Int'l Symp. Computer Architecture*, ACM, 1997, pp. 108-120.
5. L. Williams, "Pyramidal Parametrics," *Proc. 10th Ann. Conf. Computer Graphics and Interactive Techniques*, ACM, 1983, doi:acm.org/10.1145/800059.801126.
6. A. Schilling, G. Knittel, and W. Strasser, "Texram: A Smart Memory for Texturing," *IEEE Computer Graphics and Applications*, vol. 16, no. 3, 1996, pp. 32-41.
7. K. Fatahalian et al., "Reducing Shading on GPUs using Quad-Fragment Merging," *ACM Trans. Graphics*, vol. 29, no. 4, 2010, pp. 67:1-67:8.
8. D. Peachey, "Texture on Demand," tech. memo, Pixar, 1990.
9. M.D. McCool, C. Wales, and K. Moule, "Incremental and Hierarchical Hilbert-Order Edge Equation Polygon Rasterization," *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware*, ACM, 2001, pp. 65-72.
10. C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, no. 2, 2011, pp. 50-59.
11. M. Mantor and M. Houston, "AMD Graphic Core Next," *High-Performance Graphics*, Hot3D presentation, 2011.
12. E. Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, 2008, pp. 39-55.
13. K. Iourcha, K. Nayak, and Z. Hong, *System and Method for Fixed-Rate Block-Based Image Compression with Inferred Pixel Values*, US patent 5,956,431, Patent and Trademark Office, 1999.
14. H. Igehy, M. Eldridge, and K. Proudfoot, "Prefetching in a Texture Cache Architecture," *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware*, ACM, 1998, doi:10.1145/285305.285321.
15. P. Ganestam and M. Doggett, "Auto-tuning Interactive Ray Tracing using an Analytical GPU Architecture Model," *Proc. 5th Ann. Workshop General-Purpose Processing with Graphics Processing Units*, ACM, 2012, pp. 94-100.
16. J.D. Owens et al., "GPU computing," *Proc. of the IEEE*, vol. 96, no. 5, 2008, pp. 879-899.

Michael Doggett is an associate professor in the Lund University Graphics Group (LUGG) at Lund University. His research interests focus on image synthesis, graphics hardware, and parallel architectures. Doggett has a PhD in computer science and engineering from the University of New South Wales.

Direct questions or comments about this article to Michael Doggett at mike@cs.lth.se.

 Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.