THE GEFORCE 6800

GRAPHICS PROCESSING UNITS (GPUS) CONTINUE TO TAKE ON INCREASING COMPUTATIONAL WORKLOADS AND TODAY SUPPORT INTERACTIVE RENDERING THAT APPROACHES CINEMATIC QUALITY. THE ARCHITECTURAL DRIVERS FOR GPUS ARE PROGRAMMABILITY, PARALLELISM, BANDWIDTH, AND MEMORY CHARACTERISTICS. THIS ARTICLE DESCRIBES HOW ONE TEAM APPROACHED THE DESIGN PROBLEM.

John Montrym
Henry Moreton
Nvidia

The graphics processing unit (GPU) market is large, growing, and varied, shipping more than 500 million units per year. Table 1 profiles this market. The core GPU market is interactive gaming on the PC platform, where the goal is film-quality rendering with real-time response. Game releases rival movie openings in revenue. The release of Halo 2, an Xbox title, grossed $125 million in the first 24 hours (www.pcmag.com). In contrast, *The Incredibles* grossed $70.5 million during its first three days (www.the-numbers.com).

In addition to workstations used to develop motion pictures and games, GPU markets include traditional professional workstations, flight and driving simulators, and various consumer devices. General-purpose computing using GPUs is both an area of research and an emerging market. GPUs are well suited for large data-parallel problems such as fluid dynamics, weather simulation, and financial option price modeling.

The computational load on GPUs keeps growing, and image quality has made huge strides during the last 15 years. Figure 1 illustrates the evolution of image quality.

Table 1. Graphics processing unit market breakdown.

Sector	Millions of Units
Interactive gaming	50
Digital content creation	
professional	1
home	50
Computer-aided design and manufacture	1
Visual simulations	0.1
General computing	3
Consumer	
handheld devices	50
consoles	100
media centers	5
cell phones	600
Total	860.1

The graphics problem

What does a GPU do? Under the control of an application generically called a *renderer*, the GPU computes the color of each pixel. This *image synthesis* entails resampling a scene described by triangles of materials simulated using sampled images (textures) and numerically approximated properties. The GPU performs image synthesis calculations in three steps. First, it processes the triangles' vertices,

0272-1732/05/$20.00 © 2005 IEEE Published by the IEEE computer Society

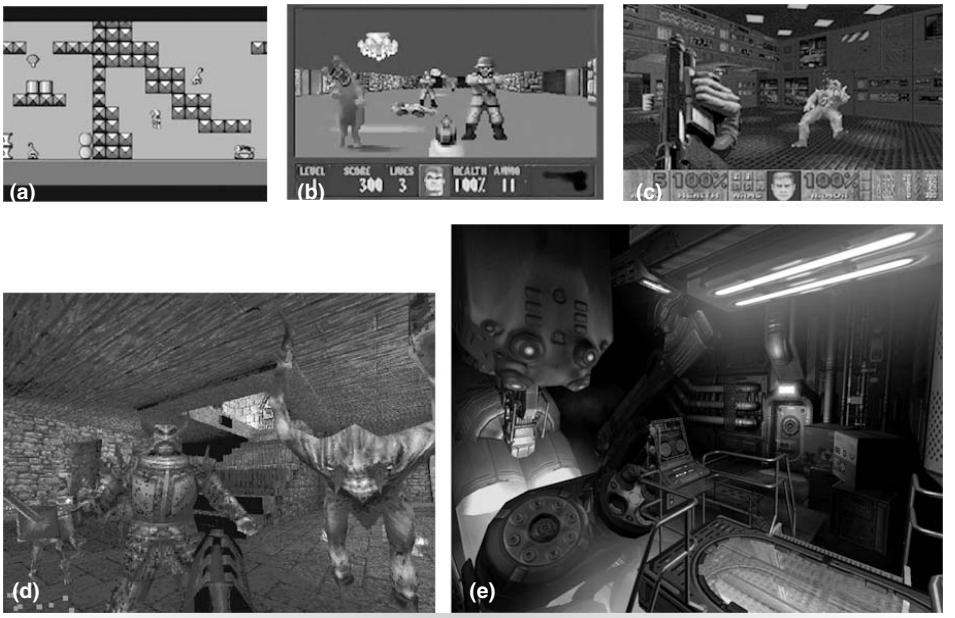

Figure 1. Evolution of image quality in PC games. In the 1990 game Marooned, the PC state of the art was two-dimensional sprites, and the graphics card was little more than a CRT controller (a). Simple three-dimensional graphics appeared in 1991, as shown in Hovertank (b). The Doom series introduced texture mapping of simple characters in 1993 (c). Quake, in 1996, brought greater quality, texture filtering, and more characters (d). Today, with Doom3, we see correct shadows, accurate lighting models, and high-quality filtering (e). *(Images used by permission of Id Software Inc. Wolfenstein 3D, DOOM, QUAKE, and DOOM 3 are either registered trademarks or trademarks of Id Software Inc. in the United States and/or other countries.)*

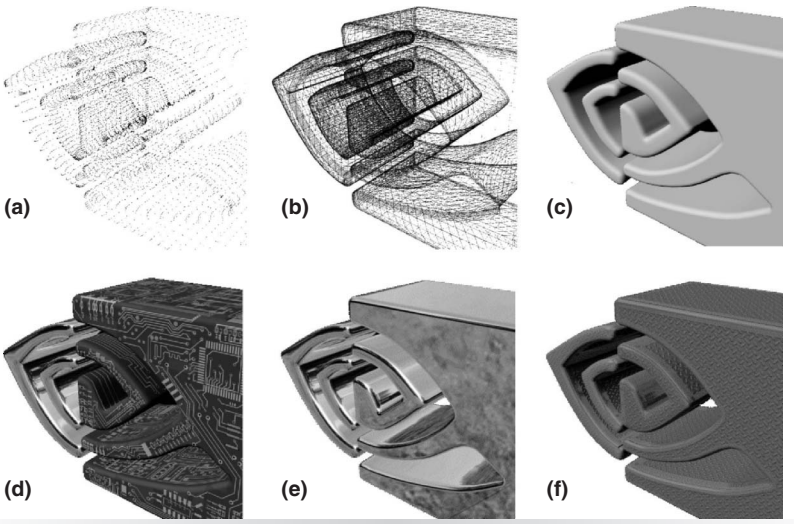

Figure 2. Basic primitives used in rendering. The renderer approximates objects using triangles, defined by vertices (a) and lines connecting the vertices to make triangles (b). The simplest form of lighting assumes a perfectly diffuse surface (c). In simple texture mapping, the GPU samples and filters images to determine pixel fragment color (d). Given the eye location and surface orientation at the fragment, the GPU can look up a reflected color (e) in a texture called an environment, or cube, map to perform reflection mapping. The GPU can also simulate bumpiness (bump mapping) by perturbing local surface orientation (f).

computing screen positions and attributes such as color and surface orientation. Next, a rasterizer samples each triangle to identify fully and partially covered pixels, called fragments. Finally, it processes the fragments using texture sampling, color calculation, visibility, and blending. The vertex and fragment processing steps enjoy a high degree of independent programmable processing.

How does a rendering application use the GPU to simulate the appearance of materials? Figure 2 shows a progression of scene drawing techniques.

The desire for increased realism has driven greater precision and functionality. A recent example is high-dynamic-range (HDR) rendering.[1] In Figure 3, for example, the light through the window is hundreds of times brighter than the obelisks, but the obelisks are not solid black. The glow produces a more cinematic image.

Figure 3. High-dynamic-range rendering.

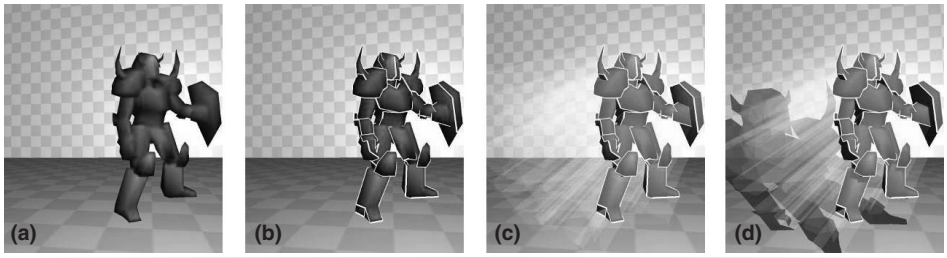

Figure 4. Steps in stencil shadow-volume calculation. For a subject character (a), the renderer computes the object's silhouette edges, shown here highlighted in white, with respect to the light (b). The renderer draws quadrilaterals (triangle pairs) starting at each silhouette edge, extruded away from the light source, and updates the stencil buffer (c). This process yields the rendered scene with shadows (d).

Until recently, in interactive systems, GPUs represented final colors with fractions between 0.0 and 1.0, at 8-bit precision. The GPU's fragment processor also clamped calculations to this limited range. Along with limited precision, this resulted in cheaper hardware. The first evolutionary step was support for increased range and precision during calculation. Today, a GPU performs calculations and stores integer and floating-point results at up to 32-bit precision.

Three-phase rendering

A typical cinematic renderer divides the work for each frame into three phases: prerendering, main rendering, and postprocessing. First, the renderer computes the data it will need for the main rendering phase. These data consist of shadow maps or shadow volumes for each light source, along with environment maps. In the main phase, the renderer draws the scene from the camera's viewpoint; this is what we usually think of as computer graphics rendering. For every light source, the renderer accumulates light energy contributions to each pixel. In the postprocessing phase, the renderer uses image processing, for example, to simulate lens flare and map HDR color values to the display device's limited gamut.

Shadows increase geometric complexity and provide important visual cues. In games, they set a mood—for example, creating fear when you see an enemy's shadow in a corridor. Shadows have long been one of the most challenging problems in interactive computer graphics. A renderer can handle most image synthesis through local calculations, but rendering shadows correctly requires considering whether each triangle obscures light to any other triangle.

One pertinent and well-known algorithm is stencil shadow volumes.[2,3] First, the renderer creates the depth buffer for the scene, writing only depth values. During the second stage, the renderer makes a preprocessing pass for each light source. It draws triangles (but not color or depth) into the stencil buffers, counting entry and exit to compute the regions of

space in which some object casts a shadow or obscures the light source. Specifically, front-facing triangles increment the stencil value at a pixel, and back-facing triangles decrement this value. After the renderer has drawn all the triangles, the stencil value at each sample indicates whether the light source illuminates that sample or whether it is in shadow. A nonzero stencil value indicates shadow.

Figure 4 illustrates shadow volume generation. The silhouette quadrilaterals combined with the facets of the model facing away from the light define the shadow volumes. Because the renderer must compute the shadow volume separately for each light, the number of shadow triangles drawn can be very large. Although the pixels are computationally simple during stencil shadow-volume generation, it's not uncommon for the shadow volume prepass to consume about two-thirds of total rendering time.

Architectural drivers

The GeForce 6800 architecture has three major drivers.

- *Programmability.* Programmable elements, evolved from configurable logic, afford much greater algorithmic flexibility. Programmability also lets content developers add value with their proprietary algorithms.
- *Parallelism.* The rendering problem has a great deal of data parallelism. The scenes comprise objects defined by vertices, which the GPU can process independently. The renderer expresses the result of its calculations as millions of independent pixels. These high levels of parallelism permit the efficient deployment of broadly and deeply parallel computational resources.
- *Memory.* The memory subsystem is the most precious resource in any graphics system, and its characteristics heavily influence the GPU's design. Designers must fit the GPU architecture to the memory subsystem's bandwidth and latency characteristics.

Programmability

The GeForce 6800's programming model enables parallelized acceleration. There are two separate programs: The application executes a vertex program independently on every vertex; similarly, the GPU applies a fragment program independently to every pixel fragment. For every vertex received in the command stream, the GPU launches a thread executing the vertex program. For every rasterized pixel fragment, the machine dispatches one thread of the fragment program. Each thread has its own unique inputs available in read-only registers. Supporting hardware loads these inputs before thread launch. Each thread also has write-only output registers, whose content the machine forwards to the next processing stage. In addition to these inputs and outputs, each thread has private temporary registers, read-only program parameters, and access to filtered and resampled texture map images.

Nvidia introduced the first programmable GPU, the GeForce3, in 2001. The GeForce3 supported a programmable vertex processor.[4] In 2002, the original GeForce FX series introduced programmable vertex and fragment processors. Now, the GeForce 6800 has unified these capabilities and made them orthogonal. The fragment processor supports dynamic flow control, as the vertex processor did in the GeForce FX. In addition, the vertex program can access the texture subsystem, previously available only to fragment programs. The FX had introduced floating-point textures and frame buffers; the GeForce 6800 adds the ability to blend and filter in floating-point. Finally, from a language and API perspective, the GeForce 6800 supports both Direct3D and OpenGL with just-in-time compiled machine-independent assembler as well as higher-level C-like programming languages.

Parallelism

Contrasting CPUs and GPUs makes it easier to understand the motivation behind the GPU architecture. The GPU workload offers more independent calculations than a typical CPU workload; the programmer's view is single threaded, while the machine is actually deeply multithreaded. The GPU can afford larger amounts of floating-point computational power because the control overhead per operation is lower than that for a CPU, and a GPU can effectively execute extensive floating-point computations. The simple programming

model and large amount of independent calculation result in deep and wide parallelism for the GeForce 6800 to exploit.

Another interesting difference between CPUs and GPUs is the use of dedicated mode-controlled functional units for specialized performance-critical tasks. In addition to the programmable vertex and fragment processors, there are specialized units for data fetch, rasterization (conversion from triangles to pixel fragments), and texture filtering. We determined the processor instruction set by analyzing the graphics workload. For example, because of their importance to graphics algorithms, the GeForce 6800 includes fast and accurate transcendental functions and inner-product instructions.

Memory

The memory bandwidth demands of GPU systems have always been insatiable, largely because there are so many concurrently active threads. CPUs have dealt with memory limitations by using ever-larger caches, but graphics working-set sizes have grown at least as fast as transistor density, and it remains prohibitive to implement an on-chip cache large enough to achieve 99 percent hit rates. Caches as part of the memory hierarchy cannot affordably support long-term reuse. Therefore, our GPU cache designs assume a 90 percent hit rate with many misses in flight. Stated another way, we implement caches that support effective streaming with local reuse of fetched data.

Because of bandwidth limitations, we aim for 100 percent memory bandwidth utilization, which forces the internal processors and fixed-function units to be latency tolerant and to respect page locality. We also schedule DRAM cycles to minimize idle data-bus time caused by read-write direction changes. GPUs improve page locality by mapping two- and three-dimensional spatial locality to corresponding locality at the granularity of a one-dimensional DRAM page.

The GeForce 6800 memory subsystem comprises four independent 64-pin partition controllers. Because of fluctuations in DRAM supply, it's important that the GeForce 6800 maintain plenty of flexibility with respect to the specific memory used. The memory controller supports double-data-rate (DDR2) and its graphics-oriented counterpart GDDR3 signaling and protocols at various clock frequencies with widely programmable memory cycle timings. The memory controller also maps linear addresses to pages and individual partitions. For efficiency, the controllers arbitrate among a dozen sources of read and write traffic, and they balance bus utilization with latency. To further increase effective bandwidth, the controller uses lossless compression and decompression, which is transparent to clients.

GeForce 6800 statistics

The GeForce 6800 has high-throughput programmable floating-point processors, efficient special-purpose engines, and a flexible memory subsystem that supports a wide range of DRAM types, from the commodity to the exotic. Its notable statistics include

- 222 million transistors,
- 303-mm^2 area,
- 550-MHz double-data-rate memory clock,
- 400+ MHz core clock,
- 400 million vertices per second, and
- 120+ Gflops peak (equal to six 5-GHz Pentium 4 processors).

Performance regimes

The GPU application space is extremely multimodal: No single performance mode characterizes any given application. For example, stencil shadow volumes can consume two-thirds of a frame's rendering time without writing any color or depth values. Different applications, and different millisecond time slices within a single application, have different characteristics. In designing for these regimes, we sought optimal use of the most expensive resource, sizing key memory clients to saturate all available memory bandwidth. Dozens of rendering regimes require "speed of light" performance limited only by memory bandwidth.

We already mentioned stencil shadow-volume rendering as a specialized non–color-updating phase of rendering. This rendering step has two highly specialized modes of operation: one mode renders only depth values, and the other updates only the stencil value. Because we designed the GPU to saturate DRAM bandwidth at 16 pixels per clock cycle when the renderer is updating both color *and* depth, the processor must deliver an even higher pixel rate to saturate memory when performing only depth or stencil work.

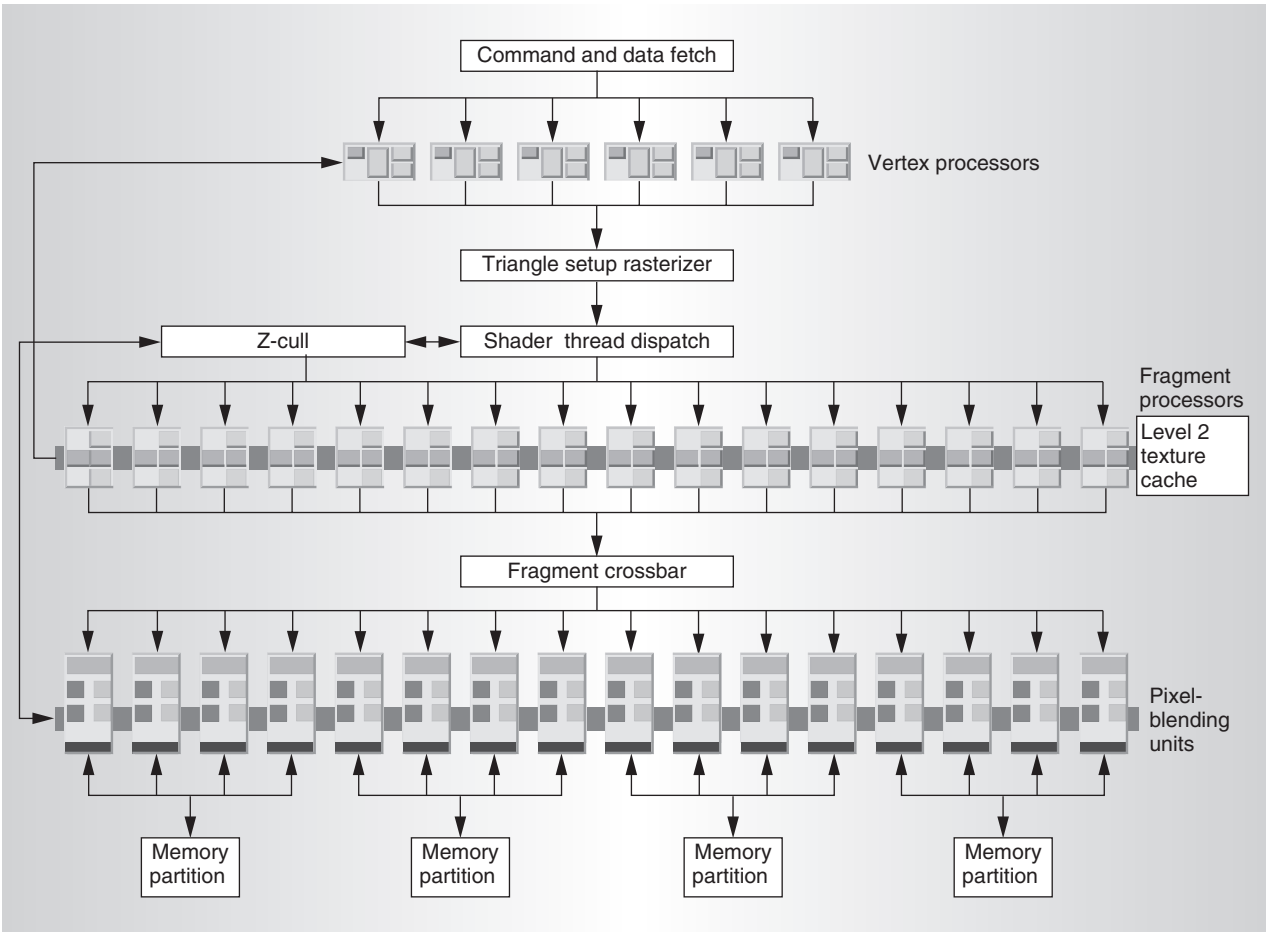

Figure 5. GeForce 6800 block diagram.

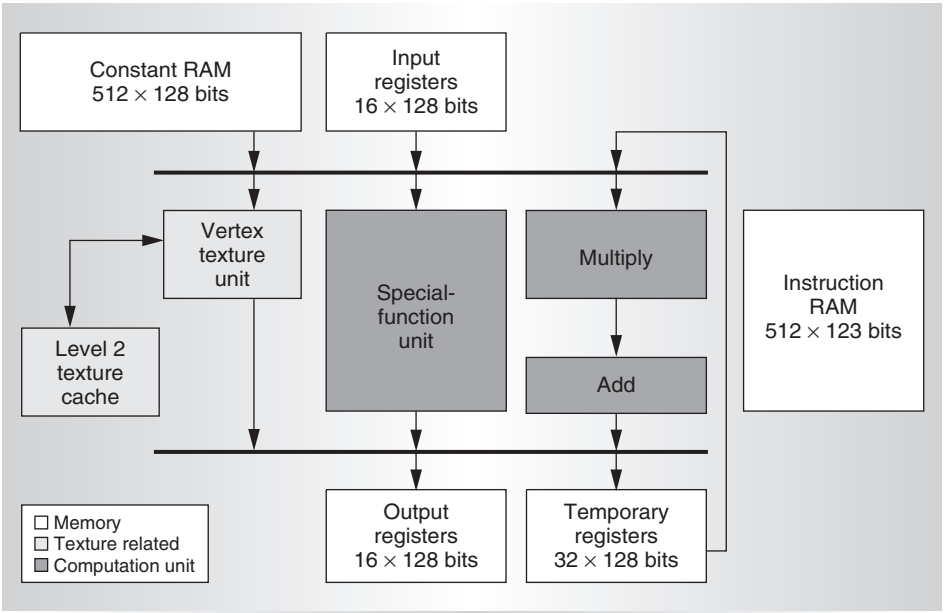

Figure 6. Vertex processor block diagram.

A tour of the GeForce 6800

Figure 5 is a top-level diagram of the GeForce 6800. Work flows from top to bottom, starting with the six identical programmable vertex processors. Because all vertices are independent of each other, the data fetcher assigns incoming work to any idle processor, and the parallel utilization is nearly perfect. The "GeForce 6800 statistics" sidebar provides more specifics.

Results from the vertex stage are reassembled in the original application-specified order to feed the triangle setup and rasterization units. For each primitive, the ras-

terizer identifies constituent pixel fragments and sends them to a fragment processor. Sixteen programmable fragment processors operate on the workload in parallel. Each thread receives the (*x*, *y*) addresses and interpolated inputs from the rasterizer. Because fragments are independent of one another, the processors approach 100 percent utilization.

Finally, a crossbar distributes color and depth results from the fragment processors to 16 fixed-function pixel-blending units, which perform frame buffer operations such as color blending, antialiasing, and stencil test and update. It's possible to feed the result from any fragment processor to any frame buffer location.

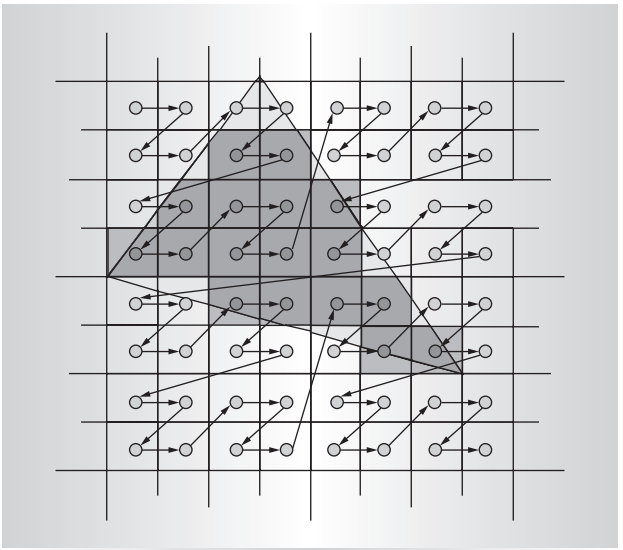

Figure 7. Page-friendly rasterization.

Vertex processor

The vertex processor executes very large instruction words. The instruction load unit forms a 123-bit internal instruction from either of two driver-visible instruction set architectures (ISAs); Nvidia supports two ISA generations to aid in streamlining initial product and driver development. As Figure 5 shows, there are six vector floating-point processors. Each processor's data path comprises a vector multiply-add unit, a scalar special-function unit, and a texture unit, as shown in Figure 6. The vector unit can perform four IEEE single-precision multiply, add, or multiply-add operations, as well as inner products, max, min, and so on. The special-function unit performs transcendental operations such as sine, cosine, log, and exponential to within one unit in the last place (ULP) of IEEE single-precision accuracy for operands in the nominal range.

The computational units fetch operands from a 512 × 128-bit constant RAM, from temporary registers up to 32 × 128 bits, and from 16 × 128-bit input registers. The processor feeds computed results back into the temporary registers or out to one of the 16 × 128-bit output registers. The vertex processor reads instructions from a 512-entry instruction RAM.

To preserve a simple implementation-independent programming model, the vertex processor uses threads to make the data path appear to have unity latency, and it uses scoreboarding to hide texture fetch latency. The implementation is fully multiple instruction, multiple data; therefore, data-dependent branches are free of the penalty normally accompanying single-instruction, multiple-data implementations. Finally, the processor can issue instructions to both vector and scalar data paths at every clock cycle.

Primitive setup and rasterizer

The APIs define the various activities occurring between the vertex and fragment stages with unique precision requirements. Therefore, these activities don't require programmability and are implemented efficiently in fixed-function units.

The primitive assembly unit assembles primitives such as lines or triangles from transformed vertices. Vertex positions arrive as 4-vectors of homogeneous coordinates, the standard method for handling perspective foreshortening.[5] Although we divide through by the fourth component, we check to see whether the assembled primitive is outside the view frustum. If so, the primitive is culled; otherwise, after perspective division, we apply the viewport scale and offset to obtain screen-space *x*, *y*, and *z* (depth). Next, the setup unit computes coefficients describing the primitive's edges. Finally, the rasterizer converts the primitive into pixel fragments for input to the array of fragment processors. The rasterizer traverses the primitive in a DRAM-page-friendly order like that shown in Figure 7.

Fragment processor

The GPU forwards attributes, specified at

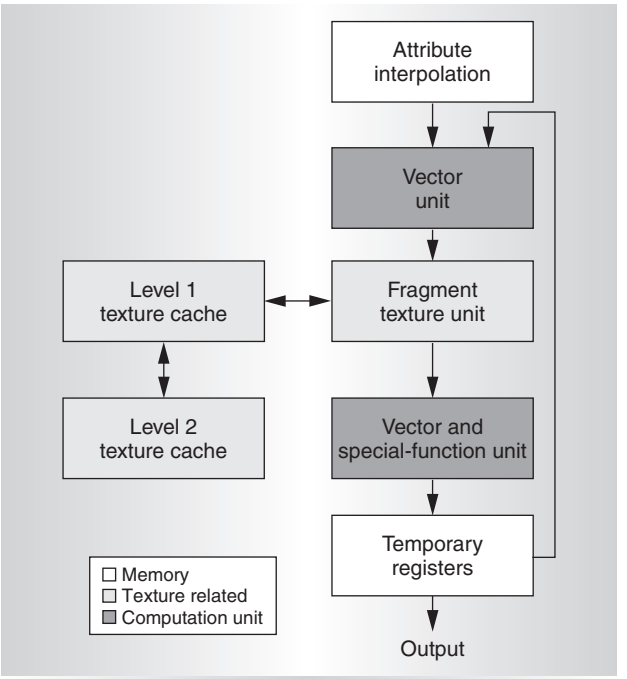

Figure 8. Fragment processor block diagram.

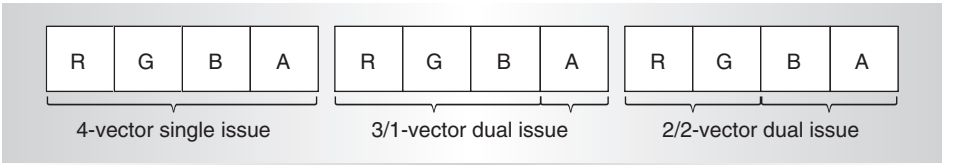

Figure 9. Vector issue options.

the triangle's vertices, from the vertex processor to the fragment processor. The fragment processor smoothly interpolates these attributes across the triangle's face. Using these interpolated input attributes, a fragment program computes output colors, using math and texture lookup instructions. The GeForce 6800 fragment processor can perform operations with 16- or 32-bit floating-point precision (FP16 and FP32). The inputs to the fragment processor are position, color, depth, fog, and 10 generic 4 × FP32 attributes. The processor sends its outputs to as many as four render target buffers. Like the vertex processor, the fragment processor is general purpose, and it has constants, temporary register resources, and branching capabilities similar to those of the vertex processor.

Fragment processor detail

As Figure 8 shows, each of the 16 fragment processors includes an interpolation block for input attributes, two vector math units, a special-function/normalize unit, and a texture unit. Both computation blocks can perform 4-vector floating-point operations. The lower block can do a multiply-add operation. Combined, the two blocks can sustain 12 floating-point operations per pixel per clock cycle. The lower block also supports the same transcendental functions supported in the vertex processor's special-function unit. To hide the latency of texture lookups that fetch from external memory, each fragment processor maintains state for hundreds of in-flight threads.

Superscalar instruction issue

Microsoft's DirectX 9 graphics API supports a vector-oriented instruction set. The assembler has instructions that perform most operations on 4-vectors of FP32 data. However, many fragment processing algorithms treat alpha, the transparency component, separately from the three color components. As a result, the assembler has provisions to indicate a pairing of instructions—that is, an instruction operating on a 3-vector, usually RGB, paired with an instruction operating on a scalar, usually alpha. This mechanism permits dual issue of source-level instructions. The GeForce 6800's fragment processor supports fully general 4-vector split operations—4-vector, 3/1-vector, and 2/2-vector operations—as Figure 9 illustrates.

The two computation stages can exploit this dual issue of instructions to perform two distinct operations on different subsets of the 4-vector. Together with texture and special functions, each fragment processor can execute up to six DirectX 9 instructions per pixel per clock cycle. Figure 10 is an example of six-issue code.

Texture unit

The literature provides a good overview of texture mapping.[6-8] A texture map is an array of data in one, two, or three dimensions. The simplest uses of texture in rendering involve mapping a decal image onto some object built from a collection of geometric primitives. Figure 11 provides an example. Because each

pixel maps to a region of the image, filtering is necessary to eliminate image frequency content above the sampling rate implied by the pixel footprint in texture space. Instead of the fragment footprint, which includes a coverage mask, the texture unit uses the fully covered pixel footprint to determine filtering.

With arbitrary programs, a texture is more generally a way to express a function of one, two, or three variables as a table. We can think of the function value as a color 4-tuple (red, green, blue, and alpha) or more generally as an *n*-tuple of arbitrary values. As with simple image mapping, the fixed-function texture unit's job is to return a properly sampled result, given the input address vector. Proper sampling is a weighted average of a collection of samples near the ideal sample location, with minimal aliasing, and it shouldn't introduce too much blurring.

The texture unit operates with a deeply pipelined cache. Typically, the cache has many hits and misses in flight. To reduce memory traffic, the application can use compressed-texture formats. To facilitate fine-grained access and random addressability, these formats use small-grained fixed-ratio schemes, with a fixed compression ratio of 4:1. Because the ratio is fixed, it is also a lossy scheme.

The texture subsystem must filter results before returning them to the requesting fragment processor. The GeForce 6800 supports four types of filtering: point-, bilinear-, and trilinear-sampled, and anisotropic. A point-sampled request simply returns the *texel* (texture element, or pixel) nearest to the address the requester provided. When performing bilinear-sampled filtering, the texture unit takes the weighted average of four texels. Trilinear-sampled filtering uses prefiltered versions of the texture, which form a hierarchy, or stack, of textures called a *mip-map*,[9] illustrated in Figure 12. In trilinear-sampled mode, the filtering operation blends eight texels—that is, the operation linearly blends two bilinearly filtered levels.

In Figure 11, a circle in screen space (Figure 11b) maps to an ellipse in texture space (Figure 11a). This means the texels needed to obtain one pixel's color value occupy an elliptical footprint in texture memory. The degree of anisotropy is the ratio of the ellipse's major and minor axes. Larger anisotropy ratios require

```
ps_2_0
def c1, 2.0, -1.0, 0.0, 0.0
dcl t0.rg
dcl t1
dcl t4.rgb
dcl v0
dcl_2d s0
dcl_2d s1
dcl_cube s2
dcl_2d s3

# clock 1
texld r0, t0, s0;       # tex fetch
madr r0,r0,c1.r,c1.g    # _bx2 in tex
nrm r1.rgb, t4          # nrm in shdr0
dp3 r1.r,r1,r0          # 3D dot in shdr1
mul r0.a,r0,r0          # dual issue in shdr1


# clock 2
mul r1.a,r0.a,c2.a      # dual issue in shdr0
mul r0.rgb,r1.r,r0      # dual issue in shdr0
add r0.a,r1.r,r1.r      # fx2 in shdr0
mad r0.rg,r0.a,c1,c1.a  # mad in shdr1
mul r1.ba,r1.a,r0.a,c2  # dual issue in shdr1

# clock 3
rcp r0.a,r0.a           # recip in shdr0
mul r0.rgr0,r0.a        # div in shdr0
mul r0.a,r0.a,r1.a      # dual issue in shdr0
texld r2,r0, s1         # texture fetch
mad r2.rgb,r0.a,r2,c5   # mad in shdr1
abs r0.a,r0.a           # abs in shdr1
log r0.a,r0.a           # log in shdr1

<< etc >>

mov oC0, r0             # output color
```

Figure 10. Annotations in this DirectX 9 program code show how the compiler schedules instruction sequences for the GeForce 6800 fragment processor.

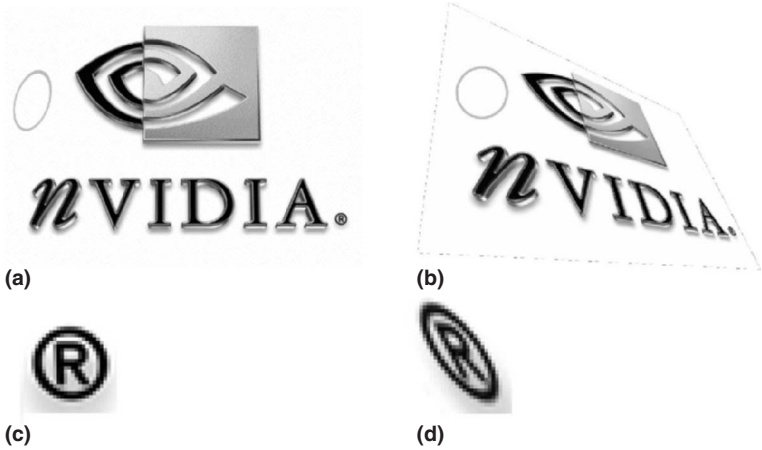

Figure 11. Texture and perspective view: texture with elliptical footprint (a), perspective image with circular footprint in screen space (b), texture close-up (c), and resampled image (d).

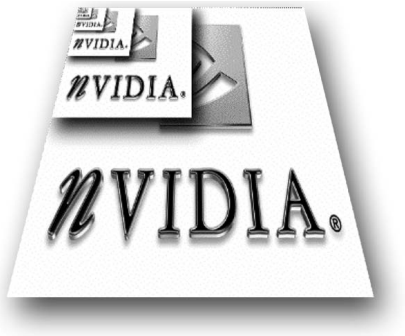

Figure 12. Mip-map hierarchy.

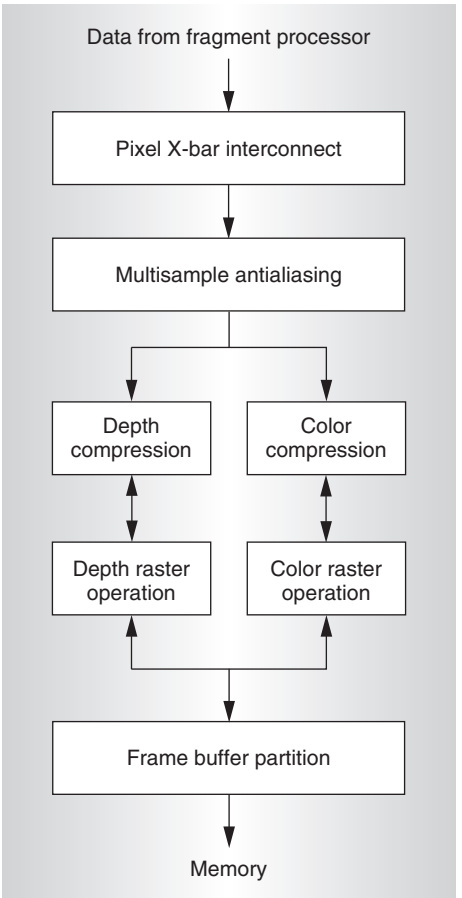

Figure 13. Pixel engine block diagram.

more texels to be read and evaluated when performing an anisotropic filtering operation. The GeForce 6800 supports up to 16:1 anisotropic filtering, and it processes texture lookup requests at four FP16 texels per clock cycle per texture unit.

Pixel engines

The GeForce 6800 contains 16 pixel engines. These fixed-function units perform depth and stencil test and update, as well as color blending, at 16 pixels per clock cycle. If no color destination is active, depth and stencil test can run at 32 pixels per clock cycle; fast depth and stencil update accelerates shadow volume rendering. Blending of 16-bit floating-point frame buffer values has proved to be one of the GeForce 6800's most important new features because it directly accelerates HDR rendering and light accumulation. The memory controller uses lossless color and depth compression to reduce bandwidth demands. Finally, the pixel engines support high-quality antialiasing (filtering).

Pixel pipeline detail. Each pixel engine connects to a specific memory partition (see Figure 5). The pixel engines expand the depth and color of each fragment into multiple samples when the renderer enables antialiasing. When possible, the engines losslessly compress depth and color, indicated by depth compression and color compression in Figure 13. The depth and color units then read and write to the local memory partition to carry out the depth and stencil, and color-blend operations.

Antialiasing. The GeForce 6800 supports various antialiasing options, which trade image quality for performance. The two primary algorithms are multisampling and supersampling. Both involve generating two, four, or eight samples for each displayed pixel, then taking a weighted average of all samples to produce the pixel's displayed color.

Multisampling executes the fragment program once per pixel fragment and reuses the resulting color value for all its samples. Supersampling reruns the fragment program to generate a unique color for every sample. In both cases, we evaluate the depth correctly and uniquely at each pixel subsample location. This frequency of evaluation is necessary to avoid image artifacts and to achieve smooth edges at silhouettes and object interpenetrations. Multisampling imposes a significantly smaller fragment processor load while antialiasing edges and interpenetrations. Supersampling multiplies the fragment processor load by the sample count to provide additional antialiasing of each fragment's resulting color.

The GeForce 6800, the flagship of an architectural line targeted at a large and diverse market, supports interactive rendering approaching cinematic quality. The architecture is tailored to its highly parallel task and can also scale down to low-power, low-cost devices. The GeForce 6800 is one of the most complex logic designs shipping in high volume today. MICRO

References

1. J. Cohen et al., "Real-time High Dynamic Range Texture Mapping," *Proc. 12th Eurographics Rendering Workshop*, European Assoc. for Computer Graphics, 2001, pp. 313-320.
2. F. Crow, "Shadow Algorithms for Computer Graphics," *Proc. 24th Ann. Conf. Computer Graphics and Interactive Techniques* (Siggraph 77), ACM Press, 1977, pp. 242-248.
3. C. Everett and M.J. Kilgard, "Practical and Robust Shadow Volumes for Hardware-Accelerated Rendering," Mar. 2002; http://developer.nvidia.com/object/robust_shadow_volumes.html.
4. E. Lindholm, M. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine," *Proc.*

28th Ann. Conf. Computer Graphics and Interactive Techniques (Siggraph 01), 2001, ACM Press, pp. 149-158.
5. J.D. Foley et al., *Computer Graphics: Principles and Practice*, 2nd ed., Addison-Wesley, 1990.
6. P.S. Heckbert and H.P. Moreton, "Interpolation for Polygon Texture Mapping and Shading," *State of the Art in Computer Graphics: Visualization and Modeling*, Springer-Verlag, 1991, pp. 101-111.
7. P.S. Heckbert, "Survey of Texture Mapping," *IEEE Computer Graphics and Applications*, vol. 6, no. 6, Nov. 1986, pp. 56-67.
8. T. Huettner and W. Strasser, "Fast Footprint MIPmapping," *Proc. Eurographics/Siggraph Workshop Graphics Hardware*, ACM Press, 1999, pp. 35-44.
9. L. Williams, "Pyramidal Parametrics," *Proc. 10th Ann. Conf. Computer Graphics and Interactive Techniques* (Siggraph 83), ACM Press, 1983, pp. 1-11.

John Montrym is the chief architect at Nvidia, where he has influenced the development of the architecture, hardware design, and design methodologies of 12 GPU products. He has a BS in electrical engineering from the Massachusetts Institute of Technology.

Henry Moreton is a member of the architecture group at Nvidia. His research interests include GPU programming models and architecture. Moreton has a PhD in computer science from the University of California, Berkeley.

Direct questions and comments about this article to John Montrym or Henry Moreton at Nvidia, 2701 San Tomas Expressway, Santa Clara, CA 95050; montrym@nvidia.com or moreton@nvidia.com.