Chapter 30

The GeForce 6 Series GPU Architecture

Emmett Kilgariff
NVIDIA Corporation

Randima Fernando
NVIDIA Corporation

The previous chapter described how GPU architecture has changed as a result of computational and communications trends in microprocessing. This chapter describes the architecture of the GeForce 6 Series GPUs from NVIDIA, which owe their formidable computational power to their ability to take advantage of these trends. Most notably, we focus on the GeForce 6800 (NVIDIA's flagship GPU at the time of writing, shown in Figure 30-1), which delivers hundreds of gigaflops of single-precision floating-point computation, as compared to approximately 12 gigaflops for current high-end CPUs. In this chapter—and throughout the book—references to GeForce 6 Series GPUs should be read to include the latest Quadro FX GPUs supporting Shader Model 3.0, which provide a superset of the functionality offered by the GeForce 6 Series. We start with a general overview of where the GPU fits into the overall computer system, and then we describe the architecture along with details of specific features and performance characteristics.

30.1 How the GPU Fits into the Overall Computer System

The CPU in a modern computer system communicates with the GPU through a graphics connector such as a PCI Express or AGP slot on the motherboard. Because the graphics connector is responsible for transferring all command, texture, and vertex data from the CPU to the GPU, the bus technology has evolved alongside GPUs over the past few years. The original AGP slot ran at 66 MHz and was 32 bits wide, giving a transfer rate of 264 MB/sec. AGP 2×, 4×, and 8× followed, each doubling the available

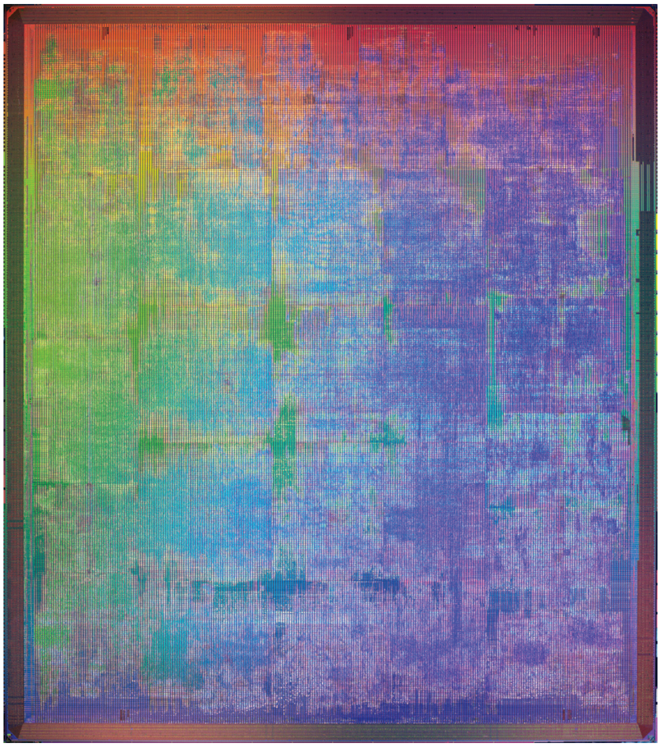


Figure 30-1. The GeForce 6800 Microprocessor

bandwidth, until finally the PCI Express standard was introduced in 2004, with a maximum theoretical bandwidth of 4 GB/sec simultaneously available to and from the GPU. (Your mileage may vary; currently available motherboard chipsets fall somewhat below this limit—around 3.2 GB/sec or less.)

It is important to note the vast differences between the GPU's memory interface bandwidth and bandwidth in other parts of the system, as shown in Table 30-1.

Table 30-1. Available Memory Bandwidth in Different Parts of the Computer System

Component	Bandwidth
GPU Memory Interface	35 GB/sec
PCI Express Bus (×16)	8 GB/sec
CPU Memory Interface (800 MHz Front-Side Bus)	6.4 GB/sec

Table 30-1 reiterates some of the points made in the preceding chapter: there is a vast amount of bandwidth available internally on the GPU. Algorithms that run on the GPU can therefore take advantage of this bandwidth to achieve dramatic performance improvements.

30.2 Overall System Architecture

The next two subsections go into detail about the architecture of the GeForce 6 Series GPUs. Section 30.2.1 describes the architecture in terms of its graphics capabilities. Section 30.2.2 describes the architecture with respect to the general computational capabilities that it provides. See Figure 30-2 for an illustration of the system architecture.

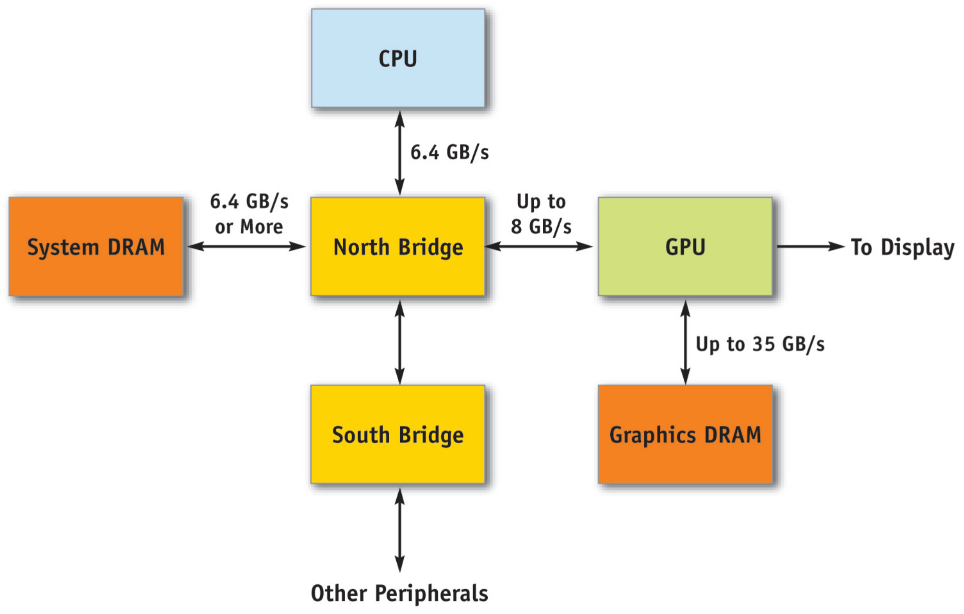


Figure 30-2. The Overall System Architecture of a PC

30.2.1 Functional Block Diagram for Graphics Operations

Figure 30-3 illustrates the major blocks in the GeForce 6 Series architecture. In this section, we take a trip through the graphics pipeline, starting with input arriving from the CPU and finishing with pixels being drawn to the frame buffer.

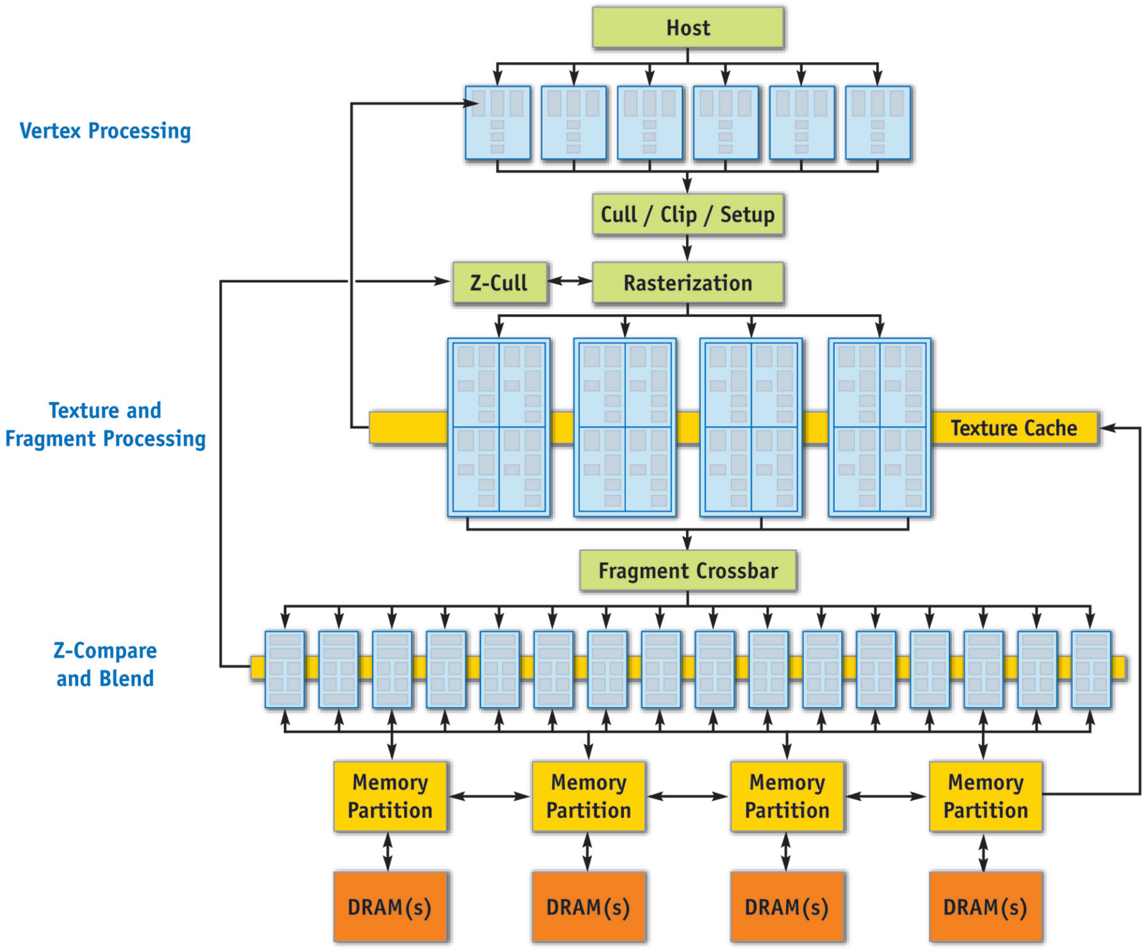


Figure 30-3. A Block Diagram of the GeForce 6 Series Architecture

First, commands, textures, and vertex data are received from the host CPU through shared buffers in system memory or local frame-buffer memory. A command stream is written by the CPU, which initializes and modifies state, sends rendering commands, and references the texture and vertex data. Commands are parsed, and a vertex fetch unit is used to read the vertices referenced by the rendering commands. The commands, vertices, and state changes flow downstream, where they are used by subsequent pipeline stages.

The vertex processors (sometimes called "vertex shaders"), shown in Figure 30-4, allow for a program to be applied to each vertex in the object, performing transformations, skinning, and any other per-vertex operation the user specifies. For the first time, a

GPU—the GeForce 6 Series—allows vertex programs to fetch texture data. All operations are done in 32-bit floating-point (fp32) precision per component. The GeForce 6 Series architecture supports scalable vertex-processing horsepower, allowing the same architecture to service multiple price/performance points. In other words, high-end models may have six vertex units, while low-end models may have two.

Because vertex processors can perform texture accesses, the vertex engines are connected to the texture cache, which is shared with the fragment processors. In addition, there is a vertex cache that stores vertex data both before and after the vertex processor, reducing fetch and computation requirements. This means that if a vertex index occurs twice in a draw call (for example, in a triangle strip), the entire vertex program doesn't have to be rerun for the second instance of the vertex—the cached result is used instead.

Vertices are then grouped into primitives, which are points, lines, or triangles. The Cull/Clip/Setup blocks perform per-primitive operations, removing primitives that aren't visible at all, clipping primitives that intersect the view frustum, and performing edge and plane equation setup on the data in preparation for rasterization.

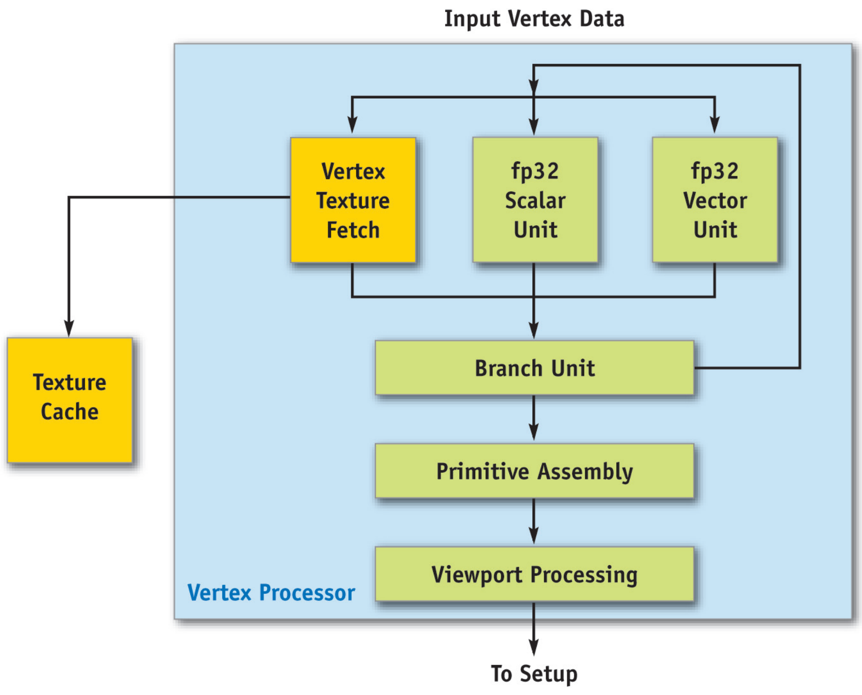


Figure 30-4. The GeForce 6 Series Vertex Processor

The rasterization block calculates which pixels (or samples, if multisampling is enabled) are covered by each primitive, and it uses the z-cull block to quickly discard pixels (or samples) that are occluded by objects with a nearer depth value. Think of a fragment as a "candidate pixel": that is, it will pass through the fragment processor and several tests, and if it gets through all of them, it will end up carrying depth and color information to a pixel on the frame buffer (or render target).

Figure 30-5 illustrates the fragment processor (sometimes called a "pixel shader") and texel pipeline. The texture and fragment-processing units operate in concert to apply a shader program to each fragment independently. The GeForce 6 Series architecture supports a scalable amount of fragment-processing horsepower. Another popular way to say this is that GPUs in the GeForce 6 Series can have a varying number of *fragment pipelines* (or "pixel pipelines"). Similar to the vertex processor, texture data is cached on-chip to reduce bandwidth requirements and improve performance.

The texture and fragment-processing unit operates on squares of four pixels (called *quads*) at a time, allowing for direct computation of derivatives for calculating texture level of detail. Furthermore, the fragment processor works on groups of hundreds of pixels at a time in single-instruction, multiple-data (SIMD) fashion (with each fragment processor engine working on one fragment concurrently), hiding the latency of texture fetch from the computational performance of the fragment processor.

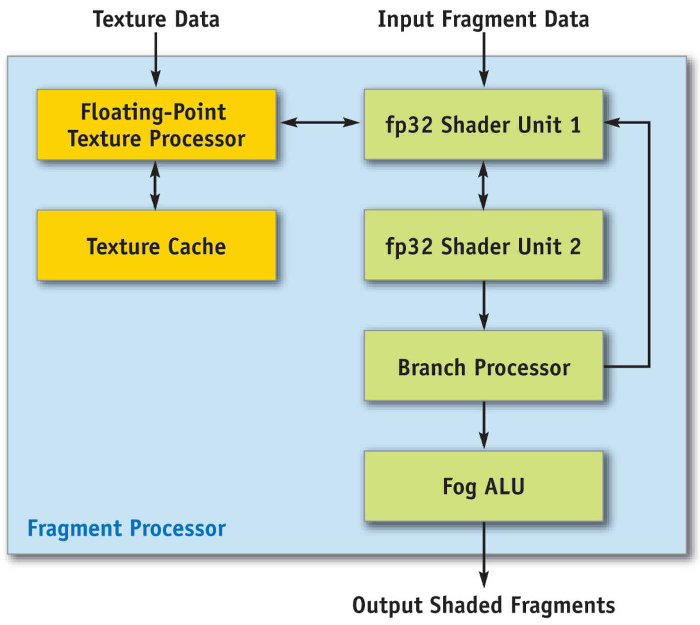


Figure 30-5. The GeForce 6 Series Fragment Processor and Texel Pipeline

The fragment processor uses the texture unit to fetch data from memory, optionally filtering the data before returning it to the fragment processor. The texture unit supports many source data formats (see Section 30.3.3, "Supported Data Storage Formats"). Data can be filtered using bilinear, trilinear, or anisotropic filtering. All data is returned to the fragment processor in fp32 or fp16 format. A texture can be viewed as a 2D or 3D array of data that can be read by the texture unit at arbitrary locations and filtered to reconstruct a continuous function. The GeForce 6 Series supports filtering of fp16 textures in hardware.

The fragment processor has two fp32 shader units per pipeline, and fragments are routed through both shader units and the branch processor before recirculating through the entire pipeline to execute the next series of instructions. This rerouting happens once for each core clock cycle. Furthermore, the first fp32 shader can be used for perspective correction of texture coordinates when needed (by dividing by *w*), or for general-purpose multiply operations. In general, it is possible to perform eight or more math operations in the pixel shader during each clock cycle, or four math operations if a texture fetch occurs in the first shader unit.

On the final pass through the pixel shader pipeline, the fog unit can be used to blend fog in fixed-point precision with no performance penalty. Fog blending happens often in conventional graphics applications and uses the following function:

```
out = FogColor * fogFraction + SrcColor * (1 - fogFraction)
```

This function can be made fast and small using fixed-precision math, but in general IEEE floating point, it requires two full multiply-adds to do effectively. Because fixed point is efficient and sufficient for fog, it exists in a separate small unit at the end of the shader. This is a good example of the trade-offs in providing flexible programmable hardware while still offering maximum performance for legacy applications.

Fragments leave the fragment-processing unit in the order that they are rasterized and are sent to the z-compare and blend units, which perform depth testing (z comparison and update), stencil operations, alpha blending, and the final color write to the target surface (an off-screen render target or the frame buffer).

The memory system is partitioned into up to four independent memory partitions, each with its own dynamic random-access memories (DRAMs). GPUs use standard DRAM modules rather than custom RAM technologies to take advantage of market economies and thereby reduce cost. Having smaller, independent memory partitions allows the memory subsystem to operate efficiently regardless of whether large or small blocks of data are transferred. All rendered surfaces are stored in the DRAMs, while textures and input data can be stored in the DRAMs or in system memory. The four independent memory partitions give the GPU a wide (256 bits), flexible memory subsystem, allowing for streaming of relatively small (32-byte) memory accesses at near the 35 GB/sec physical limit.

30.2.2 Functional Block Diagram for Non-Graphics Operations

As graphics hardware becomes more and more programmable, applications unrelated to the standard polygon pipeline (as described in the preceding section) are starting to present themselves as candidates for execution on GPUs.

Figure 30-6 shows a simplified view of the GeForce 6 Series architecture, when used as a graphics pipeline. It contains a programmable vertex engine, a programmable fragment engine, a texture load/filter engine, and a depth-compare/blending data write engine.

In this alternative view, a GPU can be seen as a large amount of programmable floating-point horsepower and memory bandwidth that can be exploited for compute-intensive applications completely unrelated to computer graphics.

Figure 30-7 shows another way to view the GeForce 6 Series architecture. When used for non-graphics applications, it can be viewed as two programmable blocks that run serially: the vertex processor and the fragment processor, both with support for fp32 operands and intermediate values. Both use the texture unit as a random-access data fetch unit and access data at a phenomenal 35 GB/sec (550 MHz DDR memory clock × 256 bits per clock cycle × 2 transfers per clock cycle). In addition, both the vertex and the fragment processor are highly computationally capable. (Performance details follow in Section 30.4.)

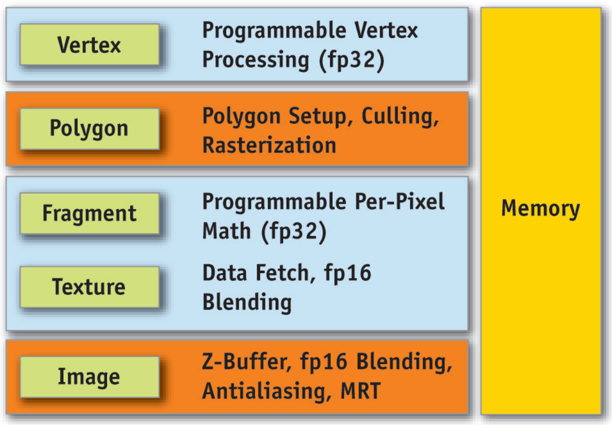
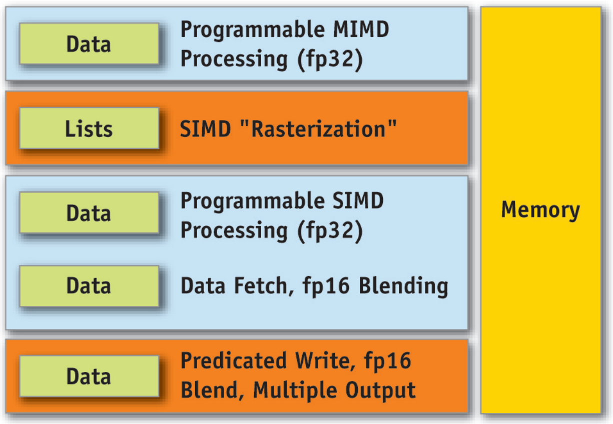

Vertex
Programmable Vertex Processing (fp32)
Polygon
Polygon Setup, Culling, Rasterization
Fragment
Programmable Per-Pixel Math (fp32)
Texture
Data Fetch, fp16 Blending
Image
Z-Buffer, fp16 Blending, Antialiasing, MRT
Memory

Figure 30-6. The GeForce 6 Series Architecture Viewed as a Graphics Pipeline



Data
Programmable MIMD Processing (fp32)
Lists
SIMD "Rasterization"
Data
Programmable SIMD Processing (fp32)
Data
Data Fetch, fp16 Blending
Data
Predicated Write, fp16 Blend, Multiple Output
Memory

Figure 30-7. The GeForce 6 Series Architecture for Non-Graphics Applications

The vertex processor operates on data, passing it directly to the fragment processor, or by using the rasterizer to expand the data into interpolated values. At this point, each triangle (or point) from the vertex processor has become one or more *fragments*.

Before a fragment reaches the fragment processor, the z-cull unit compares the pixel's depth with the values that already exist in the depth buffer. If the pixel's depth is greater, the pixel will not be visible, and there is no point shading that fragment, so the fragment processor isn't even executed. (This optimization happens only if it's clear that the fragment processor isn't going to modify the fragment's depth.) Thinking in a general-purpose sense, this *early culling* feature makes it possible to quickly decide to skip work on specific fragments based on a scalar test. Chapter 34 of this book, "GPU Flow-Control Idioms," explains how to take advantage of this feature to efficiently predicate work for general-purpose computations.

After the fragment processor runs on a potential pixel (still a "fragment" because it has not yet reached the frame buffer), the fragment must pass a number of tests in order to move farther down the pipeline. (There may also be more than one fragment that comes out of the fragment processor if multiple render targets [MRTs] are being used. Up to four MRTs can be used to write out large amounts of data—up to 16 scalar floating-point values at a time, for example—plus depth.)

First, the scissor test rejects the fragment if it lies outside a specified subrectangle of the frame buffer. Although the popular graphics APIs define scissoring at this location in the pipeline, it is more efficient to perform the scissor test in the rasterizer. Scissoring in x and y actually happens in the rasterizer, before fragment processing, and z scissoring happens during z-cull. This avoids all fragment processor work on scissored (rejected) pixels. Scissoring is rarely useful for general-purpose computation because general-purpose programmers typically draw rectangles to perform computations in the first place.

Next, the fragment's depth is compared with the depth in the frame buffer. If the depth test passes, the fragment moves on in the pipeline. Optionally, the depth value in the frame buffer can be replaced at this stage.

After this, the fragment can optionally test and modify what is known as the stencil buffer, which stores an integer value per pixel. The stencil buffer was originally intended to allow programmers to mask off certain pixels (for example, to restrict drawing to a cockpit's windshield), but it has found other uses as a way to count values by incrementing or decrementing the existing value. This feature is used for stencil shadow volumes, for example.

If the fragment passes the depth and stencil tests, it can then optionally modify the contents of the frame buffer using the blend function. A blend function can be described as

```
out = src * srcOp + dst * dstOp
```

where `source` is the fragment color flowing down the pipeline; `dst` is the color value in the frame buffer; and `srcOp` and `dstOp` can be specified to be constants, source color components, or destination color components. Full blend functionality is supported for all pixel formats up to fp16×4. However, fp32 frame buffers don't support blending—only updating the buffer is allowed.

Finally, a feature called *occlusion query* makes it possible to quickly determine if any of the fragments that would be rendered in a particular computation would cause results to be written to the frame buffer. (Recall that fragments that do not pass the z-test don't have any effect on the values in the frame buffer.) Traditionally, the occlusion query test is used to allow graphics applications to avoid making draw calls for occluded objects, but it is useful for GPGPU applications as well. For instance, if the depth test is used to determine which outputs need to be updated in a sparse array, updating depth can be used to indicate when a given output has converged and no further work is needed. In this case, occlusion query can be used to tell when all output calculations are done. See Chapter 34 of this book, "GPU Flow-Control Idioms," for further information about this idea.

30.3 GPU Features

This section covers both fixed-function features and Shader Model 3.0 support (described in detail later) in GeForce 6 Series GPUs. As we describe the various pieces, we focus on the many new features that are meant to make applications shine (in terms of both visual quality and performance) on GeForce 6 Series GPUs.

30.3.1 Fixed-Function Features

Geometry Instancing

With Shader Model 3.0, the capability for sending multiple batches of geometry with one Direct3D call has been added, greatly reducing driver overhead in these cases. The hardware feature that enables instancing is *vertex stream frequency*—the ability to read vertex attributes at a frequency less than once every output vertex, or to loop over a subset of vertices multiple times. Instancing is most useful when the same object is drawn multiple times with different positions, for example, when rendering an army of soldiers or a field of grass.

Early Culling/Clipping

GeForce 6 Series GPUs are able to cull nonvisible primitives before shading at a high rate and clip partially visible primitives at full speed. Previous NVIDIA products would cull nonvisible primitives at primitive-setup rates, and clip all partially visible primitives at full speed.

Rasterization

Like previous NVIDIA products, GeForce 6 Series GPUs are capable of rendering the following objects:

- Point sprites
- Aliased and antialiased lines
- Aliased and antialiased triangles

Multisample antialiasing is also supported, allowing accurate antialiased polygon rendering. Multisample antialiasing supports all rasterization primitives. Multisampling is supported in previous NVIDIA products, though the 4× multisample pattern was improved for GeForce 6 Series GPUs.

Z-Cull

NVIDIA GPUs since GeForce3 have technology, called *z-cull*, that allows hidden surface removal at speeds much faster than conventional rendering. The GeForce 6 Series z-cull unit is the third generation of this technology, which has increased efficiency for a wider range of cases. Also, in cases where stencil is not being updated, early stencil reject can be employed to remove rendering early when stencil test (based on equals comparison) fails.

Occlusion Query

Occlusion query is the ability to collect statistics on how many fragments passed or failed the depth test and to report the result back to the host CPU. Occlusion query can be used either while rendering objects or with color and z-write masks turned off, returning depth test status for the objects that would have been rendered, without modifying the contents of the frame buffer. This feature has been available since the GeForce3 was introduced.

Texturing

Like previous GPUs, GeForce 6 Series GPUs support bilinear, trilinear, and anisotropic filtering on 2D and cube-map textures of various formats. Three-dimensional textures support bilinear, trilinear, and quad-linear filtering, with and without mipmapping. Here are the new texturing features on GeForce 6 Series GPUs:

- Support for all texture types (2D, cube map, 3D) with fp16×2, fp16×4, fp32×1, fp32×2, and fp32×4 formats
- Support for all filtering modes on fp16×2 and fp16×4 texture formats
- Extended support for non-power-of-two textures to match support for power-of-two textures, specifically:
  - Mipmapping
  - Wrapping and clamping
  - Cube map and 3D textures

Shadow Buffer Support

NVIDIA GPUs support shadow buffering directly. The application first renders the scene from the light source into a separate z-buffer. Then during the lighting phase, it fetches the shadow buffer as a projective texture and performs z-compares of the shadow buffer data against a value corresponding to the distance from the light. If the distance passes the test, it's in light; if not, it's in shadow. NVIDIA GPUs have dedicated transistors to perform four z-compares per pixel (on four neighboring z-values) per clock, and to perform bilinear filtering of the pass/fail data. This more advanced variation of percentage-closer filtering saves many shader instructions compared to GPUs that don't have direct shadow buffer support.

High-Dynamic-Range Blending Using fp16 Surfaces, Texture Filtering, and Blending

GeForce 6 Series GPUs allow for fp16×4 (four components, each represented by a 16-bit float) filtered textures in the pixel shaders; they also allow performing all alpha-blending operations on fp16×4 filtered surfaces. This permits intermediate rendered buffers at a much higher precision and range, enabling high-dynamic-range rendering, motion blur, and many other effects. In addition, it is possible to specify a separate blending function for color and alpha values. (The lowest-end member of the GeForce 6 Series family, the GeForce 6200 TC, does not support floating-point blending or floating-point texture filtering because of its lower memory bandwidth, as well as to save area on the chip.)

30.3.2 Shader Model 3.0 Programming Model

Along with the fixed-function features listed previously, the capabilities of the vertex and the fragment processors have been enhanced in GeForce 6 Series GPUs. With Shader Model 3.0, the programming models for vertex and fragment processors are converging: both support fp32 precision, texture lookups, and the same instruction set. Specifically, here are the new features that have been added.

Vertex Processor

- **Increased instruction count.** The total instruction count is now 512 static instructions and 65,536 dynamic instructions. The static instruction count represents the number of instructions in a program as it is compiled. The dynamic instruction count represents the number of instructions actually executed. In practice, the dynamic count can be much higher than the static count due to looping and subroutine calls.
- **More temporary registers.** Up to 32 four-wide temporary registers can be used in a vertex program.
- **Support for instancing.** This enhancement was described earlier.

- **Dynamic flow control.** Branching and looping are now part of the shader model. On the GeForce 6 Series vertex engine, branching and looping have minimal overhead of just two cycles. Also, each vertex can take its own branches without being grouped in the way pixel shader branches are. So as branches diverge, the GeForce 6 Series vertex processor still operates efficiently.
- **Vertex texturing.** Textures can now be fetched in a vertex program, although only nearest-neighbor filtering is supported in hardware. More advanced filters can of course be implemented in the vertex program. Up to four unique textures can be accessed in a vertex program, although each texture can be accessed multiple times. Vertex textures generate latency for fetching data, unlike true constant reads. Therefore, the best way to use vertex textures is to do a texture fetch and follow it with arithmetic operations to hide the latency before using the result of the texture fetch.

Each vertex engine is capable of simultaneously performing a four-wide SIMD `MAD` (multiply-add) instruction and a scalar special function per clock cycle. Special function instructions include:

- Exponential functions: `EXP`, `EXPP`, `LIT`, `LOG`, `LOGP`
- Reciprocal instructions: `RCP`, `RSQ`
- Trigonometric functions: `SIN`, `COS`

Fragment Processor

- **Increased instruction count.** The total instruction count is now 65,535 static instructions and 65,535 dynamic instructions. There are limitations on how long the operating system will wait while the shader finishes working, so a long shader program working on a full screen of pixels may time-out. This makes it important to carefully consider the shader length and number of fragments rendered in one draw call. In practice, the number of instructions exposed by the driver tends to be smaller, because the number of instructions can expand as code is translated from Direct3D pixel shaders or OpenGL fragment programs to native hardware instructions.
- **Multiple render targets.** The fragment processor can output to up to four separate color buffers, along with a depth value. All four separate color buffers must be the same format and size. MRTs can be particularly useful when operating on scalar data, because up to 16 scalar values can be written out in a single pass by the fragment processor. Sample uses of MRTs include particle physics, where positions and velocities are computed simultaneously, and similar GPGPU algorithms. Deferred shading is another technique that computes and stores multiple four-component floating-point values simultaneously: it computes all material properties and stores them in

separate textures. So, for example, the surface normal and the diffuse and specular material properties could be written to textures, and the textures could all be used in subsequent passes when lighting the scene with multiple lights. This is illustrated in Figure 30-8.

- **Dynamic flow control (branching).** Shader Model 3.0 supports conditional branching and looping, allowing for more flexible shader programs.
- **Indexing of attributes.** With Shader Model 3.0, an index register can be used to select which attributes to process, allowing for loops to perform the same operation on many different inputs.
- **Up to ten full-function attributes.** Shader Model 3.0 supports ten full-function attributes/texture coordinates, instead of Shader Model 2.0's eight full-function attributes plus specular color and diffuse color. All ten Shader Model 3.0 attributes are interpolated at full fp32 precision, whereas Shader Model 2.0's diffuse and specular color were interpolated at only 8-bit integer precision.
- **Centroid sampling.** Shader Model 3.0 allows a per-attribute selection of center sampling, or *centroid sampling*. Centroid sampling returns a value inside the covered portion of the fragment, instead of at the center, and when used with multisampling, it can remove some artifacts associated with sampling outside the polygon (for example, when calculating diffuse or specular color using texture coordinates, or when using texture atlases).
- **Support for fp32 and fp16 internal precision.** Fragment programs can support full fp32-precision computations and intermediate storage or partial-precision fp16 computations and intermediate storage.

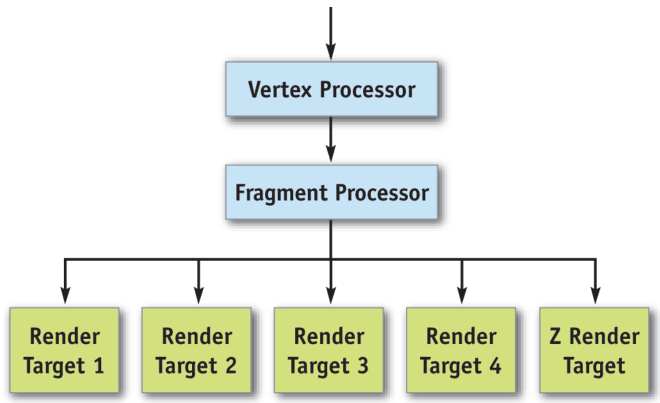


Figure 30-8. How MRTs Work
MRTs make it possible for a fragment program to return four four-wide color values plus a depth value.

- **3:1 and 2:2 coissue.** Each four-component-wide vector unit is capable of executing two independent instructions in parallel, as shown in Figure 30-9: either one three-wide operation on RGB and a separate operation on alpha, or one two-wide operation on red-green and a separate two-wide operation on blue-alpha. This gives the compiler more opportunity to pack scalar computations into vectors, thereby doing more work in a shorter time.
- **Dual issue.** Dual issue is similar to coissue, except that the two independent instructions can be executed on different parts of the shader pipeline. This makes the pipeline easier to schedule and, therefore, more efficient. See Figure 30-10.

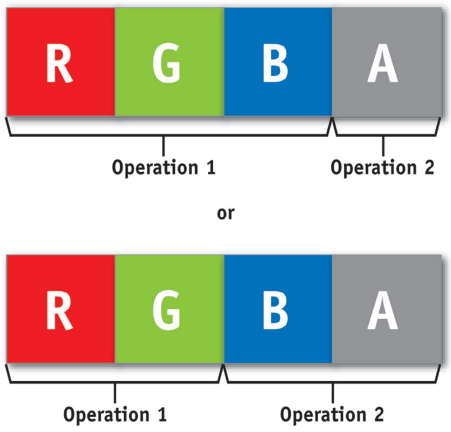


Figure 30-9. How Coissue Works
Two separate operations can concurrently execute on different parts of a four-wide register.

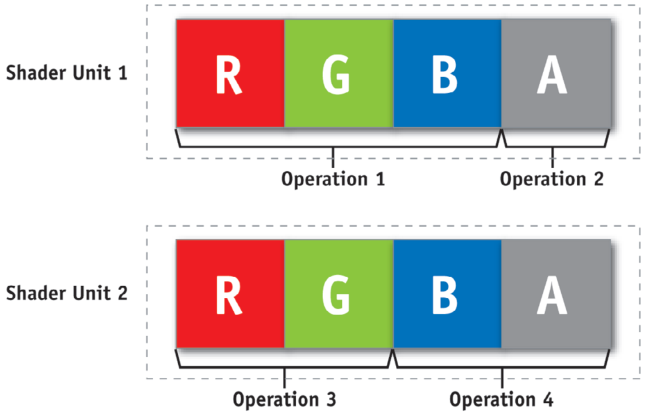


Figure 30-10. How Dual Issue Works
Independent instructions can be executed on independent units in the computational pipeline.

Fragment Processor Performance

The GeForce 6 Series fragment processor architecture has the following performance characteristics:

- Each pipeline is capable of performing a four-wide, coissue-able multiply-add (`MAD`) or four-term dot product (`DP4`), plus a four-wide, coissue-able and dual-issuable multiply instruction per clock in series, as shown in Figure 30-11. In addition, a multifunction unit that performs complex operations can replace the alpha channel `MAD` operation. Operations are performed at full speed on both fp32 and fp16 data, although storage and bandwidth limitations can favor fp16 performance sometimes. In practice, it is sometimes possible to execute eight math operations and a texture lookup in a single cycle.
- Dedicated fp16 normalization hardware exists, making it possible to normalize a vector at fp16 precision in parallel with the multiplies and `MAD`s just described.
- An independent reciprocal operation can be performed in parallel with the multiply, `MAD`, and fp16 normalization described previously.
- Because the GeForce 6800 has 16 fragment-processing pipelines, the overall available performance of the system is given by these values multiplied by 16 and then by the clock rate.
- There is some overhead to flow-control operations, as defined in Table 30-2.

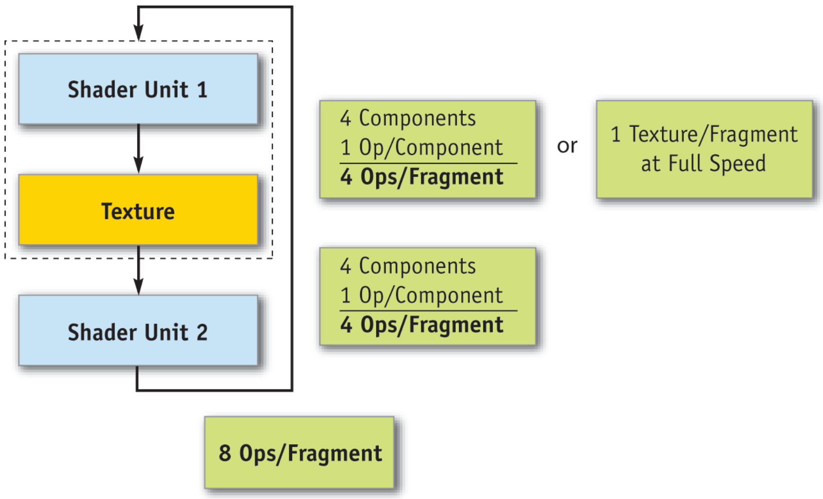


Figure 30-11. Shader Units and Capabilities in the Fragment Processor

Table 30-2. Overhead Incurred When Executing Flow-Control Operations in Fragment Programs

Instruction	Cost (Cycles)
`If` / `endif`	4
`If` /`else` / `endif`	6
`Call`	2
`Ret`	2
`Loop` / `endloop`	4

Furthermore, branching in the fragment processor is affected by the level of divergence of the branches. Because the fragment processor operates on hundreds of pixels per instruction, if a branch is taken by some fragments and not others, all fragments execute both branches, but only writing to the registers on the branches each fragment is supposed to take. For low-frequency and mid-frequency branch changes, this effect is hidden, although it can become a limiter as the branch frequency increases.

30.3.3 Supported Data Storage Formats

Table 30-3 summarizes the data formats supported by the graphics pipeline.

30.4 Performance

The GeForce 6800 Ultra is the flagship product of the GeForce 6 Series family at the time of writing. Its performance is summarized as follows:

- 425 MHz internal graphics clock
- 550 MHz memory clock
- 600 million vertices/second
- 6.4 billion texels/second
- 12.8 billion pixels/second, rendering z/stencil-only (useful for shadow volumes and shadow buffers)
- 6 four-wide fp32 vector `MAD`s per clock cycle in the vertex shader, plus one scalar multifunction operation (a complex math operation, such as a sine or reciprocal square root)
- 16 four-wide fp32 vector `MAD`s per clock cycle in the fragment processor, plus 16 four-wide fp32 multiplies per clock cycle
- 64 pixels per clock cycle early z-cull (reject rate)

As you can see, there's plenty of programmable floating-point horsepower in the vertex and fragment processors that can be exploited for computationally demanding problems.

Table 30-3. Data Storage Formats Supported by GeForce 6 Series GPUs ✓ = Yes ✗ = No

Format	Description of Data in Memory	Vertex Texture Support	Fragment Texture Support	Render Target Support
B8	One 8-bit fixed-point number	✗	✓	✓
A1R5G5B5	A 1-bit value and three 5-bit unsigned fixed-point numbers	✗	✓	✓
A4R4G4B4	Four 4-bit unsigned fixed-point numbers	✗	✓	✗
R5G6B5	5-bit, 6-bit, and 5-bit fixed-point numbers	✗	✓	✓
A8R8G8B8	Four 8-bit fixed-point numbers	✗	✓	✓
DXT1	Compressed 4×4 pixels into 8 bytes	✗	✓	✗
DXT2,3,4,5	Compressed 4×4 pixels into 16 bytes	✗	✓	✗
G8B8	Two 8-bit fixed-point numbers	✗	✓	✓
B8R8_G8R8	Compressed as YVYU; two pixels in 32 bits	✗	✓	✗
R8B8_R8G8	Compressed as VYUY; two pixels in 32 bits	✗	✓	✗
R6G5B5	6-bit, 5-bit, and 5-bit unsigned fixed-point numbers	✗	✓	✗
DEPTH24_D8	A 24-bit unsigned fixed-point number and 8 bits of garbage	✗	✓	✓
DEPTH24_D8_FLOAT	A 24-bit unsigned float and 8 bits of garbage	✗	✓	✓
DEPTH16	A 16-bit unsigned fixed-point number	✗	✓	✓
DEPTH16_FLOAT	A 16-bit unsigned float	✗	✓	✓
X16	A 16-bit fixed-point number	✗	✓	✗
Y16_X16	Two 16-bit fixed-point numbers	✗	✓	✗
R5G5B5A1	Three unsigned 5-bit fixed-point numbers and a 1-bit value	✗	✓	✓
HILO8	Two unsigned 16-bit values compressed into two 8-bit values	✗	✓	✗
HILO_S8	Two signed 16-bit values compressed into two 8-bit values	✗	✓	✗
W16_Z16_Y16_X16 FLOAT	Four fp16 values	✗	✓	✓
W32_Z32_Y32_X32 FLOAT	Four fp32 values	✓ (unfiltered)	✓ (unfiltered)	✓
X32_FLOAT	One 32-bit floating-point number	✓ (unfiltered)	✓ (unfiltered)	✓
D1R5G5B5	1 bit of garbage and three unsigned 5-bit fixed-point numbers	✗	✓	✓
D8R8G8B8	8 bits of garbage and three unsigned 8-bit fixed-point numbers	✗	✓	✓
Y16_X16 FLOAT	Two 16-bit floating-point numbers	✗	✓	✗

30.5 Achieving Optimal Performance

While graphics hardware is becoming more and more programmable, there are still some tricks to ensuring that you exploit the hardware fully to get the most performance. This section lists some common techniques that you may find helpful. A more detailed discussion of performance advice is available in the *NVIDIA GPU Programming Guide*, which is freely available in several languages from the NVIDIA Developer Web site (http://developer.nvidia.com/object/gpu_programming_guide.html).

30.5.1 Use Z-Culling Aggressively

Z-cull avoids work that won't contribute to the final result. It's better to determine early on that a computation doesn't matter and save doing the work. In graphics, this can be done by rendering the z-values for all objects first, before shading. For general-purpose computation, the z-cull unit can be used to select which parts of the computation are still active, culling computational threads that have already resolved. See Section 34.2.3 of Chapter 34, "GPU Flow-Control Idioms," for more details on this idea.

30.5.2 Exploit Texture Math When Loading Data

The texture unit filters data before returning it to the fragment processor, thus reducing the total data needed by the shader. The texture unit's bilinear filtering can frequently be used to reduce the total work done by the shader if it's performing more sophisticated shading.

Often, large filter kernels can be dissected into groups of bilinear footprints, which are scaled and accumulated to build the large kernel. A few caveats apply here, most notably that all filter coefficients must be positive for bilinear footprint assembly to work properly. (See Chapter 20, "Fast Third-Order Texture Filtering," for more information about this technique.)

Similarly, the filtering support given by shadow buffering can be used to offload the work from the processor when performing compares, then filtering the results.

30.5.3 Use Branching in Fragment Programs Judiciously

Because the fragment processor is a SIMD machine operating on many fragments at a time, if some fragments in a given group take one branch and other fragments in that group take another branch, the fragment processor needs to take both branches. Also, there is a six-cycle overhead for if-else-endif control structures. These two effects can reduce the performance of branching programs if not considered carefully. Branching can be very beneficial, as long as the work avoided outweighs the cost of branching.

Alternatively, conditional writes (that is, write if a condition code is set) can be used when branching is not performance-effective. In practice, the compiler will use the method that delivers higher performance when possible.

30.5.4 Use fp16 Intermediate Values Wherever Possible

Because GeForce 6 Series GPUs support a full-speed fp16 normalize instruction in parallel with the multiplies and adds, and because fp16 intermediate values reduce internal storage and datapath requirements, using fp16 intermediate values wherever possible can be a performance win, saving fp32 intermediate values for cases where the precision is needed.

Excessive internal storage requirements can adversely affect performance in the following way: The shader pipeline is optimized to keep hundreds of fragments in flight given a fixed amount of register space per fragment (four fp32×4 registers or eight fp16×4 registers). If the register space is exceeded, then fewer fragments can remain in flight, reducing the latency tolerance for texture fetches, and adversely affecting performance. The GeForce 6 Series fragment processor will have the maximum number of fragments in flight when shader programs use up to four fp32×4 temporary registers (or eight fp16×4 registers). That is, at any one time, a maximum of four temporary fp32×4 (or eight fp16×4) registers are in use. This decision was based on the fact that for the overwhelming majority of analyzed shaders, four or fewer simultaneously active fp32×4 registers proved to be the sweet spot during the shaders' execution. In addition, the architecture is designed so that performance degrades slowly if more registers are used.

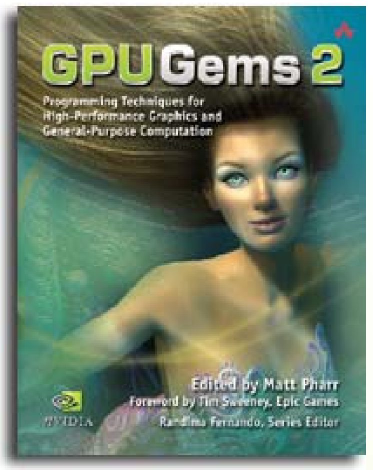
Similarly, the register file has enough read and write bandwidth to keep all the units busy if reading fp16×4 values, but it may run out of bandwidth to feed all units if using fp32×4 values exclusively. NVIDIA's compiler technology is smart enough to reduce this effect's impact substantially, but fp16 intermediate values are never slower than fp32 values; because of the resource restrictions and the fp16 normalize hardware, they can often be much faster.

30.6 Conclusion

GeForce 6 Series GPUs provide the GPU programmer with unparalleled flexibility and performance in a product line that spans the entire PC market. After reading this chapter, you should have a better understanding of what GeForce 6 Series GPUs are capable of, and you should be able to use this knowledge to develop applications—either graphical or general purpose—in a more efficient way.

GPU Gems 2

Programming Techniques for High-Performance Graphics and General-Purpose Computation

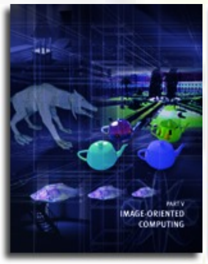

GPU Gems 2
Programming Techniques for High-Performance Graphics and General-Purpose Computation
Edited by Matt Pharr
Foreword by Tim Sweeney, Epic Games
Randima Fernando, Series Editor
NVIDIA

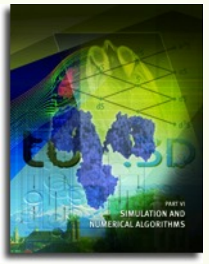
- 880 full-color pages, 330 figures
- Hard cover
- $59.99
- Available at GDC 2005 (March 7, 2005)
- Experts from universities and industry

Graphics Programming

- Geometric Complexity
- Shading, Lighting, and Shadows
- High-Quality Rendering

GPGPU Programming





- General Purpose Computation on GPUs: A Primer
- Image-Oriented Computing
- Simulation and Numerical Algorithms

Sign up for e-mail notification when the book is available at:
http://developer.nvidia.com/object/gpu_gems_2_notification.html

For more information, please visit:
http://developer.nvidia.com/object/gpu_gems_2_home.html