Readings on graphics architecture for Advanced Computer Architecture class

Attached are several short readings on graphics architecture. They are a mix of application-focused and hardware-focused readings. As is the case for most specialized architectures – particularly specialized parallel architectures – one needs an understanding of the application domain in order to understand the hardware architecture so these readings combine application and hardware discussion.

The readings are:

1. pp. 13-20 from "The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics", 2003.
   This reading provides an introduction to the functionality and high-level organization of graphics hardware.

2. Short excerpts from "A user-programmable vertex engine", Lindholm et al, SIGGRAPH 2001.
   This reading provides an introduction to the ISA used in the vertex engine of graphics processors, along with a short description of the typical pipelined and multithreaded microarchitecture of a programmable core.

3. Texture caching and prefectching, pp. 689-690 in Real-Time 3D Graphics, 2nd edition, Akenine-Moller and Haines, 2002.
   This reading provides a brief introduction to the caching, prefetching, and multithreading mechanisms used to hide latency and reduce off-chip bandwidth requirements for the memory accesses required by texture mapping.

4. The GeForce 6 Series GPU Architecture (sections 30.1-30.4), from GPU Gems 2, 2005.
   This reading is the most up-to-date public description of a graphics architecture.

If you're interested in additional readings on this topic, send me (Bill Mark) an email, and I can provide you with additional references.

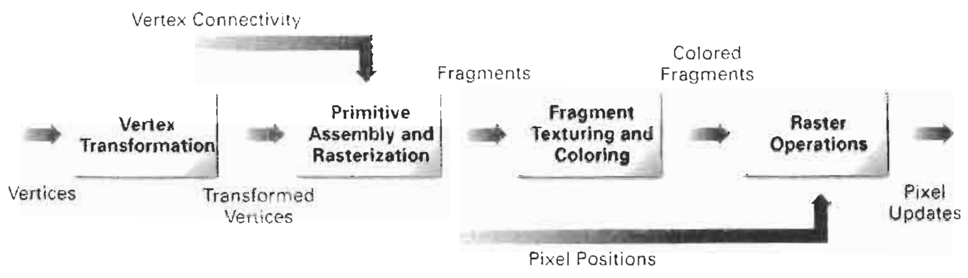

Figure 1-3. The Graphics Hardware Pipeline

1.2.3 The Graphics Hardware Pipeline

A *pipeline* is a sequence of stages operating in parallel and in a fixed order. Each stage receives its input from the prior stage and sends its output to the subsequent stage. Like an assembly line where dozens of automobiles are manufactured at the same time, with each automobile at a different stage of the line, a conventional graphics hardware pipeline processes a multitude of vertices, geometric primitives, and fragments in a pipelined fashion.

Figure 1-3 shows the graphics hardware pipeline used by today's GPUs. The 3D application sends the GPU a sequence of vertices batched into geometric primitives: typically polygons, lines, and points. As shown in Figure 1-4, there are many ways to specify geometric primitives.

Every vertex has a position but also usually has several other attributes such as a color, a secondary (or *specular*) color, one or multiple texture coordinate sets, and a normal vector. The normal vector indicates what direction the surface faces at the vertex, and is typically used in lighting calculations.

Vertex Transformation

Vertex transformation is the first processing stage in the graphics hardware pipeline. Vertex transformation performs a sequence of math operations on each vertex. These operations include transforming the vertex position into a screen position for use by the rasterizer, generating texture coordinates for texturing, and lighting the vertex to determine its color. We will explain many of these tasks in subsequent chapters.

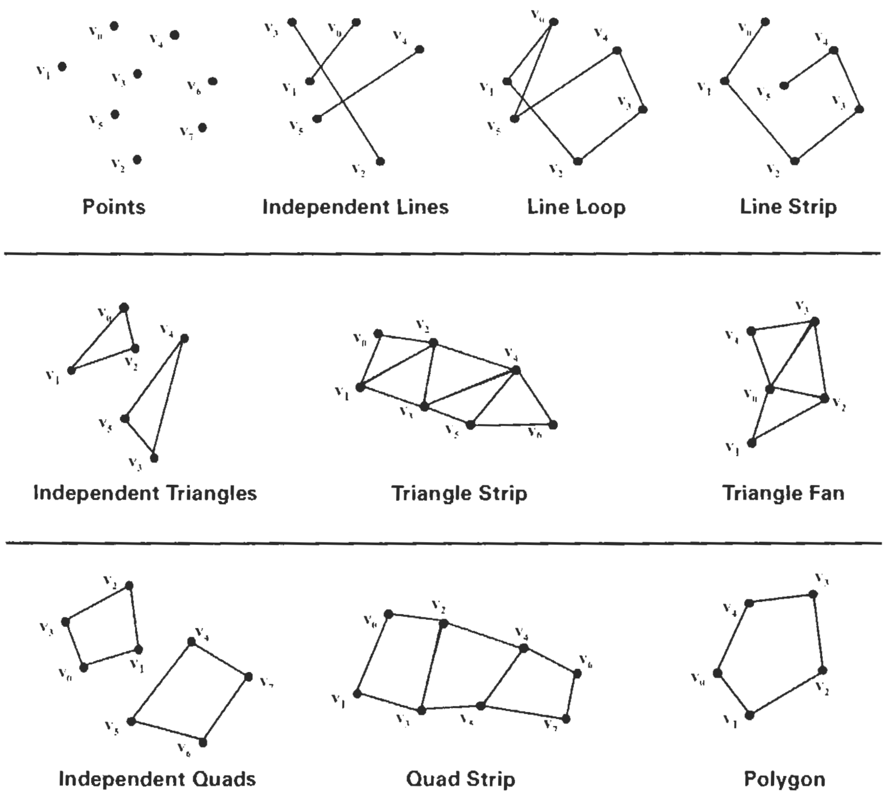

Figure 1-4. Types of Geometric Primitives

Primitive Assembly and Rasterization

The transformed vertices flow in sequence to the next stage, called *primitive assembly and rasterization*. First, the primitive assembly step assembles vertices into geometric primitives based on the geometric primitive batching information that accompanies the sequence of vertices. This results in a sequence of triangles, lines, or points. These primitives may require clipping to the *view frustum* (the view's visible region of 3D space), as well as any enabled application-specified clip planes. The rasterizer may also discard polygons based on whether they face forward or backward. This process is known as *culling*.

Polygons that survive these clipping and culling steps must be rasterized. Rasterization is the process of determining the set of pixels covered by a geometric primitive. Polygons, lines, and points are each rasterized according to the rules specified for each type of primitive. The results of rasterization are a set of pixel locations as well as a set of fragments. There is no relationship between the number of vertices a primitive has and the number of fragments that are generated when it is rasterized. For example, a triangle made up of just three vertices could take up the entire screen, and therefore generate millions of fragments!

Earlier, we told you to think of a fragment as a pixel if you did not know precisely what a fragment was. At this point, however, the distinction between a fragment and a pixel becomes important. The term *pixel* is short for "picture element." A pixel represents the contents of the frame buffer at a specific location, such as the color, depth, and any other values associated with that location. A *fragment* is the state required potentially to update a particular pixel.

The term "fragment" is used because rasterization breaks up each geometric primitive, such as a triangle, into pixel-sized fragments for each pixel that the primitive covers. A fragment has an associated pixel location, a depth value, and a set of interpolated parameters such as a color, a secondary (specular) color, and one or more texture coordinate sets. These various interpolated parameters are derived from the transformed vertices that make up the particular geometric primitive used to generate the fragments. You can think of a fragment as a "potential pixel." If a fragment passes the various rasterization tests (in the raster operations stage, which is described shortly), the fragment updates a pixel in the frame buffer.

Interpolation, Texturing, and Coloring

Once a primitive is rasterized into a collection of zero or more fragments, the *interpolation, texturing, and coloring* stage interpolates the fragment parameters as necessary, performs a sequence of texturing and math operations, and determines a final color for each fragment. In addition to determining the fragment's final color, this stage may also determine a new depth or may even discard the fragment to avoid updating the frame buffer's corresponding pixel. Allowing for the possibility that the stage may discard a fragment, this stage emits one or zero colored fragments for every input fragment it receives.

Raster Operations

The *raster operations* stage performs a final sequence of per-fragment operations immediately before updating the frame buffer. These operations are a standard part of OpenGL and Direct3D. During this stage, hidden surfaces are eliminated through a

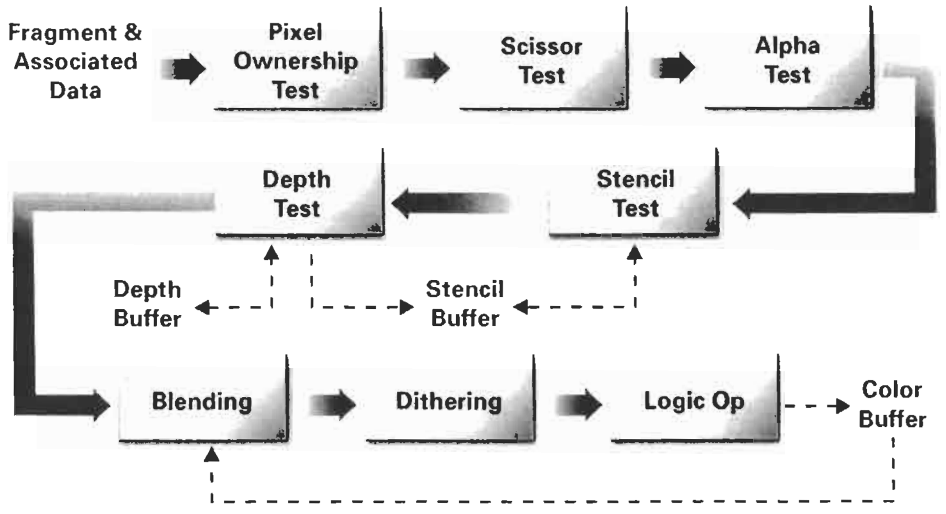

Figure 1-5. Standard OpenGL and Direct3D Raster Operations

process known as *depth testing*. Other effects, such as blending and stencil-based shadowing, also occur during this stage.

The raster operations stage checks each fragment based on a number of tests, including the scissor, alpha, stencil, and depth tests. These tests involve the fragment's final color or depth, the pixel location, and per-pixel values such as the depth value and stencil value of the pixel. If any test fails, this stage discards the fragment without updating the pixel's color value (though a stencil write operation may occur). Passing the depth test may replace the pixel's depth value with the fragment's depth. After the tests, a blending operation combines the final color of the fragment with the corresponding pixel's color value. Finally, a frame buffer write operation replaces the pixel's color with the blended color. Figure 1-5 shows this sequence of operations.

Figure 1-5 shows that the raster operations stage is actually itself a series of pipeline stages. In fact, all of the previously described stages can be broken down into substages as well.

Visualizing the Graphics Pipeline

Figure 1-6 depicts the stages of the graphics pipeline. In the figure, two triangles are rasterized. The process starts with the transformation and coloring of vertices. Next, the primitive assembly step creates triangles from the vertices, as the dotted lines indi-

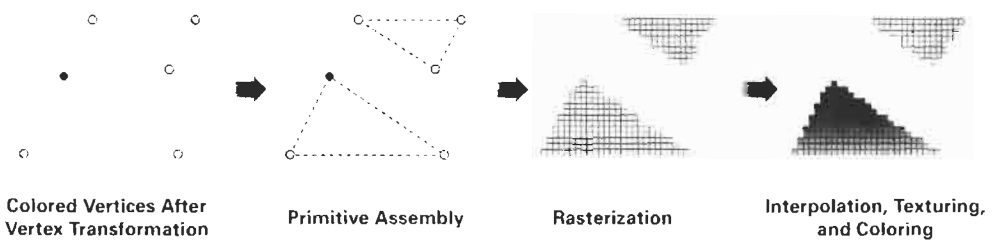
Colored Vertices After Vertex Transformation
Primitive Assembly
Rasterization
Interpolation, Texturing, and Coloring

Figure 1-6. Visualizing the Graphics Pipeline

cate. After this, the rasterizer "fills in" the triangles with fragments. Finally, the register values from the vertices are interpolated and used for texturing and coloring. Notice that many fragments are generated from just a few vertices.

1.2.4 The Programmable Graphics Pipeline

The dominant trend in graphics hardware design today is the effort to expose more programmability within the GPU. Figure 1-7 shows the vertex processing and fragment processing stages in the pipeline of a programmable GPU.

Figure 1-7 shows more detail than Figure 1-3, but more important, it shows the vertex and fragment processing broken out into programmable units. The *programmable*

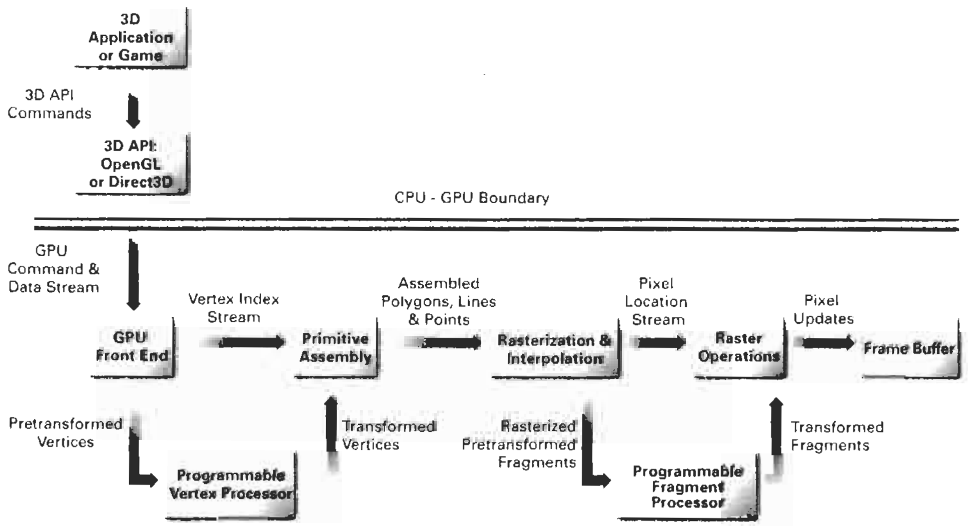

Figure 1-7. The Programmable Graphics Pipeline

vertex processor is the hardware unit that runs your Cg vertex programs, whereas the *programmable fragment processor* is the unit that runs your Cg fragment programs.

As explained in Section 1.2.2, GPU designs have evolved, and the vertex and fragment processors within the GPU have transitioned from being configurable to being programmable. The descriptions in the next two sections present the critical functional features of programmable vertex and fragment processors.

The Programmable Vertex Processor

Figure 1-8 shows a flow chart for a typical programmable vertex processor. The data-flow model for vertex processing begins by loading each vertex's attributes (such as

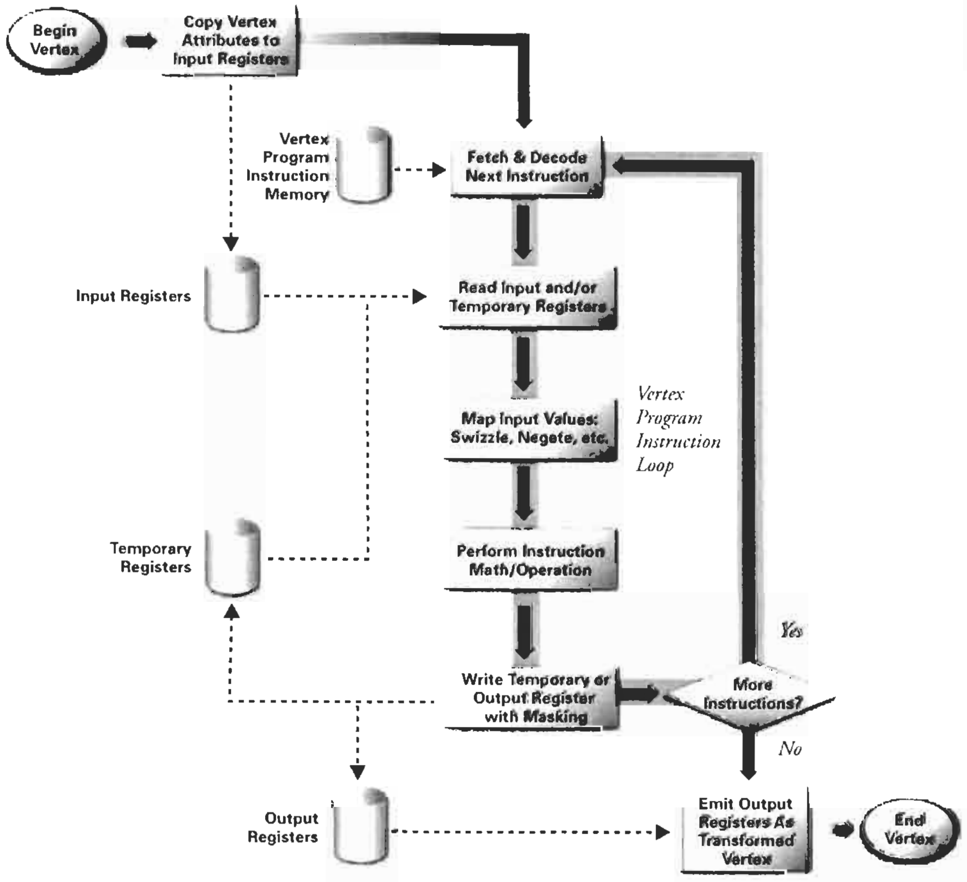
Begin Vertex
Copy Vertex Attributes to Input Registers
Vertex Program Instruction Memory
Fetch & Decode Next Instruction
Input Registers
Read Input and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Vertex Program Instruction Loop
Temporary Registers
Perform Instruction Math/Operation
Yes
Write Temporary or Output Register with Masking
More Instructions?
No
Output Registers
Emit Output Registers As Transformed Vertex
End Vertex

Figure 1-8. Programmable Vertex Processor Flow Chart

position, color, texture coordinates, and so on) into the vertex processor. The vertex processor then repeatedly fetches the next instruction and executes it until the vertex program terminates. Instructions access several distinct sets of registers banks that contain vector values, such as position, normal, or color. The vertex attribute registers are read-only and contain the application-specified set of attributes for the vertex. The temporary registers can be read and written and are used for computing intermediate results. The output result registers are write-only. The program is responsible for writing its results to these registers. When the vertex program terminates, the output result registers contain the newly transformed vertex. After triangle setup and rasterization, the interpolated values for each register are passed to the fragment processor.

Most vertex processing uses a limited palette of operations. Vector math operations on floating-point vectors of two, three, or four components are necessary. These operations include add, multiply, multiply-add, dot product, minimum, and maximum. Hardware support for vector negation and component-wise swizzling (the ability to reorder vector components arbitrarily) generalizes these vector math instructions to provide negation, subtraction, and cross products. Component-wise write masking controls the output of all instructions. Combining reciprocal and reciprocal square root operations with vector multiplication and dot products, respectively, enables vector-by-scalar division and vector normalization. Exponential, logarithmic, and trigonometric approximations facilitate lighting, fog, and geometric computations. Specialized instructions can make lighting and attenuation functions easier to compute.

Further functionality, such as relative addressing of constants and flow-control support for branching and looping, is also available in more recent programmable vertex processors.

The Programmable Fragment Processor

Programmable fragment processors require many of the same math operations as programmable vertex processors do, but they also support texturing operations. Texturing operations enable the processor to access a texture image using a set of texture coordinates and then to return a filtered sample of the texture image.

Newer GPUs offer full support for floating-point values; older GPUs have more limited fixed-point data types. Even when floating-point operations are available, fragment operations are often more efficient when using lower-precision data types. GPUs must process so many fragments at once that arbitrary branching is not available in current GPU generations, but this is likely to change over time as hardware evolves.

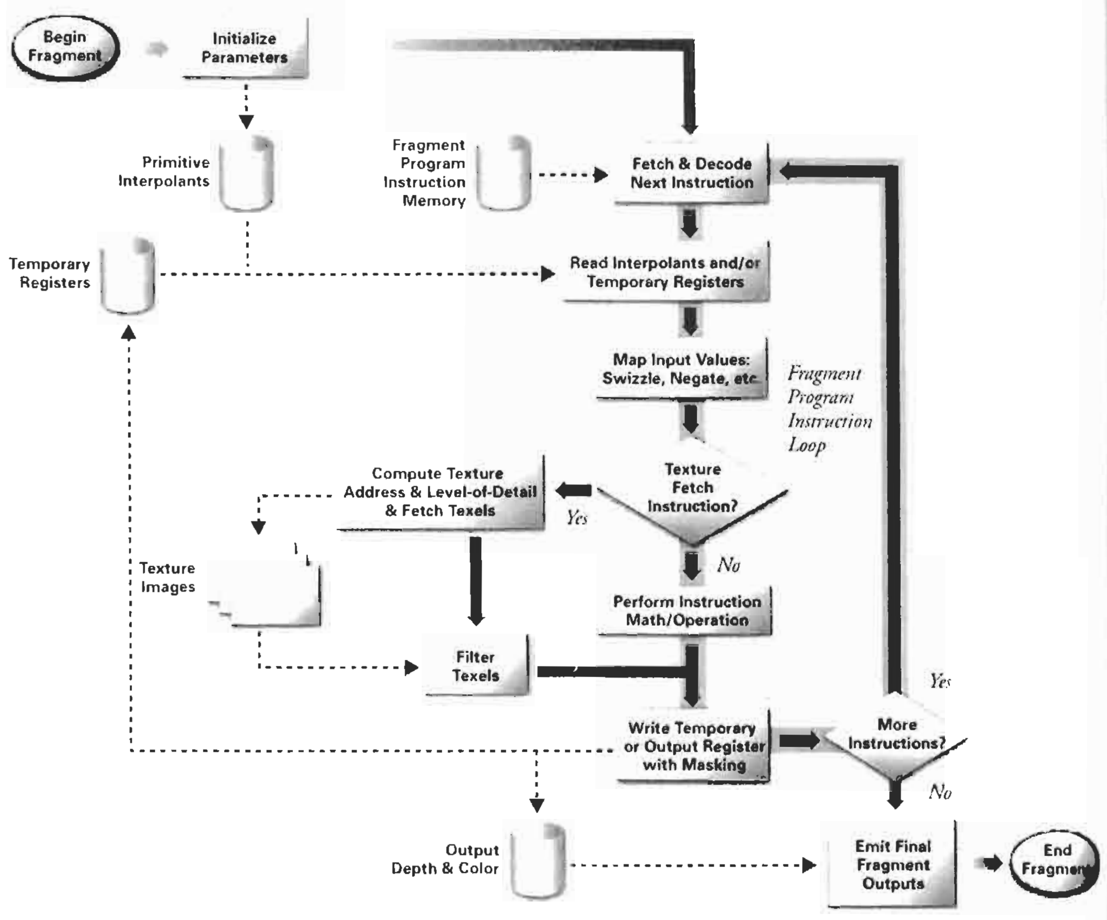
Begin Fragment
Initialize Parameters
Primitive Interpolants
Fragment Program Instruction Memory
Fetch & Decode Next Instruction
Temporary Registers
Read Interpolants and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Fragment Program Instruction Loop
Compute Texture Address & Level-of-Detail & Fetch Texels
Texture Fetch Instruction?
Yes
No
Texture Images
Perform Instruction Math/Operation
Filter Texels
Yes
Write Temporary or Output Register with Masking
More Instructions?
No
Output Depth & Color
Emit Final Fragment Outputs
End Fragment

Figure 1-9. Programmable Fragment Processor Flow Chart

Cg still allows you to write fragment programs that branch and iterate by simulating such constructs with conditional assignment operations or loop unrolling.

Figure 1-9 shows the flow chart for a current programmable fragment processor. As with a programmable vertex processor, the data flow involves executing a sequence of instructions until the program terminates. Again, there is a set of input registers. However, rather than vertex attributes, the fragment processor's read-only input registers contain interpolated per-fragment parameters derived from the per-vertex parameters of the fragment's primitive. Read/write temporary registers store intermediate values. Write operations to write-only output registers become the color and optionally the new depth of the fragment. Fragment program instructions include texture fetches.

From "A user-programmable vertex engine"
Lindholm et al.
SIGGRAPH 2001

3.7 Instruction Set

The instruction set consists of 17 operations. These can be divided into vector, scalar, and miscellaneous operation. We discuss the instructions selected after explaining the constraints we chose to impose.

OpCode	Full Name	Description
MOV	Move	vector -> vector
MUL	Multiply	vector -> vector
ADD	Add	vector -> vector
MAD	Multiply and add	vector -> vector
DST	Distance	vector -> vector
MIN	Minimum	vector -> vector
MAX	Maximum	vector -> vector
SLT	Set on less than	vector -> vector
SGE	Set on greater or equal	vector -> vector
RCP	Reciprocal	scalar-> replicated scalar
RSQ	Reciprocal square root	scalar-> replicated scalar
DP3	3 term dot product	vector-> replicated scalar
DP4	4 term dot product	vector-> replicated scalar
LOG	Log base 2	miscellaneous
EXP	Exp base 2	miscellaneous
LIT	Phong lighting	miscellaneous
ARL	Address register load	miscellaneous

Table 2: Instruction Set

~~3.7.1 No Branching~~

~~The fixed function transform paths in OpenGL®[25] and Direct3D™[6] are both controlled by global state that does not depend on the actual data supplied with each vertex. This allows for driver optimizations at the time the first vertex is supplied by the application since all subsequent vertices (until a new state change) can then share this carefully optimized code. The result is a code segment that removes state checking and branching. It is therefore possible to support the full fixed function transform path (at least to homogenous clip space) without branching. The decision was therefore made to not support branching, keeping the hardware as simple as possible. Also, late binding changes in control flow disrupt pipeline efficiency. Simple if/then/else evaluation is still supported through sum-of-products using 1.0 and 0.0, which can be generated with SLT and SGE.~~

Newer machines have branching

3.7.2 Constant Latency

One instruction set constraint we imposed was that our hardware implementation must issue any instruction per clock and execute all instructions with the same latency, limiting the complexity of any instruction. This improves programmability and simplifies the hardware. All operands are immediately available, limiting the size of register and memory banks.

3.7.3 Instruction Set Rationale

Since we wanted to use the same instruction set for vertex programs and fixed function (non-programmable) mode, we started by analyzing the fixed function implementation of a previous architecture. We found that the equivalents of the MOV, MUL, ADD, and MAD instructions were used about 50% of the time, and that the DP3, and DP4 equivalents were used about 40% of the time. We support dot products for their coding convenience, and also because as the number of cycles spent on a vertex decreases over architectural generations, it becomes more important to have powerful concise instructions. Cross products are also important, and they can be done via an efficient MUL, MAD sequence with source vector rotations. For example, R1 = R0×R2 is done as:

```
MUL R1, R0.zxyw, R2.yzxw ;
MAD R1, R0.yzxw, R2.zxyw, -R1;
```

We support reciprocal (RCP) instead of division due to the constant latency restriction. The RCP instruction is also scalar since the main use of it is in the perspective division of w in homogeneous clip space (done after the vertex program) which involves the multiply of the (x,y,z) vector with the scalar $1/w$.

The reciprocal square root (RSQ) is mainly used in normalizing vectors to be used in lighting equations. The typical sequence is a DP3 to find the vector length squared, a RSQ to get the reciprocal length, and a MUL to normalize the vector. It is very convenient to use the vector w component for storing the length squared and reciprocal length values. RSQ is also a scalar operator.

To avoid problems with vector lengths of 0.0 causing RSQ to return infinity, we mandated that 0.0 times anything be 0.0. This is also useful in conditional evaluation when multiplying by 0.0. Another mandate is that 1.0 times anything be the same value.

A major exception to our goal of similar performance in fixed function and program mode involved lighting. The previous architecture design has a separate hard-wired lighting engine. Since it was too hard to expose this engine in program mode, the decision was made to turn it off when running vertex programs. Fixed function performance with heavy lighting can therefore be twice as fast as a comparable vertex program. To alleviate this problem, two instructions were included: DST and LIT. The DST instruction assists in constructing attenuation factors of the form:

$$(K0,K1,K2) \bullet (1,d,d*d) \;=\; K0 + K1*d + K2*d*d$$

where d is some distance. Since $d*d$ and $1/d$ are natural byproducts of the vector normalization process, these values are input as (NA,d*d,d*d,NA) and (NA,1/d,NA,1/d)) to DST, which then returns the $(1,d,d*d,1/d)$ vector. The last $1/d$ term can be used with a DP4 operation if desired.

The LIT instruction does the fairly complex ambient, diffuse, and specular calculations with clamping based on **N•L**, **N•H**, and the power p. The calculations are:

```
Output.x = 1.0;                     // ambient
Output.y = max(N•L,0.0);            // diffuse
Output.z = 0.0;                     // specular
if (N•L > 0.0 && p == 0.0)
  Output.z = 1.0;
else if (N•L > 0.0 && N•H > 0.0)
  Output.z = (N•H)^p;
Output.w = 1.0;
```

Since LIT implements the specular power function via use of a log, multiply, and exp sequence, we also decided to expose the LOG and EXP instructions. Since the power is a variable in the LIT source, a table needing a pre-known specular power was not an option. We also wanted an accurate power function conforming to the cos^n model; hence known approximations would not suffice. It is possible to implement the LIT instruction with about 10 other instructions, but the performance loss is extreme.

The LOG base 2 instruction returns an output accurate to about 11 mantissa bits as well as two partial results: the exponent and mantissa of the source scalar. A more accurate user programmed approximation based on the limited range mantissa can be done with the result added to the exponent. The EXP base 2 instruction also returns an output accurate to about 11 mantissa bits as well as two partial results: two raised to power of *floor*(source) and *fraction*(source). A more accurate user programmed approximation based on the limited range fraction can be done with the result multiplied by the power output. The precision of these instructions was based on the desired 8-bit color precision of the specular LIT operation. It takes about 10 instructions to achieve full accuracy LOG and EXP evaluation.

The MIN and MAX operations allow for clamping and absolute value computations (MAX of source and -source). Related to these are the SLT and SGE instructions that return 1.0 if the component compare is true and 0.0 if false.

The ARL instruction was added to allow support of vertex specific constant access such as a matrix or plane equation. It converts a floating-point scalar into a signed integer, which can be used as an offset into the constant memory. Out-of-range reads from the constant memory return (0,0,0,0).

Sources are negated by prefixing a "-" sign, and can be swizzled via four optional subscripts that describe the component rearrangement desired. For example:

```
MOV R0, -R1.wyzy ;
```

moves the negated w component of register R1 into the x component of register R0, moves the negated y and z components across, and uses the negated y component again to place into the R0 w component.

The destination of an instruction has an optional write mask of the desired $xyzw$ components to be written. For example:

```
ADD R0.xw, R1, R2 ;
```

updates the x and w components of R0 with sum of R1 and R2.

4.3 The Floating-Point Core

The floating-point core is a multi-threaded vector processor operating on quad-float data. Vertex data is read from the input buffers and transformed into the output buffers (OB). The latency of the vector and special function units are equal and multiple vertex threads are used to hide this latency.

The SIMD Vector Unit is responsible for the MOV, MUL, ADD, MAD, DP3, DP4, DST, MIN, MAX, SLT, and SGE operations. The Special Function Unit is responsible for the RCP, RSQ, LOG, EXP, and LIT operations.

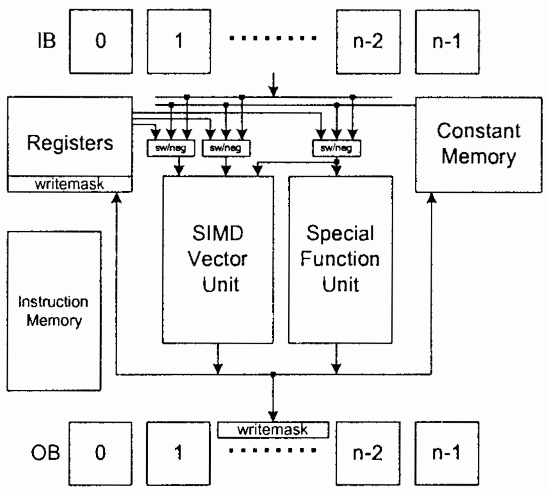

Figure 5: Floating Point Core

The Vector Unit floating-point precision is approximately IEEE. There is no support for de-normalized numbers or exceptions, and rounding is always towards negative infinity. The hardware outputs 0.0 for a multiply with any source of 0.0, including 0.0*infinity and 0.0**NaN*. The Special Function Unit calculates the RCP and RSQ functions to within about 1.5 bits of IEEE precision using two-pass Newton-Raphson iteration from a seed table. While lighting may suffice with a lower precision RSQ, texture and position evaluation can require much higher precision. It was not felt necessary to provide a low-precision RSQ option.

The hardware accepts one instruction per clock and fully implements all instruction set input/output options with no performance penalty. All input vectors are available with no latency.

perspective, programmability has been introduced in the geometry stage. A vertex shader (see Section 6.5) is a small program that can be sent down to the geometry stage. This program is then executed for each vertex sent to it. Each vertex shader unit can be implemented in custom hardware as a limited form of a CPU.

The Rasterizer Stage

The rasterizer must be implemented on custom chip(s) for the sake of speed. The first step in rasterization is *triangle setup*, which computes various deltas and slopes from the vertex information. Such computation is needed in order to rasterize the primitive and do interpolation. Color and depth are interpolated over the primitive. If there is a texture associated with the primitive, then texture coordinates are interpolated and the texture applied. Some hardware supports interpolating two colors, allowing a separate specular color [600] to avoid the weakening of this component's effect when texturing. Interpolation is not always linear; see Section 15.2.

Primitive setup and interpolation produces a fragment, which is simply data for a pixel covered by a primitive, i.e., its depth, alpha value, color, etc. The rasterizer in modern hardware is also programmable (see Section 6.6). With this data, fragment processing is performed. These tests include, in order of execution:

- Texture interpolation, which interpolates texture coordinates in perspective, and fetches the filtered texels.
- Color interpolation, which interpolates one or more colors and sums them.
- Fog, which can blend a fragment with a fog color.
- The alpha test; If alpha is 0, the fragment is not visible at all.
- The stencil test, which may mask the fragment depending on the contents of the stencil buffer.
- The depth test, which determines whether the fragment is visible by comparing its depth to the stored depth value.
- Alpha blending, which can blend the fragment's data with the original pixel's data.
- Dithering, to make the color look better with a 15- or 16-bit display mode.

Note that some rasterizer chips support only triangles, and do not sup port points and lines directly—these primitives are rendered by drawin rectangles representing them. Also, rasterizers use subpixel addressing t help avoid holes and overlaps caused by T-vertices (see Section 11.2.2) [474] This technique also helps avoid having objects poke through one another as discussed in Section 15.1.3. Subpixel addressing also helps eliminat the shifting and snapping of objects and textures as the viewer moves through a scene. This effect can be seen on older hardware, and early three-dimensional game consoles such as the Playstation and Nintendo 64.

Texture Caching and Prefetching

The performance increase in terms of pure computations has grown exponentially for many years. However, the latency and bandwidth of memory accesses have not increased nearly as fast. In addition, the trend is to use more and more textures per primitive. In fact, reading from texture memory is often the main consumer of bandwidth [12]. Therefore, when reading out texels from memory, care must be taken to reduce the used bandwidth and to hide latency. To save bandwidth, most architectures use caches at various places in the pipeline, and to hide latency, a technique called prefetching is often used. These two techniques combined will be described here in a texturing system.

Caching is implemented with a small on-chip memory, where the result of recent texture reads are stored and access is very fast. Texture caching was first investigated by Hakura and Gupta [313]. If neighboring pixels need to access the same or closely located texels, then it is likely to find these in the cache. This is what is done for standard CPUs as well. However, reading texels into the cache takes time, and most often entire cache blocks (e.g., 32 bytes) are read in at once. So, if a texel is not in the cache, it may take relatively long before it can be found there. Therefore, caching is often combined with prefetching, with excellent results.

Igehy et al. [381] suggest the architecture illustrated in Figure 15.10. The basic function is as follows. The rasterizer unit produces fragments that should be textured. The part of the fragment that is not related to texturing is sent to the fragment FIFO (first-in, first out) for use later. The fragment also includes a set of addresses of texels needed to filter that fragment, and these are sent to a cache tag unit. This checks whether the texels with those addresses are in the cache. If a texel is not in the cache, then the cache tags are updated with a cache address of the wanted texel, and the cache address sent to the fragment FIFO. However, at this point, the texel is not in the cache, so a request must also be sent to the request FIFO, which sends these requests in order to the texture memory. If the

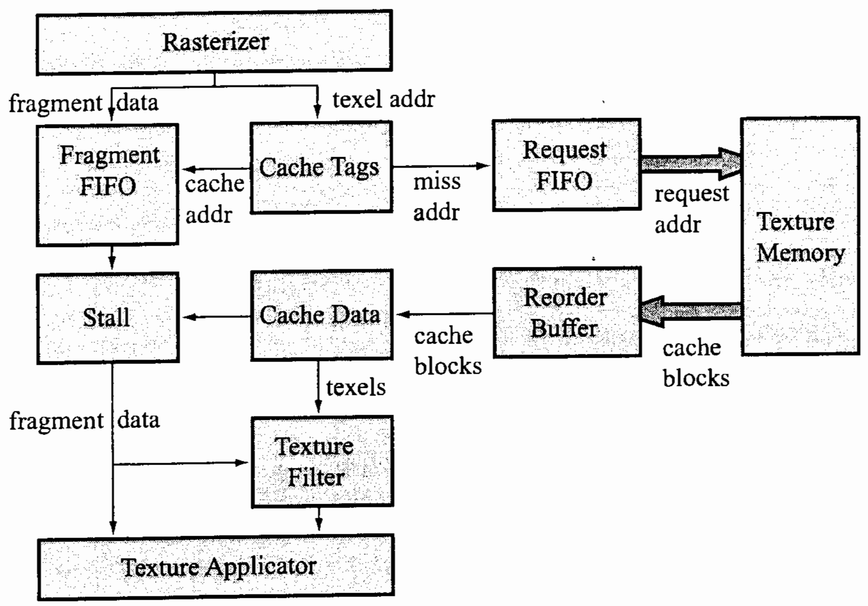

Figure 15.10. A texture prefetching architecture. *(Illustration after Igehy et al. [381].)*

texel is in the cache, its cache address is sent to the fragment FIFO. Cache blocks arrive from the texture memory to the reorder buffer, whose task is to order the blocks so that they arrive to the cache in the order that they were requested. The fragment FIFO holds a list of fragments, with a set of cache addresses containing the texels needed to do texture filtering for that fragment. The fragment on its way out from the fragment FIFO stalls until all texels are available in the texture cache. When the stall is resolved, the texels are sent to a unit that computes the filtered texel. Finally, this is forwarded to the unit that applies it to the fragment.

Simulation shows that such a system could attain 97 percent of the performance of a system with no latency at all.

More about this architecture and texture memory organization can be found in the paper by Igehy et al. [381], where they show that a certain blocking scheme for storing the textures reduces miss rate in the cache. Several researchers have also shown that the texture cache miss rate is reduced by doing rasterization in a tiled fashion, i.e., one rasterizes all the pixels inside, say, an 8×8 tile at a time [315, 381, 533]. In another study, Igehy et al. [382] investigate parallel texture caching schemes for use with parallel rasterization.

15.3.2 Memory Architecture

Here, we will mention a few different architectures that have been used for memory layout. The SGI O2 and the Xbox (Section 15.3.6) use a *Unified Memory Architecture*(UMA), which means that the graphics accelerator can use any part of the host memory for textures and different kinds of buffers [420]. An example of UMA is shown in Figure 15.12 on page 697. As can be seen both the CPU and the graphics accelerator uses the same memory, and thus also the same bus. A somewhat less unified layout is to have dedicated memory on the graphics card, which can then be used for textures and buffers in any way desired, but cannot be used directly by the CPU. This is the approach taken by the KYRO architecture (Section 15.3.8), which uses a local memory for scene data and for textures. The InfiniteReality [568] (see Section 15.3.7) also uses a nonunified memory architecture, and due to its scalable nature, it has to replicate the texture memory across each rasterizer.

15.3.3 Port and Bus Bandwidth

A port is a channel for sending data between two devices, and a bus is a shared channel for sending data among more than two devices. Bandwidth is the term used to describe throughput of data over the port or bus, and is measured in bytes per second, b/s. Ports and buses are important in computer graphics architecture because, simply put, they "glue" together different building blocks. Also important is that bandwidth is a scarce resource, and so a careful design and analysis must be done before building a graphics system. An example of a port is one that connects the CPU with the graphics accelerator, such as the Accelerated Graphics Port (AGP) used in PCs. 2Since ports and buses both provide data transfer capabilities, ports are often referred to as buses, a convention we will follow here.

When it comes to sending data to the graphics hardware over the bus, there are two methods—*push* and *pull*. The pull method works by writing data to the system memory. The graphics hardware can then *pull* data from those memory locations during an entire frame. This is also called *Direct Memory Access* (DMA), since the graphics hardware is allowed to directly access the memory and thus bypass the CPU. This is possible because a certain region of memory is locked by the graphics hardware, meaning that the CPU cannot use that region until it is unlocked. In general, DMA allows the GPU to work faster, and frees CPU time. The AGP can be used to pull data in this way, because it has a direct path from the graphics

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

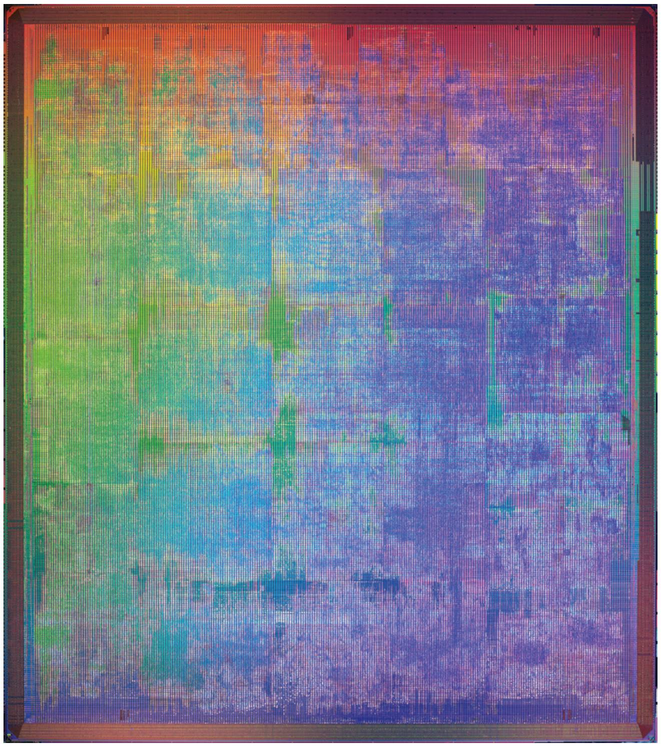

Figure 30-1. The GeForce 6800 Microprocessor

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.

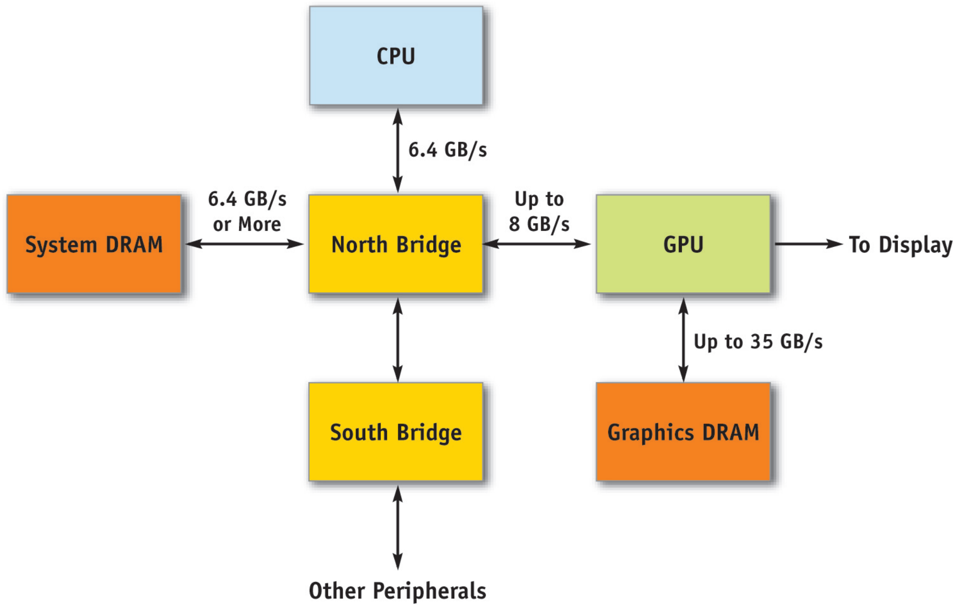

Figure 30-2. The Overall System Architecture of a PC

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.

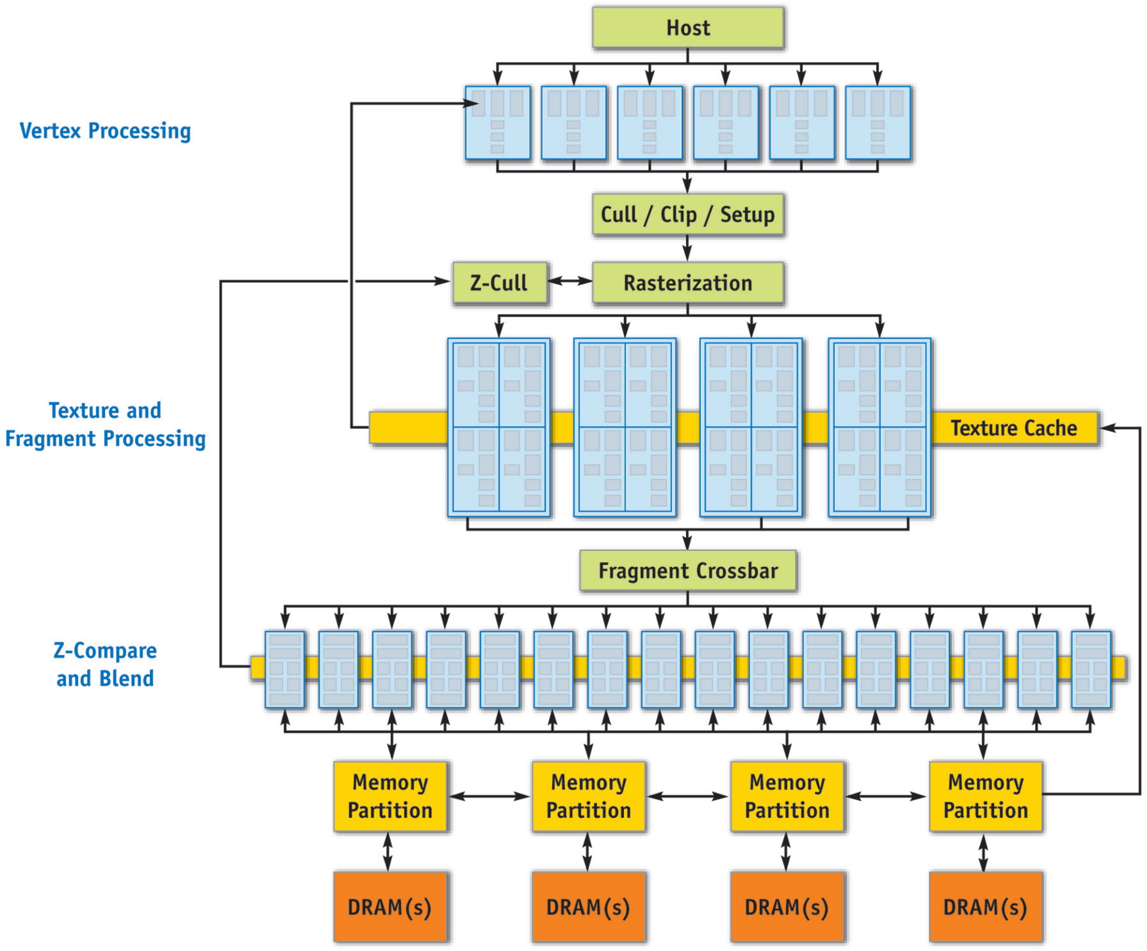

Figure 30-3. A Block Diagram of the GeForce 6 Series Architecture

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.

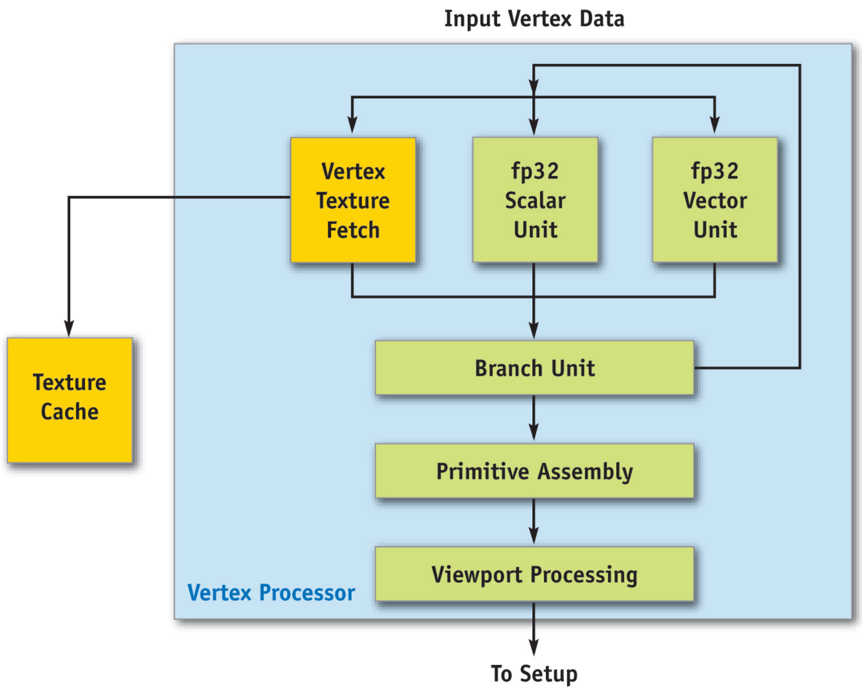

Figure 30-4. The GeForce 6 Series Vertex Processor

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.

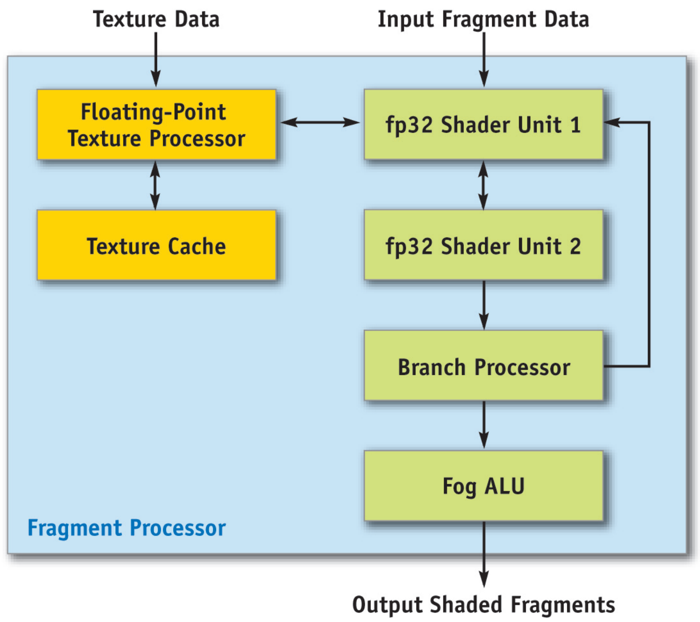

Figure 30-5. The GeForce 6 Series Fragment Processor and Texel Pipeline

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.)

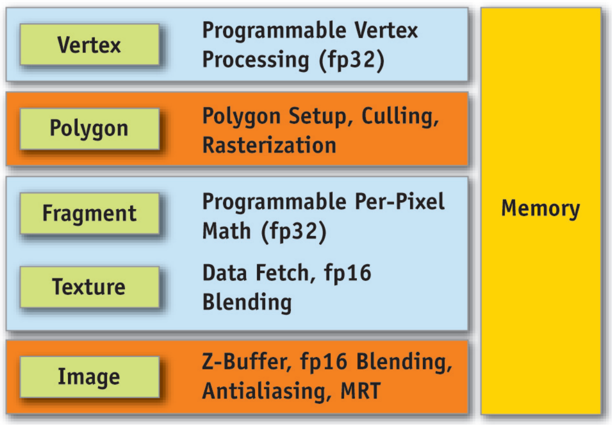
Vertex
Programmable Vertex Processing (fp32)
Polygon
Polygon Setup, Culling, Rasterization
Fragment
Programmable Per-Pixel Math (fp32)
Texture
Data Fetch, fp16 Blending
Image
Z-Buffer, fp16 Blending, Antialiasing, MRT
Memory

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

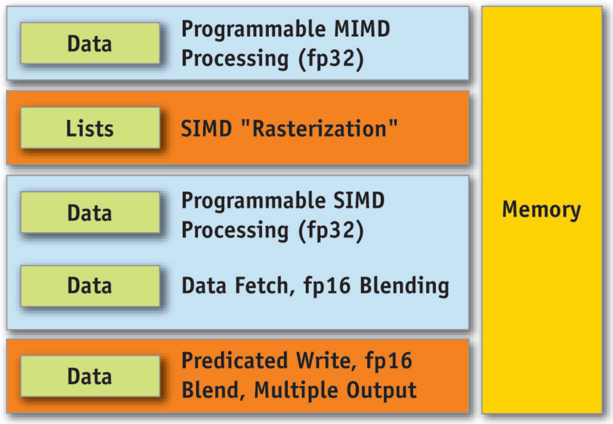
Data
Programmable MIMD Processing (fp32)
Lists
SIMD "Rasterization"
Data
Programmable SIMD Processing (fp32)
Data
Data Fetch, fp16 Blending
Data
Predicated Write, fp16 Blend, Multiple Output
Memory

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.

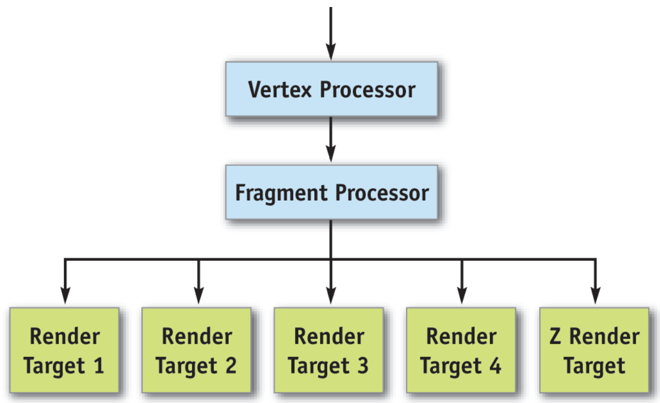

Figure 30-8. How MRTs Work
MRTs make it possible for a fragment program to return four four-wide color values plus a depth value.

- **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.

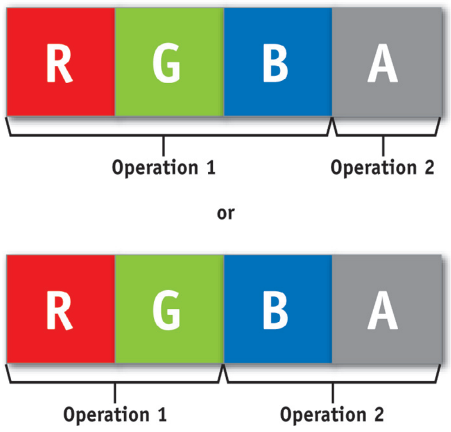

Figure 30-9. How Coissue Works
Two separate operations can concurrently execute on different parts of a four-wide register.

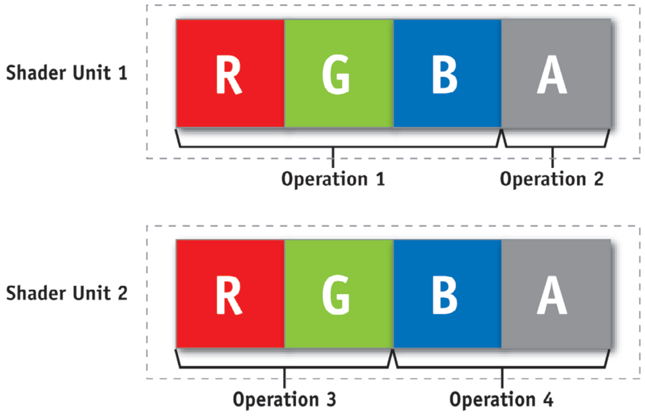

Figure 30-10. How Dual Issue Works
Independent instructions can be executed on independent units in the computational pipeline.

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.

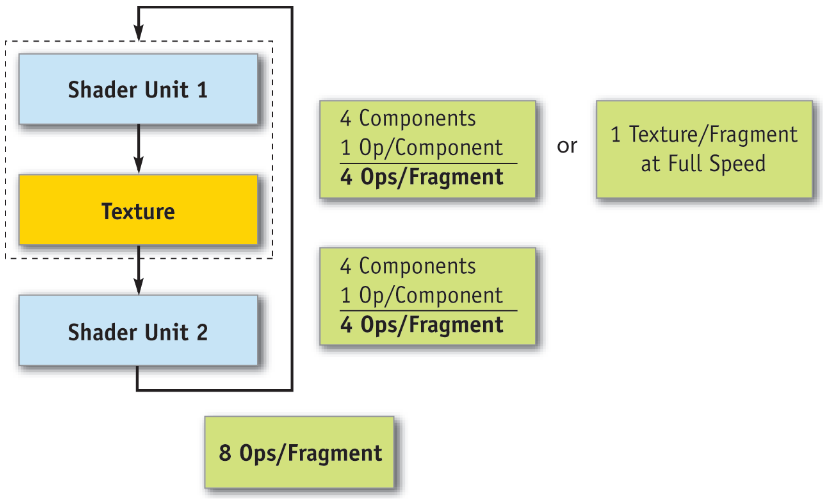

Figure 30-11. Shader Units and Capabilities in the Fragment Processor

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	✗	✓	✗