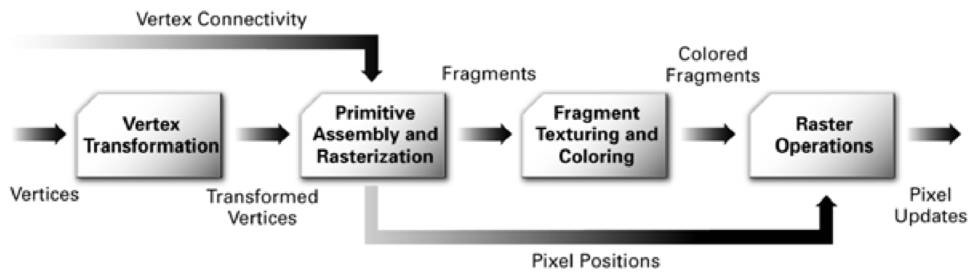

Figure 1-3. The Graphics Hardware Pipeline

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.

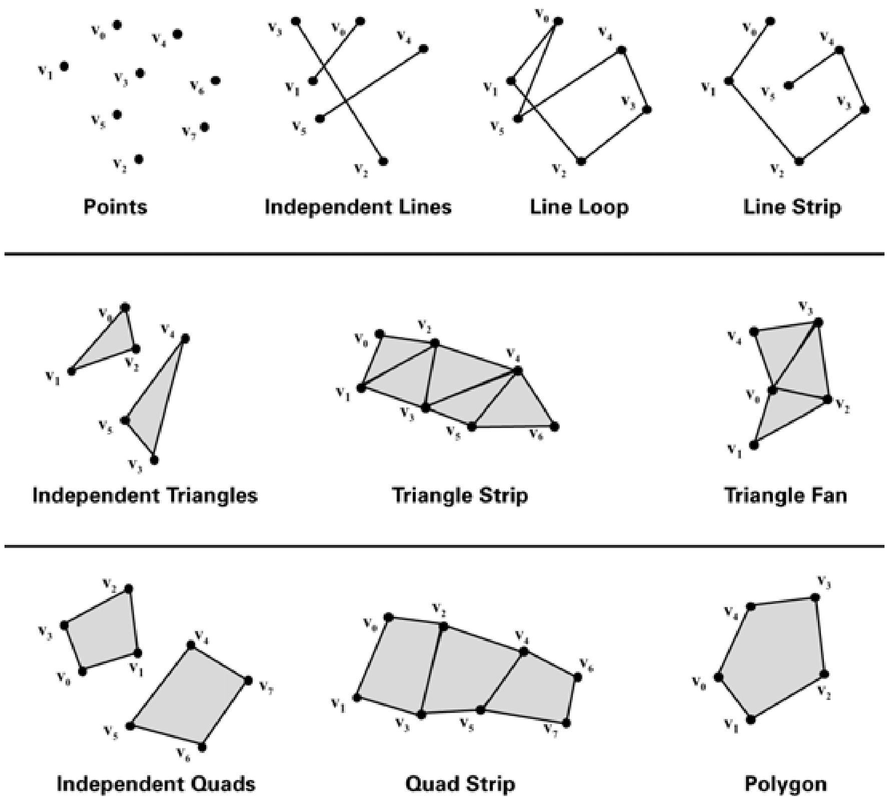

Figure 1-4. Types of Geometric Primitives

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

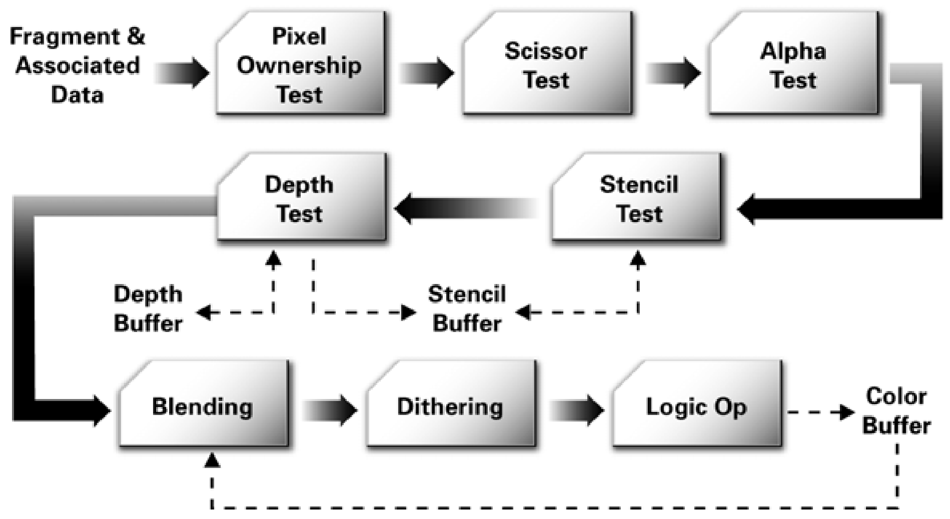

Figure 1-5. Standard OpenGL and Direct3D Raster Operations

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-

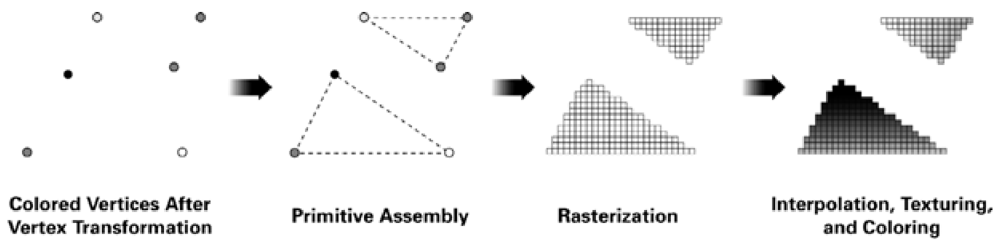
Colored Vertices After Vertex Transformation
Primitive Assembly
Rasterization
Interpolation, Texturing, and Coloring

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*

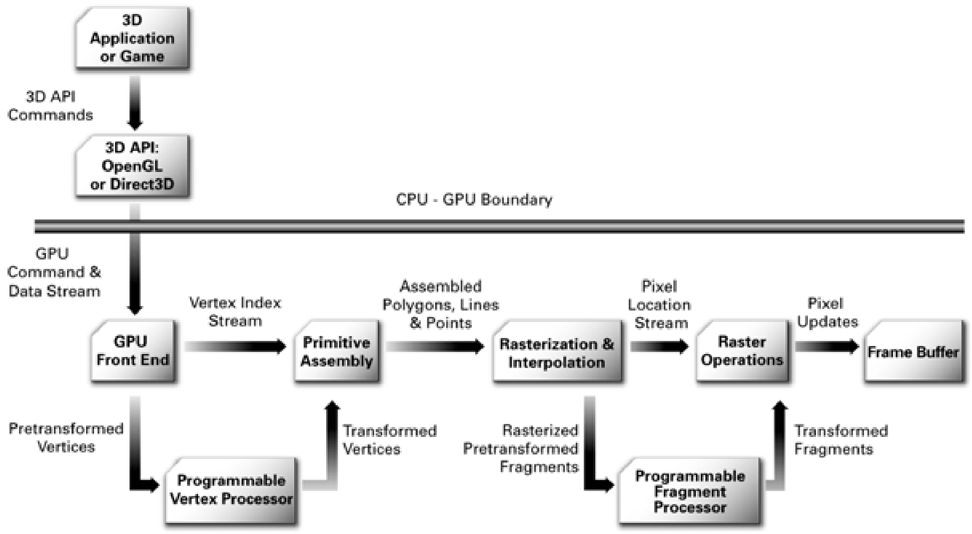

Figure 1-7. The Programmable Graphics Pipeline

vertex processor is the hardware unit that runs your Cg[1] 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

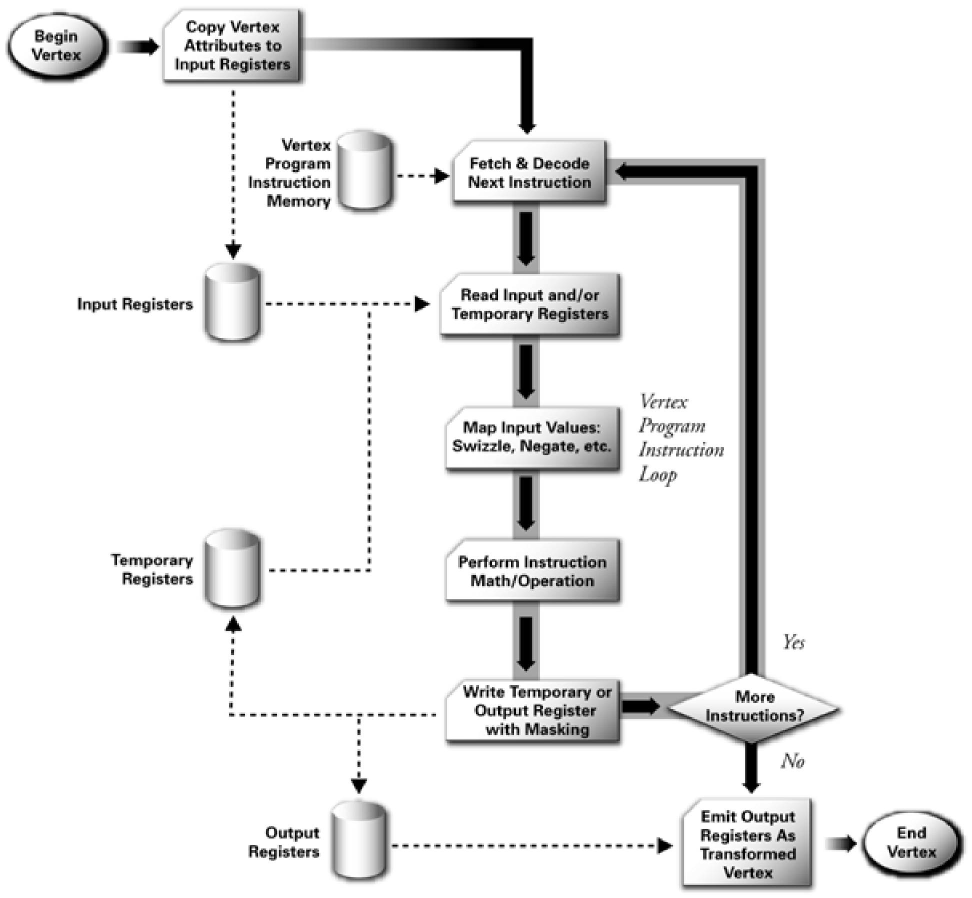
Begin Vertex
Copy Vertex Attributes to Input Registers
Vertex Program Instruction Memory
Fetch & Decode Next Instruction
Input Registers
Read Input and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Vertex Program Instruction Loop
Temporary Registers
Perform Instruction Math/Operation
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

1: Cg is Nvidia's GPU programming language. Cg is short for "C for graphics. In class we will talk about OpenGL's GPU programming language called GLSL. Cg & GlSL overlap significantly 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.

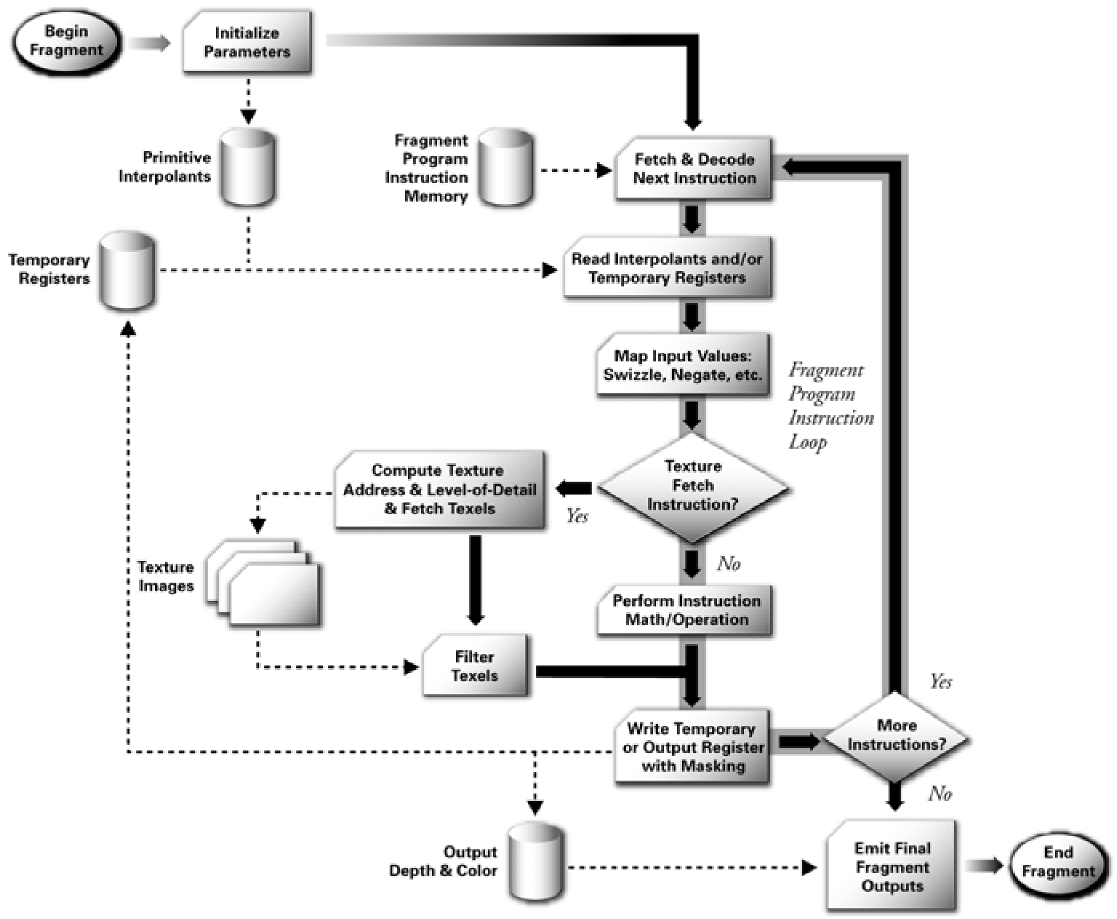
Begin Fragment
Initialize Parameters
Primitive Interpolants
Fragment Program Instruction Memory
Fetch & Decode Next Instruction
Temporary Registers
Read Interpolants and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Fragment Program Instruction Loop
Compute Texture Address & Level-of-Detail & 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.