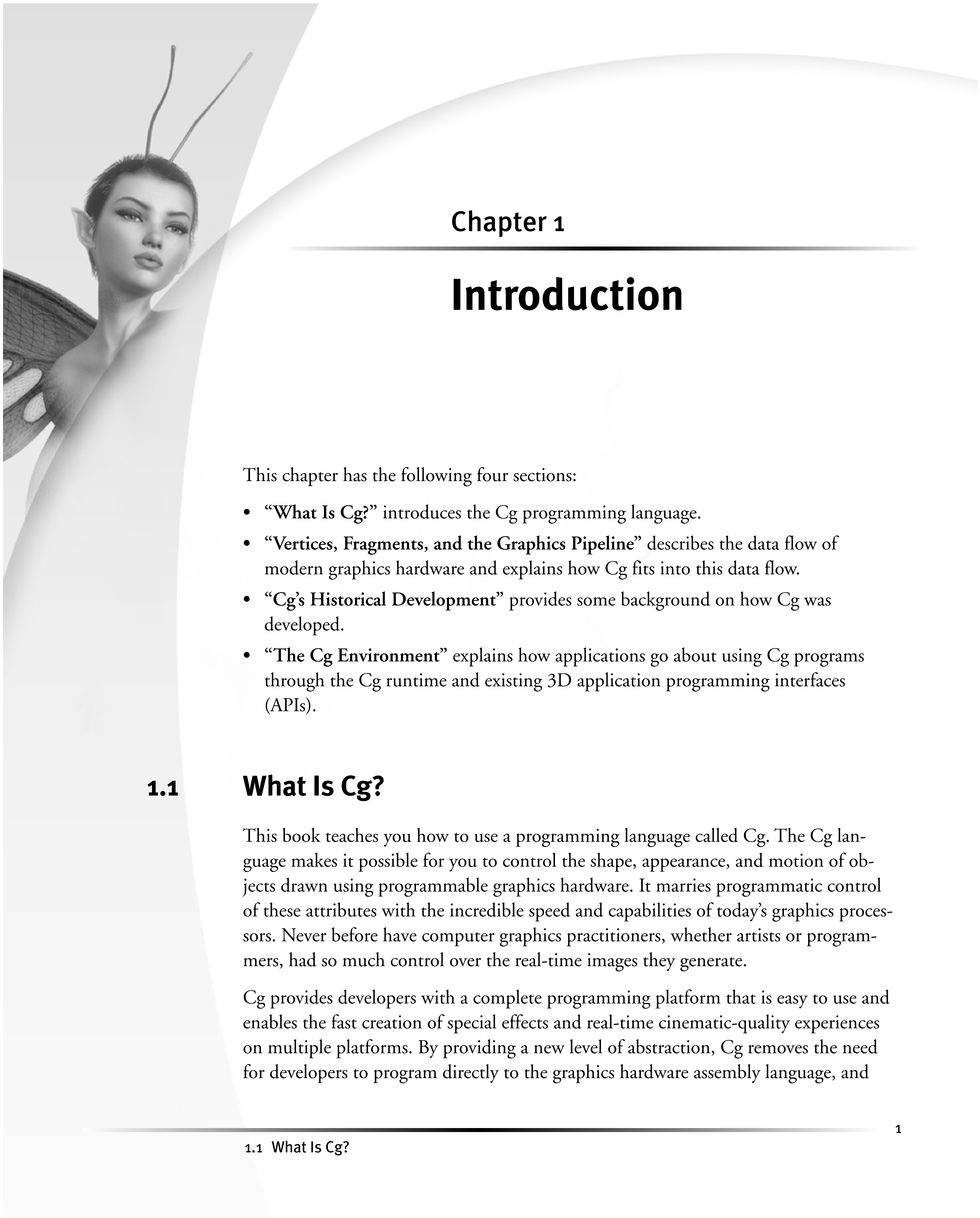

Chapter 1

# Introduction

This chapter has the following four sections:

- "**What Is Cg?**" introduces the Cg programming language.
- "**Vertices, Fragments, and the Graphics Pipeline**" describes the data flow of modern graphics hardware and explains how Cg fits into this data flow.
- "**Cg's Historical Development**" provides some background on how Cg was developed.
- "**The Cg Environment**" explains how applications go about using Cg programs through the Cg runtime and existing 3D application programming interfaces (APIs).

## 1.1 What Is Cg?

This book teaches you how to use a programming language called Cg. The Cg language makes it possible for you to control the shape, appearance, and motion of objects drawn using programmable graphics hardware. It marries programmatic control of these attributes with the incredible speed and capabilities of today's graphics processors. Never before have computer graphics practitioners, whether artists or programmers, had so much control over the real-time images they generate.

Cg provides developers with a complete programming platform that is easy to use and enables the fast creation of special effects and real-time cinematic-quality experiences on multiple platforms. By providing a new level of abstraction, Cg removes the need for developers to program directly to the graphics hardware assembly language, and thereby more easily target OpenGL, DirectX, Windows, Linux, Macintosh OS X, and console platforms such as the Xbox. Cg was developed in close collaboration with Microsoft Corporation and is compatible with both the OpenGL API and Microsoft's High-Level Shading Language (HLSL) for DirectX 9.0.

Cg stands for "C for graphics." The C programming language is a popular, general-purpose language invented in the 1970s. Because of its popularity and clean design, C provided the basis for several subsequent programming languages. For example, C++ and Java base their syntax and structure largely on C. The Cg language bases itself on C as well. If you are familiar with C or one of the many languages derived from C, then Cg will be easy to learn.

On the other hand, if you are not familiar with C or even programming languages in general but you enjoy computer graphics and want to learn something new, read on anyway. Cg programs tend to be short and understandable.

Much of this chapter is background that provides valuable context for understanding Cg and using it effectively. On the other hand, you may find Cg is easier to learn by doing. Feel free to skip to Chapter 2 at any time if you feel more comfortable just diving into the tutorial.

### 1.1.1 A Language for Programming Graphics Hardware

Cg is different from C, C++, and Java because it is very specialized. No one will ever write a spreadsheet or word processor in Cg. Instead, Cg targets the ability to programmatically control the shape, appearance, and motion of objects rendered using graphics hardware. Broadly, this type of language is called a *shading language*. However, Cg can do more than just shading. For example, Cg programs can perform physical simulation, compositing, and other nonshading tasks.

Think of a Cg program as a detailed recipe for how to render an object by using programmable graphics hardware. For example, you can write a Cg program to make a surface appear bumpy or to animate a virtual character. Later, in Section 1.3, you will learn more about the history of shading languages and where Cg fits into this history.

### 1.1.2 Cg's Data-Flow Model

In addition to being specialized for graphics, Cg and other shading languages are different from conventional programming languages because they are based on a data-flow computational model. In such a model, computation occurs in response to data that flows through a sequence of processing steps.

Cg programs operate on vertices and fragments (think "pixels" for now if you do not know what a fragment is) that are processed when rendering an image. Think of a Cg program as a black box into which vertices or fragments flow on one side, are somehow transformed, and then flow out on the other side. However, the box is not really a black box because you get to determine, by means of the Cg programs you write, exactly what happens inside.

Every time a vertex is processed or the rasterizer generates a fragment while rendering a 3D scene, your corresponding vertex or fragment Cg program executes. Section 1.3 explains Cg's data-flow model further.

Most recent personal computers—and all recent game consoles—contain a graphics processing unit (GPU) that is dedicated to graphics tasks such as transforming and rasterizing 3D models. Your Cg programs actually execute within the GPU of your computer.

### 1.1.3 GPU Specialization and CPU Generalization

Whether or not a personal computer or game console has a GPU, there must be a CPU that runs the operating system and application programs. CPUs are, by design, general purpose. CPUs execute applications (for example, word processors and accounting packages) written in general-purpose languages, such as C++ or Java.

Because of the GPU's specialized design, it is much faster at graphics tasks, such as rendering 3D scenes, than a general-purpose CPU would be. New GPUs process tens of millions of vertices per second and rasterize hundreds of millions or even billions of fragments per second. Future GPUs will be even speedier. This is overwhelmingly faster than the rate at which a CPU could process a similar number of vertices and fragments. However, the GPU cannot execute the same arbitrary, general-purpose programs that a CPU can.

The specialized, high-performance nature of the GPU is why Cg exists. General-purpose programming languages are too open-ended for the specialized task of processing vertices and fragments. In contrast, the Cg language is fully dedicated to this task. Cg also provides an abstract execution model that matches the GPU's execution model. You will learn about the unique execution model of GPUs in Section 1.2.

## 1.1.4 The Performance Rationale for Cg

To sustain the illusion of interactivity, a 3D application needs to maintain an animation rate of 15 or more images per second. Generally, we consider 60 or more frames per second to be "real time," the rate at which interaction with applications appears to occur instantaneously. The computer's display may have a million or more pixels that require redrawing. For 3D scenes, the GPU typically processes every pixel on the screen many times to account for how objects occlude each other, or to improve the appearance of each pixel. This means that real-time 3D applications can require hundreds of millions of pixel updates per second. Along with the required pixel processing, 3D models are composed of vertices that must be transformed properly before they are assembled into polygons, lines, and points that will be rasterized into pixels. This can require transforming tens of millions of vertices per second.

Moreover, this graphical processing happens in addition to the considerable amount of effort required of the CPU to update the animation for each new image. The reality is that we need both the CPU and the GPU's specialized graphics-oriented capabilities. Both are required to render scenes at the interactive rates and quality standards that users of 3D applications and games demand. This means a developer can write a 3D application or game in C++ and then use Cg to make the most of the GPU's additional graphics horsepower.

## 1.1.5 Coexistence with Conventional Languages

In no way does Cg replace any existing general-purpose languages. Cg is an auxiliary language, designed specifically for GPUs. Programs written for the CPU in conventional languages such as C or C++ can use the Cg runtime (described in Section 1.4.2) to load Cg programs for GPUs to execute. The Cg runtime is a standard set of subroutines used to load, compile, manipulate, and configure Cg programs for execution by the GPU. Applications supply Cg programs to instruct GPUs on how to accomplish the programmable rendering effects that would not otherwise be possible on a CPU at the rendering rates a GPU is capable of achieving.

Cg enables a specialized style of parallel processing. While your CPU executes a conventional application, that application also orchestrates the parallel processing of vertices and fragments on the GPU, by programs written in Cg.

If a real-time shading language is such a good idea, why didn't someone invent Cg sooner? The answer has to do with the evolution of computer graphics hardware. Prior to 2001, most computer graphics hardware—certainly the kind of inexpensive graphics hardware in PCs and game consoles—was hard-wired to the specific tasks of vertex and fragment processing. By "hard-wired," we mean that the algorithms were fixed within the hardware, as opposed to being programmable in a way that is accessible to graphics applications. Even though these hard-wired graphics algorithms could be configured by graphics applications in a variety of ways, the applications could not reprogram the hardware to do tasks unanticipated by the designers of the hardware. Fortunately, this situation has changed.

Graphics hardware design has advanced, and vertex and fragment processing units in recent GPUs are truly programmable. Before the advent of programmable graphics hardware, there was no point in providing a programming language for it. Now that such hardware is available, there is a clear need to make it easier to program this hardware. Cg makes it much easier to program GPUs in the same manner that C made it much easier to program CPUs.

Before Cg existed, addressing the programmable capabilities of the GPU was possible only through low-level assembly language. The cryptic instruction syntax and manual hardware register manipulation required by assembly languages—such as DirectX 8 vertex and pixel shaders and some OpenGL extensions—made it a painful task for most developers. As GPU technology made longer and more complex assembly language programs possible, the need for a high-level language became clear. The extensive low-level programming that had been required to achieve optimal performance could now be delegated to a compiler, which optimizes the code output and handles tedious instruction scheduling. Figure 1-1 is a small portion of a complex assembly language fragment program used to represent skin. Clearly, it is hard to comprehend, particularly with the specific references to hardware registers.

In contrast, well-commented Cg code is more portable, more legible, easier to debug, and easier to reuse. Cg gives you the advantages of a high-level language such as C while delivering the performance of low-level assembly code.

### 1.1.6 Other Aspects of Cg

Cg is a language for programming "in the small." That makes it much simpler than a modern general-purpose language such as C++. Because Cg specializes in transforming vertices and fragments, it does not currently include many of the complex features required for massive software engineering tasks. Unlike C++ and Java, Cg does not support classes and other features used in object-oriented programming. Current Cg

```
. . .
DEFINE LUMINANCE = {0.299, 0.587, 0.114, 0.0};
TEX  H0, f[TEX0], TEX4, 2D;
TEX  H1, f[TEX2], TEX5, CUBE;
DP3X H1.xyz, H1, LUMINANCE;
MULX H0.w, H0.w, LUMINANCE.w;
MULX H1.w, H1.x, H1.x;
MOVH H2, f[TEX3].wxyz;
MULX H1.w, H1.x, H1.w;
DP3X H0.xyz, H2.xzyw, H0;
MULX H0.xyz, H0, H1.w;
TEX H1, f[TEX0], TEX1, 2D;
TEX H3, f[TEX0], TEX3, 2D;
MULX H0.xyz, H0, H3;
MADX H1.w, H1.w, 0.5, 0.5;
MULX H1.xyz, H1, {0.15, 0.15, 1.0, 0.0};
MOVX H0.w, H1.w;
TEX H1, H1, TEX7, CUBE;
TEX H3, f[TEX3], TEX2, 1D;
MULX H3.w, H0.w, H2.w;
MULX H3.xyz, H3, H3.w;
. . .
```

**Figure 1-1.** A Snippet of Assembly Language Code

implementations do not provide pointers or even memory allocation (though future implementations may, and keywords are appropriately reserved). Cg has absolutely no support for file input/output operations. By and large, these restrictions are not permanent limitations in the language, but rather are indicative of the capabilities of today's highest performance GPUs. As technology advances to permit more general programmability on the GPU, you can expect Cg to grow appropriately. Because Cg is closely based on C, future updates to Cg are likely to adopt language features from C and C++.

Cg provides arrays and structures. It has all the flow-control constructs of a modern language: loops, conditionals, and function calls.

Cg natively supports vectors and matrices because these data types and related math operations are fundamental to graphics and most graphics hardware directly supports vector data types. Cg has a library of functions, called the Standard Library, that is well suited for the kind of operations required for graphics. For example, the Cg Standard Library includes a **`reflect`** function for computing reflection vectors.

Cg programs execute in relative isolation. This means that the processing of a particular vertex or fragment has no effect on other vertices or fragments processed at the same time. There are no side effects to the execution of a Cg program. This lack of interdependency among vertices and fragments makes Cg programs extremely well suited for hardware execution by highly pipelined and parallel hardware.

### 1.1.7 The Limited Execution Environment of Cg Programs

When you write a program in a language designed for modern CPUs using a modern operating system, you expect that a more-or-less arbitrary program, as long as it is correct, will compile and execute properly. This is because CPUs, by design, execute general-purpose programs for which the overall system has more than sufficient resources.

However, GPUs are specialized rather than general-purpose, and the feature set of GPUs is still evolving. Not everything you can write in Cg can be compiled to execute on a given GPU. Cg includes the concept of hardware "profiles," one of which you specify when you compile a Cg program. Each profile corresponds to a particular combination of GPU architecture and graphics API. Your program not only must be correct, but it also must limit itself to the restrictions imposed by the particular profile used to compile your Cg program. For example, a given fragment profile may limit you to no more than four texture accesses per fragment.

As GPUs evolve, additional profiles will be supported by Cg that correspond to more capable GPU architectures. In the future, profiles will be less important as GPUs become more full-featured. But for now Cg programmers will need to limit programs to ensure that they can compile and execute on existing GPUs. In general, future profiles will be supersets of current profiles, so that programs written for today's profiles will compile without change using future profiles.

This situation may sound limiting, but in practice the Cg programs shown in this book work on tens of millions of GPUs and produce compelling rendering effects. Another reason for limiting program size and scope is that the smaller and more efficient your Cg programs are, the faster they will run. Real-time graphics is often about balancing increased scene complexity, animation rates, and improved shading. So it's always good to maximize rendering efficiency through judicious Cg programming.

Keep in mind that the restrictions imposed by profiles are really limitations of current GPUs, not Cg. The Cg language is powerful enough to express shading techniques that are not yet possible with all GPUs. With time, GPU functionality will evolve far enough that Cg profiles will be able to run amazingly complex Cg programs. Cg is a language for both current and future GPUs.

## 1.2 Vertices, Fragments, and the Graphics Pipeline

To put Cg into its proper context, you need to understand how GPUs render images. This section explains how graphics hardware is evolving and then explores the modern graphics hardware-rendering pipeline.

### 1.2.1 The Evolution of Computer Graphics Hardware

Computer graphics hardware is advancing at incredible rates. Three forces are driving this rapid pace of innovation, as shown in Figure 1-2. First, the semiconductor industry has committed itself to doubling the number of transistors (the basic unit of computer hardware) that fit on a microchip every 18 months. This constant redoubling of

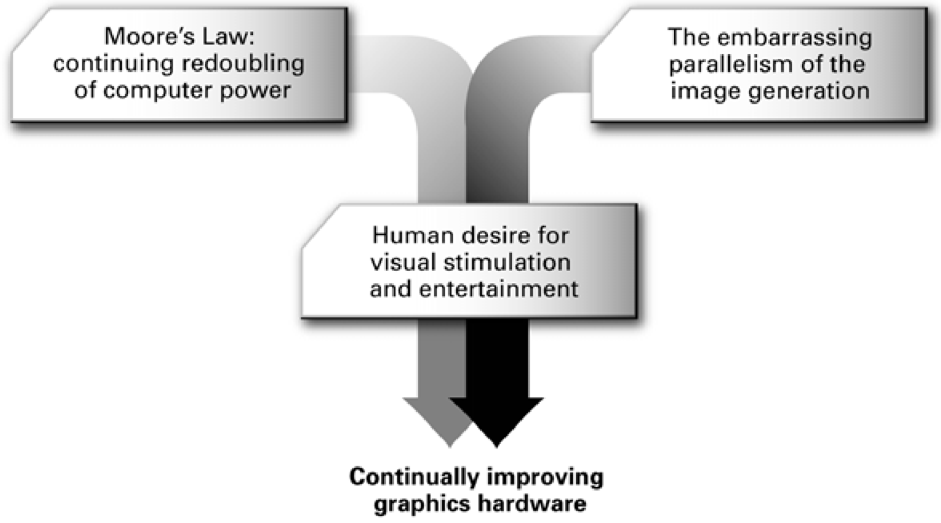
Moore's Law: continuing redoubling of computer power
The embarrassing parallelism of the image generation
Human desire for visual stimulation and entertainment
Continually improving graphics hardware

**Figure 1-2.** Forces Driving Graphics Hardware Innovation

computer power, historically known as Moore's Law, means cheaper and faster computer hardware, and is the norm for our age.

The second force is the vast amount of computation required to simulate the world around us. Our eyes consume and our brains comprehend images of our 3D world at an astounding rate and with startling acuity. We are unlikely ever to reach a point where computer graphics becomes a substitute for reality. Reality is just too real. Undaunted, computer graphics practitioners continue to rise to the challenge. Fortunately, generating images is an embarrassingly parallel problem. What we mean by "embarrassingly parallel" is that graphics hardware designers can repeatedly split up the problem of creating realistic images into more chunks of work that are smaller and easier to tackle. Then hardware engineers can arrange, in parallel, the ever-greater number of transistors available to execute all these various chunks of work.

Our third force is the sustained desire we all have to be stimulated and entertained visually. This is the force that "connects" the source of our continued redoubling of computer hardware resources to the task of approximating visual reality ever more realistically than before.

As Figure 1-2 illustrates, these insights let us confidently predict that computer graphics hardware is going to get much faster. These innovations whet our collective appetite for more interactive and compelling 3D experiences. Satisfying this demand is what motivated the development of the Cg language.

### 1.2.2 Four Generations of Computer Graphics Hardware

In the mid-1990s, the world's fastest graphics hardware consisted of multiple chips that worked together to render images and display them to a screen. The most complex computer graphics systems consisted of dozens of chips spread over several boards. As time progressed and semiconductor technology improved, hardware engineers incorporated the functionality of complicated multichip designs into a single graphics chip. This development resulted in tremendous economies of integration and scale.

You may be surprised to learn that the GPU now exceeds the CPU in the number of transistors present in each microchip. Transistor count is a rough measure of how much computer hardware is devoted to a microchip. For example, Intel packed its 2.4 GHz Pentium 4 with 55 million transistors; NVIDIA used over 125 million transistors in the original GeForce FX GPU.

NVIDIA introduced the term “GPU” in the late 1990s when the legacy term “VGA controller” was no longer an accurate description of the graphics hardware in a PC. IBM had introduced Video Graphics Array (VGA) hardware in 1987. At that time, the VGA controller was what we now call a “dumb” frame buffer. This meant that the CPU was responsible for updating all the pixels. Today the CPU rarely manipulates pixels directly. Instead, graphics hardware designers build the “smarts” of pixel updates into the GPU.

Industry observers have identified four generations of GPU evolution so far. Each generation delivers better performance and evolving programmability of the GPU feature set. Each generation also influences and incorporates the functionality of the two major 3D programming interfaces, OpenGL and DirectX. OpenGL is an open standard for 3D programming for Windows, Linux, UNIX, and Macintosh computers. DirectX is an evolving set of Microsoft multimedia programming interfaces, including Direct3D for 3D programming.

## Pre-GPU Graphics Acceleration

Prior to the introduction of GPUs, companies such as Silicon Graphics (SGI) and Evans & Sutherland designed specialized and expensive graphics hardware. The graphics systems developed by these companies introduced many of the concepts, such as vertex transformation and texture mapping, that we take for granted today. These systems were very important to the historical development of computer graphics, but because they were so expensive, they did not achieve the mass-market success of single-chip GPUs designed for PCs and video game consoles. Today, GPUs are far more powerful and much cheaper than any prior systems.

## First-Generation GPUs

The first generation of GPUs (up to 1998) includes NVIDIA’s TNT2, ATI’s Rage, and 3dfx’s Voodoo3. These GPUs are capable of rasterizing pre-transformed triangles and applying one or two textures. They also implement the DirectX 6 feature set. When running most 3D and 2D applications, these GPUs completely relieve the CPU from updating individual pixels. However, GPUs in this generation suffer from two clear limitations. First, they lack the ability to transform vertices of 3D objects; instead, vertex transformations occur in the CPU. Second, they have a quite limited set of math operations for combining textures to compute the color of rasterized pixels.

## Second-Generation GPUs

The second generation of GPUs (1999–2000) includes NVIDIA's GeForce 256 and GeForce2, ATI's Radeon 7500, and S3's Savage3D. These GPUs offload 3D vertex transformation and lighting (T&L) from the CPU. Fast vertex transformation was one of the key capabilities that differentiated high-end workstations from PCs prior to this generation. Both OpenGL and DirectX 7 support hardware vertex transformation. Although the set of math operations for combining textures and coloring pixels expanded in this generation to include cube map textures and signed math operations, the possibilities are still limited. Put another way, this generation is more configurable, but still not truly programmable.

## Third-Generation GPUs

The third generation of GPUs (2001) includes NVIDIA's GeForce3 and GeForce4 Ti, Microsoft's Xbox, and ATI's Radeon 8500. This generation provides vertex programmability rather than merely offering more configurability. Instead of supporting the conventional transformation and lighting modes specified by OpenGL and DirectX 7, these GPUs let the application specify a sequence of instructions for processing vertices. Considerably more pixel-level configurability is available, but these modes are not powerful enough to be considered truly programmable. Because these GPUs support vertex programmability but lack true pixel programmability, this generation is transitional. DirectX 8 and the multivendor **`ARB_vertex_program`** OpenGL extension expose vertex-level programmability to applications. DirectX 8 pixel shaders and various vendor-specific OpenGL extensions expose this generation's fragment-level configurability.

## Fourth-Generation GPUs

The fourth and current generation of GPUs (2002 and on) includes NVIDIA's GeForce FX family with the CineFX architecture and ATI's Radeon 9700. These GPUs provide both vertex-level and pixel-level programmability. This level of programmability opens up the possibility of offloading complex vertex transformation and pixel-shading operations from the CPU to the GPU. DirectX 9 and various OpenGL extensions expose the vertex-level and pixel-level programmability of these GPUs. This is the generation of GPUs where Cg gets really interesting. Table 1-1 lists selected NVIDIA GPUs representing these various GPU generations.

**Table 1-1.** Features and Performance Evolution of Selected NVIDIA GPUs, by Generation

| Generation | Year | Product Name | Process | Transistors | Antialiasing Fill Rate | Polygon Rate | Note |
|---|---|---|---|---|---|---|---|
| First | Late 1998 | RIVA TNT | 0.25 μ | 7 M | *50 M* | 6 M | 1 |
| First | Early 1999 | RIVA TNT2 | 0.22 μ | 9 M | *75 M* | 9 M | 2 |
| Second | Late 1999 | GeForce 256 | 0.22 μ | 23 M | *120 M* | 15 M | 3 |
| Second | Early 2000 | GeForce2 | 0.18 μ | 25 M | *200 M* | 25 M | 4 |
| Third | Early 2001 | GeForce3 | 0.15 μ | 57 M | 800 M | 30 M | 5 |
| Third | Early 2002 | GeForce4 Ti | 0.15 μ | 63 M | 1200 M | 60 M | 6 |
| Fourth | Early 2003 | GeForce FX | 0.13 μ | 125 M | 2000 M | 200 M | 7 |

**Notes**
1. Dual texture DirectX 6
2. AGP 4×
3. Fixed-function vertex hardware, register combiners, cube maps, DirectX 7
4. Performance, double data-rate (DDR) memory
5. Vertex programs, quad-texturing, texture shaders, DirectX 8
6. Performance, antialiasing
7. Massive vertex and fragment programmability, floating-point pixels, DirectX 9, AGP 8×

The table uses the following terms:

- ***Process***—the minimum feature size in microns (μ, millionths of a meter) for the semiconductor process used to fabricate each microchip
- ***Transistors***—an approximate measure, in millions (M), of the chips' design and manufacturing complexity
- ***Antialiasing fill rate***—a GPU's ability to fill pixels, measured in millions (M) of 32-bit RGBA pixels per second, assuming two-sample antialiasing; numbers in *italics* indicate fill rates that are de-rated because the hardware lacks true antialiased rendering
- ***Polygon rate***—a GPU's ability to draw triangles, measured in millions (M) of triangles per second

The notes highlight the most significant improvements in each design. Performance rates may not be comparable with designs from other hardware vendors.

Future GPUs will further generalize the programmable aspects of current GPUs, and Cg will make this additional programmability easy to use.

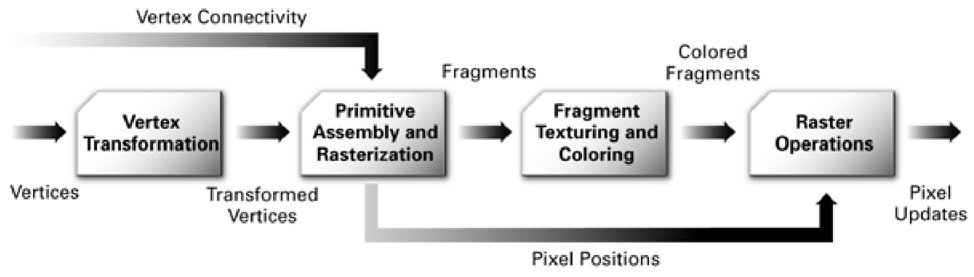

**Figure 1-3.** The Graphics Hardware Pipeline

## 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.

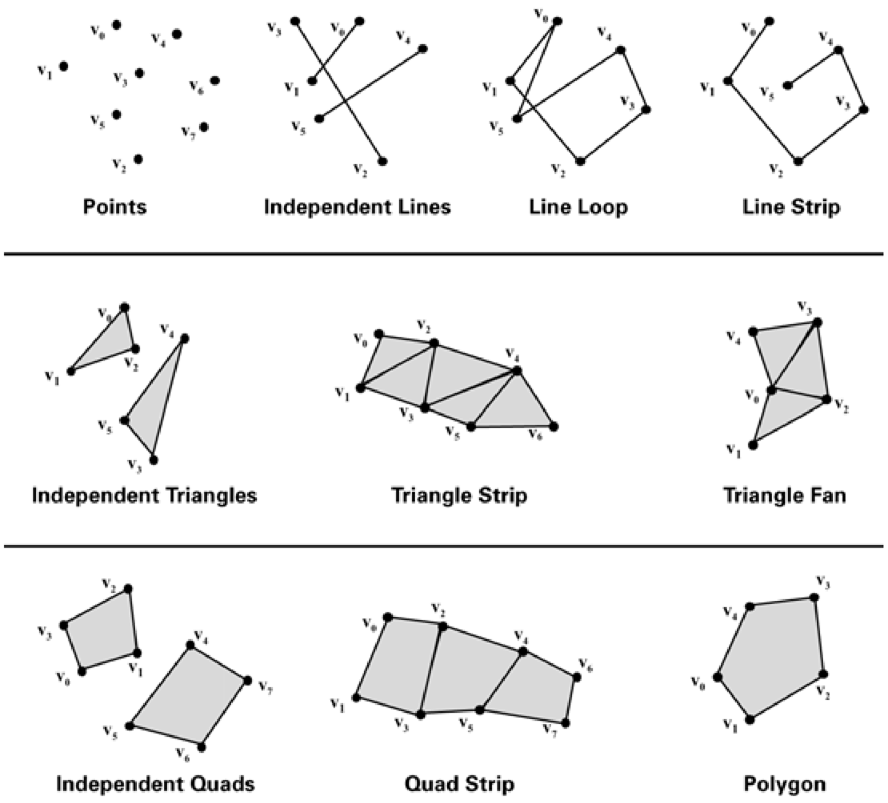

**Figure 1-4.** Types of Geometric Primitives

## 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

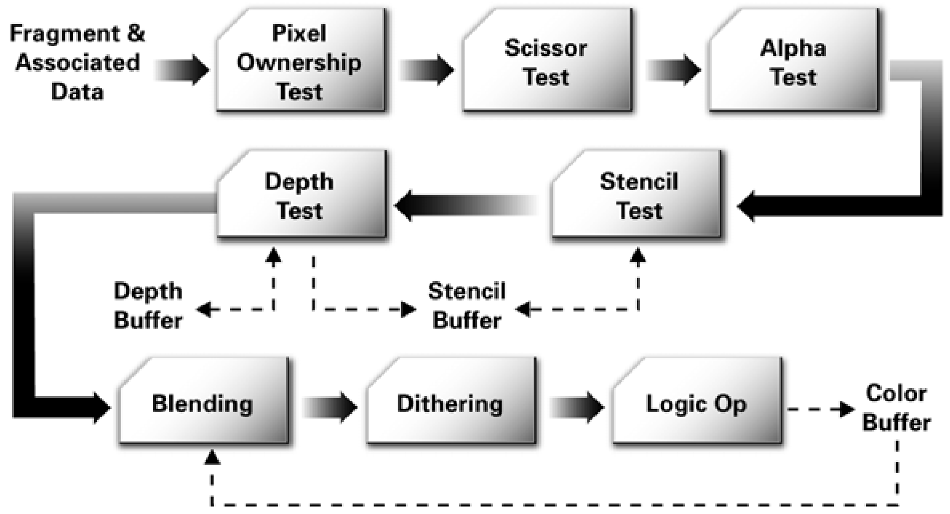

**Figure 1-5.** Standard OpenGL and Direct3D Raster Operations

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-

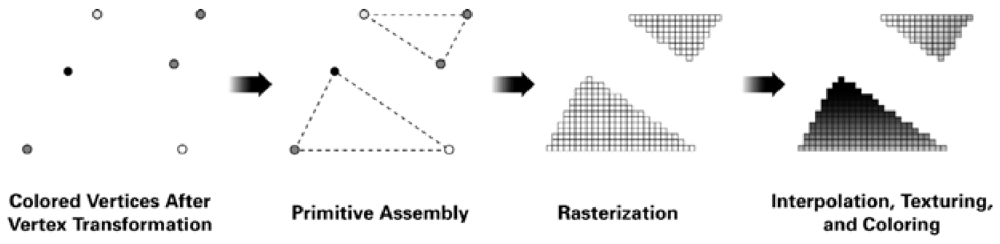
Colored Vertices After Vertex Transformation
Primitive Assembly
Rasterization
Interpolation, Texturing, and Coloring

**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*

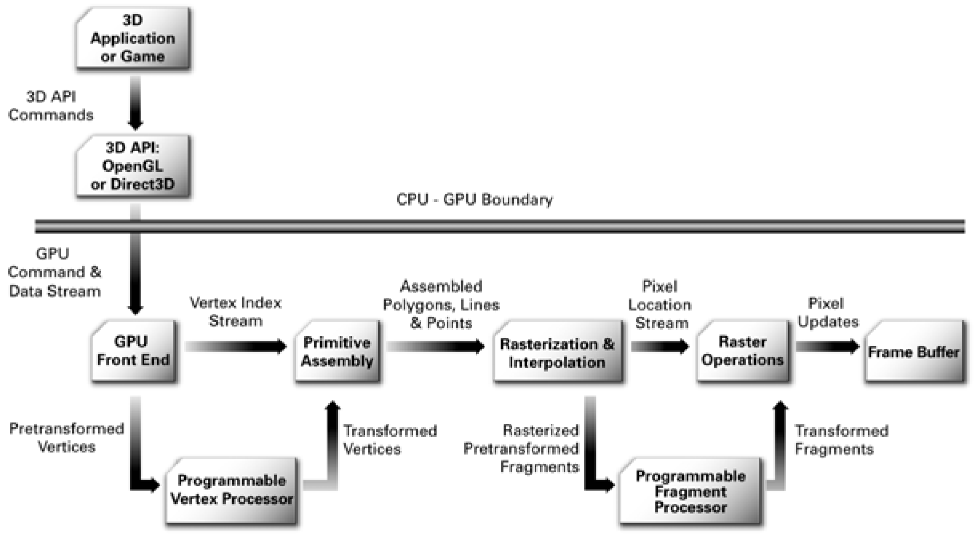

**Figure 1-7.** The Programmable Graphics Pipeline

*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

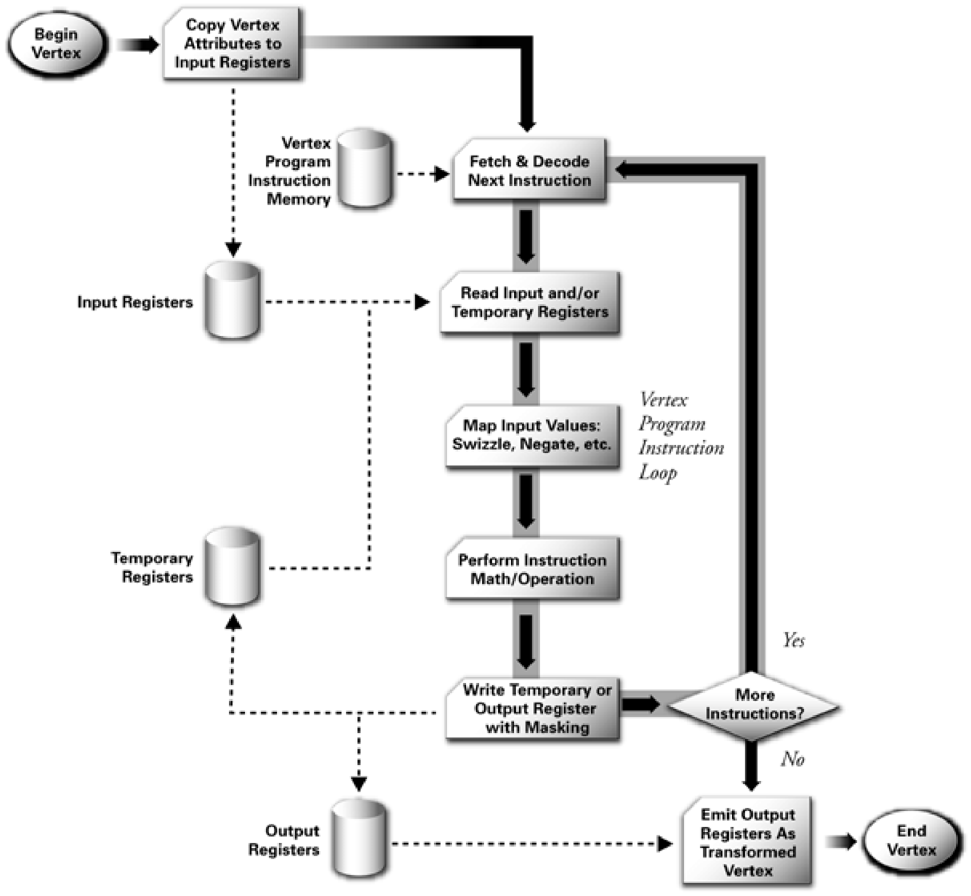
Begin Vertex
Copy Vertex Attributes to Input Registers
Vertex Program Instruction Memory
Fetch & Decode Next Instruction
Input Registers
Read Input and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Vertex Program Instruction Loop
Temporary Registers
Perform Instruction Math/Operation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

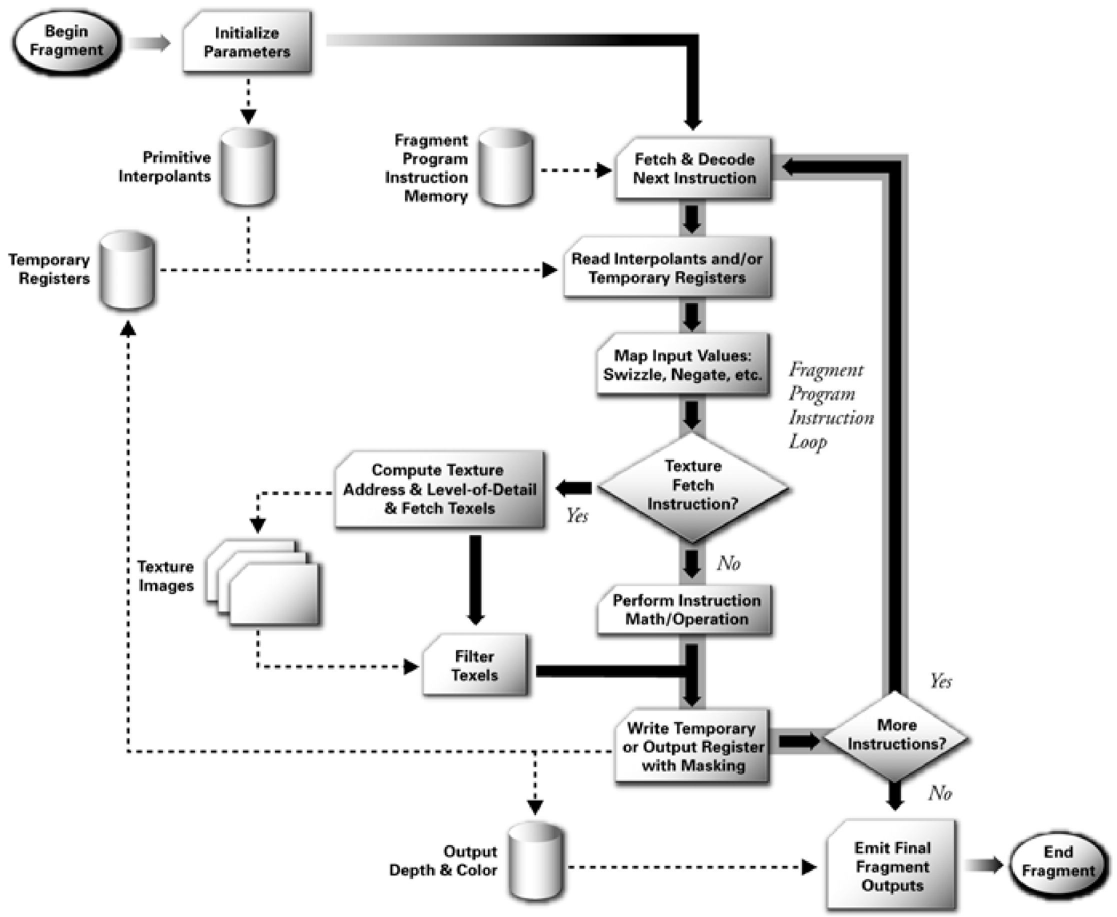
Begin Fragment
Initialize Parameters
Primitive Interpolants
Fragment Program Instruction Memory
Fetch & Decode Next Instruction
Temporary Registers
Read Interpolants and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Fragment Program Instruction Loop
Compute Texture Address & Level-of-Detail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

### 1.2.5 Cg Provides Vertex and Fragment Programmability

These two programmable processors in your GPU require you, the application programmer, to supply a program for each processor to execute. What Cg provides is a language and a compiler that can translate your shading algorithm into a form that your GPU's hardware can execute. With Cg, rather than program at the level shown in Figures 1-8 and 1-9, you can program in a high-level language very similar to C.

## 1.3 Cg's Historical Development

Cg's heritage comes from three sources, as shown in Figure 1-10. First, Cg bases its syntax and semantics on the general-purpose C programming language. Second, Cg incorporates many concepts from offline shading languages such as the RenderMan Shading Language, as well as prior hardware shading languages developed by academia. Third, Cg bases its graphics functionality on the OpenGL and Direct3D programming interfaces for real-time 3D.

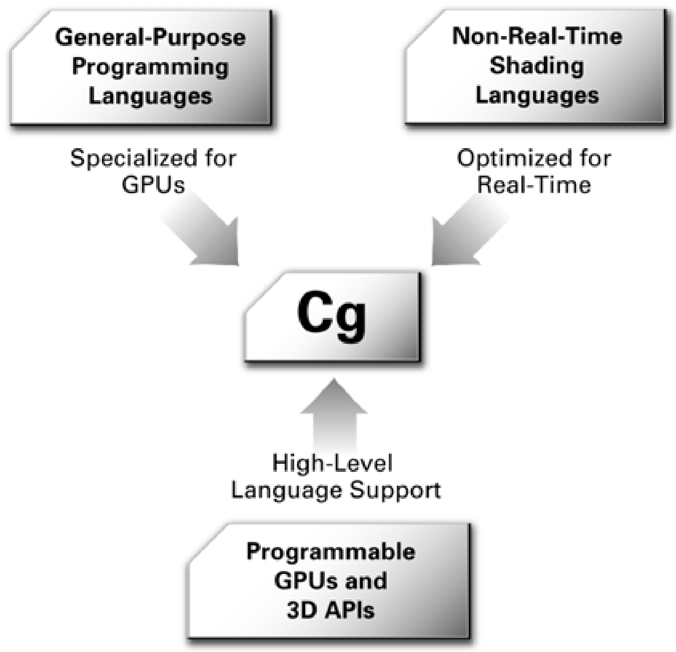
General-Purpose Programming Languages
Non-Real-Time Shading Languages
Specialized for GPUs
Optimized for Real-Time
Cg
High-Level Language Support
Programmable GPUs and 3D APIs

**Figure 1-10.** Sources of Cg's Technology Heritage

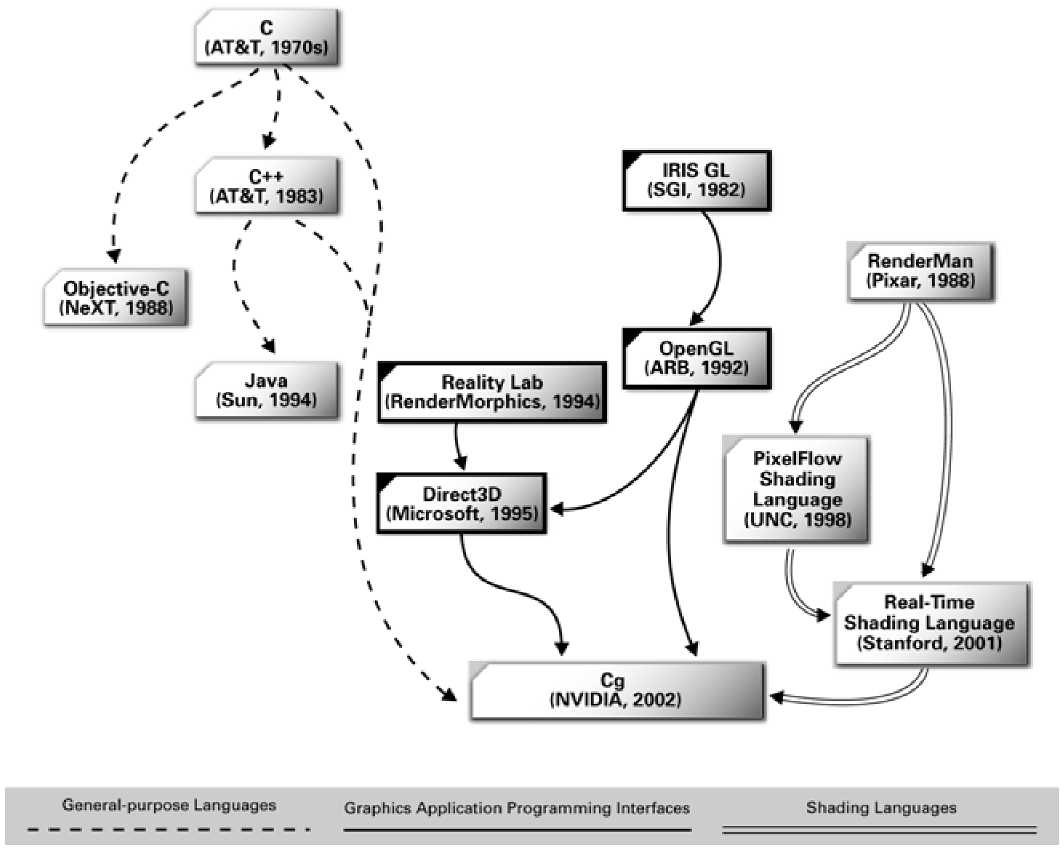
C (AT&T, 1970s)
C++ (AT&T, 1983)
Objective-C (NeXT, 1988)
Java (Sun, 1994)
IRIS GL (SGI, 1982)
RenderMan (Pixar, 1988)
OpenGL (ARB, 1992)
Reality Lab (RenderMorphics, 1994)
Direct3D (Microsoft, 1995)
PixelFlow Shading Language (UNC, 1998)
Real-Time Shading Language (Stanford, 2001)
Cg (NVIDIA, 2002)
General-purpose Languages
Graphics Application Programming Interfaces
Shading Languages

**Figure 1-11.** Inspirations for Cg's Development

Figure 1-11 shows the general-purpose programming languages, 3D application programming interfaces, and shading languages that inspired Cg's development.

Earlier, we mentioned how Cg leverages C's syntax and semantics. Over the course of this book, you will find that Cg mostly does what C programmers expect. Cg differs from C in situations where either Cg's specialization for GPUs or performance justifies a change.

## 1.3.1 Microsoft and NVIDIA's Collaboration to Develop Cg and HLSL

NVIDIA and Microsoft collaborated to develop the Cg language. Microsoft calls its implementation High-Level Shading Language, or HLSL for short. HLSL and Cg are the same language but reflect the different names each company uses to identify the language and its underlying technology. HLSL is a part of Microsoft's DirectX Graphics, a component of the DirectX 9 multimedia framework. Direct3D is the 3D component of Microsoft's DirectX Graphics. Cg is independent of the 3D programming interface and fully integrates with either Direct3D or OpenGL. A properly written Cg application can be written once and then work with either OpenGL or Direct3D.

This flexibility means that NVIDIA's Cg implementation provides a way to author programs that work with both dominant 3D programming interfaces and whatever operating system you choose. Cg works whether you choose Windows, Linux, Mac OS X, a game console, or embedded 3D hardware as your 3D computing platform. Cg programs work with hardware from multiple hardware vendors because Cg layers cleanly upon either Direct3D or OpenGL. Cg programs work on programmable GPUs from all the major graphics hardware vendors, such as 3Dlabs, ATI, Matrox, and NVIDIA.

The multivendor, cross-API, and multiplatform nature of the Cg language makes it the best choice when writing programs for programmable GPUs.

## 1.3.2 Noninteractive Shading Languages

The RenderMan Interface Standard describes the best-known shading language for noninteractive shading. Pixar developed the language in the late 1980s to generate high-quality computer animation with sophisticated shading for films and commercials. Pixar has created a complete rendering system with its implementation of the RenderMan Interface Standard, the offline renderer PRMan (PhotoRealistic RenderMan). The RenderMan Shading Language is just one component of this system.

### Shade Trees

The inspiration for the RenderMan Shading Language came from an earlier idea called *shade trees*. Rob Cook, then at Lucasfilm Ltd., which later spun off Pixar, published a SIGGRAPH paper about shade trees in 1984. A shade tree organizes various shading operations as nodes within a tree structure. Figure 1-12 shows a shade tree for rendering a copper surface. The leaf nodes are data inputs to the shade tree. The nonleaf nodes represent simple shading operations. During the process of rendering, the renderer evaluates the shade tree associated with a given surface to determine the color of the surface in the rendered image. To evaluate a shade tree, a renderer performs the

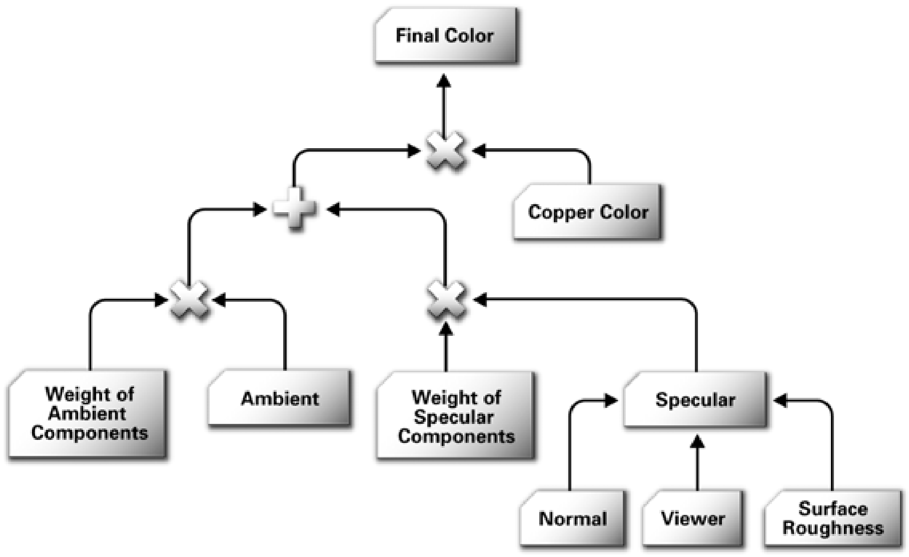

**Figure 1-12.** A Shade Tree Example, Based on Rob Cook's Original SIGGRAPH Paper

shading operation associated with the topmost node in the shade tree. However, to evaluate a given node, the renderer must first evaluate the node's child nodes. This rule is applied recursively to evaluate the shade tree fully. The result of a shade tree evaluation at a given point on a surface is the color of that point.

Shade trees grew out of the realization that a single predefined shading model would never be sufficient for all the objects and scenes one might want to render.

Shade tree diagrams are great for visualizing a data flow of shading operations. However, if the shade trees are complex, their diagrams become unwieldy. Researchers at Pixar and elsewhere recognized that each shade tree is a limited kind of program. This realization provided the impetus for a new kind of programming language known as a shading language.

## The RenderMan Shading Language

The RenderMan Shading Language grew out of shade trees and the realization that open-ended control of the appearance of rendered surfaces in the pursuit of photorealism requires programmability.

Today most offline renderers used in actual production have some type of support for a shading language. The RenderMan Shading Language is the most established and best known for offline rendering, and it was significantly overhauled and extended in the late 1990s.

### Hardware-Amenable Shading Languages

A hardware implementation of an algorithm is most efficient when the task decomposes into a long sequence of stages in which each stage's communication is limited to its prior stage and its subsequent stage (that is, when it can be pipelined).

The vertex-based and fragment-based pipeline described in Section 1.2 is extremely amenable to hardware implementation. However, the Reyes algorithm used by PhotoRealistic RenderMan is not very suitable for efficient hardware implementation, primarily due to its higher-level geometry handling. Contemporary GPUs rely completely on a graphics pipeline based on vertices and fragments.

Researchers at the University of North Carolina (UNC) began investigating programmable graphics hardware in the mid-1990s, when UNC was developing a new programmable graphics hardware architecture called PixelFlow. This project fostered a new line of computer graphics research into hardware-amenable shading languages by Marc Olano and others at UNC. Unfortunately, PixelFlow was too expensive and failed commercially.

Subsequently, researchers at Silicon Graphics worked on a system to translate shaders into multiple passes of OpenGL rendering. Although the targeted OpenGL hardware was not programmable in the way GPUs are today, the OpenGL Shader system orchestrates numerous rendering passes to achieve a shader's intended effect.

Researchers at Stanford University, including Kekoa Proudfoot, Bill Mark, Svetoslav Tzvetkov, and Pat Hanrahan, began building a shading language designed specifically for second-generation and third-generation GPUs. This language, known as the Stanford Real-Time Shading Language (RTSL), could compile shaders written in RTSL into one or more OpenGL rendering passes.

The research at Stanford inspired NVIDIA's own effort to develop a commercial-quality hardware-amenable shading language. Bill Mark joined NVIDIA in 2001 to lead the effort to define and implement the shading language we now call Cg. During this time, NVIDIA collaborated with Microsoft to agree on a common language syntax and feature set.

### 1.3.3 Programming Interfaces for 3D Graphics

The third influence on Cg was the pair of standard 3D programming interfaces, OpenGL and Direct3D. The influence of these programming interfaces on Cg is ongoing, as is explained in the next section.

## 1.4 The Cg Environment

Cg is just one component of the overall software and hardware infrastructure for rendering complex 3D scenes with programmable GPUs at real-time rates. This section explains how Cg interacts with actual 3D applications and games.

### 1.4.1 Standard 3D Programming Interfaces: OpenGL and Direct3D

In the old days of 3D graphics on a PC (before there were GPUs), the CPU handled all the vertex transformation and pixel-pushing tasks required to render a 3D scene. The graphics hardware provided only the buffer of pixels that the hardware displayed to the screen. Programmers had to implement their own 3D graphics rendering algorithms in software. In a sense, everything about vertex and fragment processing back then was completely programmable. Unfortunately, the CPU was too slow to produce compelling 3D effects.

These days, 3D applications no longer implement their own 3D rendering algorithms using the CPU; rather, they rely on either OpenGL or Direct3D, the two standard 3D programming interfaces, to communicate rendering commands to the GPU.

#### OpenGL

In the early 1990s, Silicon Graphics developed OpenGL in coordination with an organization called the OpenGL Architecture Review Board (ARB), which comprised all the major computer graphics system vendors. Originally, OpenGL ran only on powerful UNIX graphics workstations. Microsoft, a founding member of the ARB, then implemented OpenGL as a way to support 3D graphics for its Windows NT operating system. Microsoft later added OpenGL support to Windows 95 and all of Microsoft's desktop operating systems.

OpenGL is not limited to a single operating or windowing system. In addition to supporting UNIX workstations and Windows PCs, OpenGL is supported by Apple for its Macintosh personal computers. Linux users can use either the Mesa open-source implementation of OpenGL or a hardware-accelerated implementation such as NVIDIA's OpenGL driver for Linux. This flexibility makes OpenGL the industry's best cross-platform programming interface for 3D graphics.

Over the last decade, OpenGL has evolved along with graphics hardware. OpenGL is extensible, meaning that OpenGL implementers can add new functionality to OpenGL in an incremental way. Today, scores of OpenGL extensions provide access to all the latest GPU features. This includes ARB-standardized extensions for vertex and fragment programmability. As extensions are established, they are often rolled into the core OpenGL standard so that the standard as a whole advances. At the time of this writing, the current version of OpenGL is 1.4. Ongoing work to evolve OpenGL is underway in various OpenGL ARB working groups. This work includes both assembly-level and high-level programmable interfaces. Because Cg operates as a layer above such interfaces, it will continue to function with future revisions of OpenGL in a compatible manner.

### Direct3D

Microsoft began developing the Direct3D programming interface about 1995 as part of its DirectX multimedia initiative. Direct3D is one of the programming interfaces that make up DirectX. Microsoft introduced DirectX and Direct3D to jump-start the consumer market for 3D graphics, particularly gaming, on Windows PCs. Microsoft's Xbox game console also supports Direct3D. Direct3D is the most popular graphics API for games on Windows, due to its history of closely matching the capabilities of available graphics hardware.

Every year or so, Microsoft has updated DirectX, including Direct3D, to keep up with the rapid pace of PC hardware innovation. The current version of DirectX at the time of this writing is DirectX 9, which includes HLSL, Microsoft's implementation of the same language syntax and constructs found in Cg.

### 3D Programming Interface Détente

A few years ago, OpenGL and Direct3D competed to see which programming interface would dominate, particularly in the domain of Windows PCs. The competition continues to be good for both programming interfaces, and each has improved in performance, quality, and functionality. In the area of GPU programmability that Cg addresses, both programming interfaces have comparable capabilities. This is because both OpenGL and Direct3D run on the same GPU hardware and the graphics hardware determines the available functionality and performance. OpenGL has a slight advantage in functionality because hardware vendors are better able to expose their entire feature set through OpenGL, though vendor-specific extensions do add some complexity for developers.

Most software developers now choose a 3D programming interface based on programmer preference, history, and their target market and hardware platform, rather than on technical grounds.

Cg supports either programming interface. You can write Cg programs so that they work with either the OpenGL or Direct3D programming interface. This is a huge boon for 3D content developers. They can pair their 3D content with programs written in Cg and then render the content no matter what programming interface the final application uses for 3D rendering.

### 1.4.2 The Cg Compiler and Runtime

No GPU can execute Cg programs directly from their textual form. A process known as compilation must translate Cg programs into a form that the GPU can execute. The Cg compiler first translates your Cg program into a form accepted by the application's choice of 3D programming interface, either OpenGL or Direct3D. Then your application transfers the OpenGL or Direct3D translation of your Cg program to the GPU using the appropriate OpenGL or Direct3D commands. The OpenGL or Direct3D driver performs the final translation into the hardware-executable form your GPU requires.

The details of this translation depend on the combined capabilities of the GPU and 3D programming interface. How a Cg program compiles its intermediate OpenGL or Direct3D form depends on the type and generation of GPU in your computer. It may be that your GPU is not capable of supporting a particular valid Cg program because of limitations of the GPU itself. For example, your Cg fragment program will not compile if your program accesses more texture units than your target GPU supports.

### Support for Dynamic Compilation

When you compile a program with a conventional programming language such as C or C++, compilation is an offline process. Your compiler compiles the program into an executable that runs directly on the CPU. Once compiled, your program does not need to be recompiled, unless you change the program code. We call this *static compilation*.

Cg is different because it encourages *dynamic compilation,* although static compilation is also supported. The Cg compiler is not a separate program but part of a library known as the Cg runtime. 3D applications and games using Cg programs must link with the Cg runtime. Applications using Cg then call Cg runtime routines, all prefixed with the letters **`cg`**, to compile and manipulate Cg programs. Dynamic compilation allows Cg programs to be optimized for the particular model of GPU installed in the user's machine.

### CgGL and CgD3D, the 3D-API-Specific Cg Libraries

In addition to the core Cg runtime, Cg provides two closely related libraries. If your application uses OpenGL, you will use the CgGL library to invoke the appropriate OpenGL routines to pass your translated Cg program to the OpenGL driver. Likewise, if your application uses Direct3D, you will use the CgD3D library to invoke the appropriate Direct3D routines to pass your translated Cg program to the Direct3D driver. Normally, you would use either the CgGL or the CgD3D library, but not both, because most applications use either OpenGL or Direct3D, not both.

Compared with the core Cg runtime library that contains the Cg compiler, the CgGL and CgD3D libraries are relatively small. Their job is to make the appropriate OpenGL or Direct3D calls for you to configure Cg programs for execution. These calls transfer a translated Cg program to the appropriate driver that will further translate the program into a form your GPU can execute. For the most part, the CgGL and CgD3D libraries have similar routines. The routines in the CgGL library begin with **`cgGL`**; the routines in the CgD3D library begin with **`cgD3D`**.

### How the Cg Runtime Fits into Your Application

Figure 1-13 shows how a typical 3D application uses the Cg libraries. If you are a programmer, you will want to learn more about the Cg runtime and the specific library for the 3D API your application uses to render 3D graphics. Most of this book focuses on the Cg language itself and on how to write Cg programs, but Appendix B has more information about the Cg runtime library.

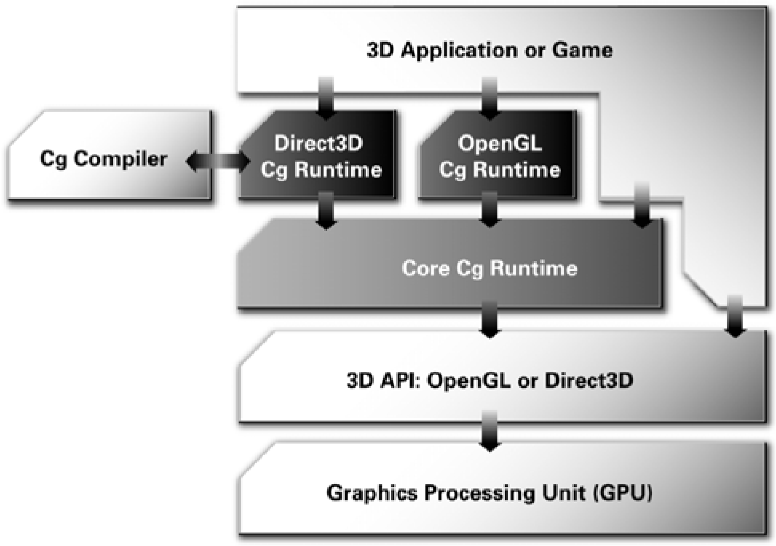

**Figure 1-13.** How Cg Fits into a Standard Cg Application

## 1.4.3 The CgFX Toolkit and File Format

Cg programs need 3D models, textures, and other data to operate on. A Cg program without any associated data is useless. Cg programs and data also require the correct 3D programming interface configuration and state. It is often helpful to have a way to bundle all the information required to render a 3D model, including its associated Cg program.

### What CgFX Provides

CgFX is a standardized file format for representing complete effects and appearances. As they did with Cg, Microsoft and NVIDIA collaborated to develop the CgFX format. CgFX files are text-based, with a syntax that is a superset of Cg's, and may contain any number of Cg programs. The `.fx` suffix identifies CgFX files. A CgFX file describes the complete render state for a particular effect: multiple passes, texture states, and any number of individual vertex and fragment programs may be defined to create a complete appearance or effect. An accompanying development toolkit is provided for using and parsing CgFX files. The toolkit exposes user-interface hooks to host applications, so that CgFX-aware applications can automatically supply meaningful controls and semantics to users and developers alike.

Cg programs describe the vertex or fragment processing that takes place in a single rendering pass, but some complex shading algorithms require multiple rendering passes. CgFX offers a format to encode complex multipass effects, including designating which Cg program is used for each rendering pass.

More specifically, CgFX supports three additional capabilities beyond what the core Cg language supports:

1. CgFX provides a mechanism for specifying multiple rendering passes and optional multiple implementations for a single effect.
2. CgFX allows you to specify nonprogrammable rendering states, such as alpha-test modes and texture-filtering. The settings for these render states may take the form of simple expressions, which are evaluated on the CPU when the effect is initialized.
3. CgFX allows annotations to be added to shaders and shader parameters. These annotations provide additional information to applications, including content creation applications. For example, an annotation can specify the allowed range of values for a shader parameter.

### Multiple Shader Instancing

The CgFX file format encapsulates multiple implementations of Cg programs for a given shader. This means you can have one Cg shader program written for a third-generation or fourth-generation GPU, while also including a simpler program that supports a less capable, second-generation GPU. An application loading the CgFX file can determine at runtime the most appropriate shader implementation to use based on the computer's available GPU.

Multiple instancing of Cg programs with CgFX is one way to address the functional variations in GPUs of different generations or different hardware vendors. Multiple instancing also lets you develop a Cg program specialized for a particular 3D API—for example, if OpenGL exposes extra functionality through an extension. Cg programs specialized for Direct3D, standard OpenGL, or OpenGL with extensions can all be contained in a single CgFX file.

### CgFX and Digital Content Creation

The CgFX Toolkit consists of the CgFX compiler, which supports the full CgFX syntax; the CgFX runtime API, for loading and manipulating CgFX files; and plug-in modules for major digital content creation (DCC) applications such as

Alias|Wavefront's Maya and Discreet's 3ds max. Figure 1-14 shows these applications making use of CgFX. Softimage|XSI 3.0 provides direct support for Cg compilation in its Render Tree.

Prior to CgFX, there was no standard way for a DCC application to export 3D content with all the associated shading knowledge necessary to render the content in real time. Now the major DCC applications use CgFX in their content creation process and support the CgFX file format. This means that CgFX can significantly improve the artistic workflow from DCC applications to real-time games and other 3D applications. Using CgFX, artists can view and tweak Cg shaders and associated 3D content to see, from within the DCC tool of their choice, how their work will appear in a 3D game or application.

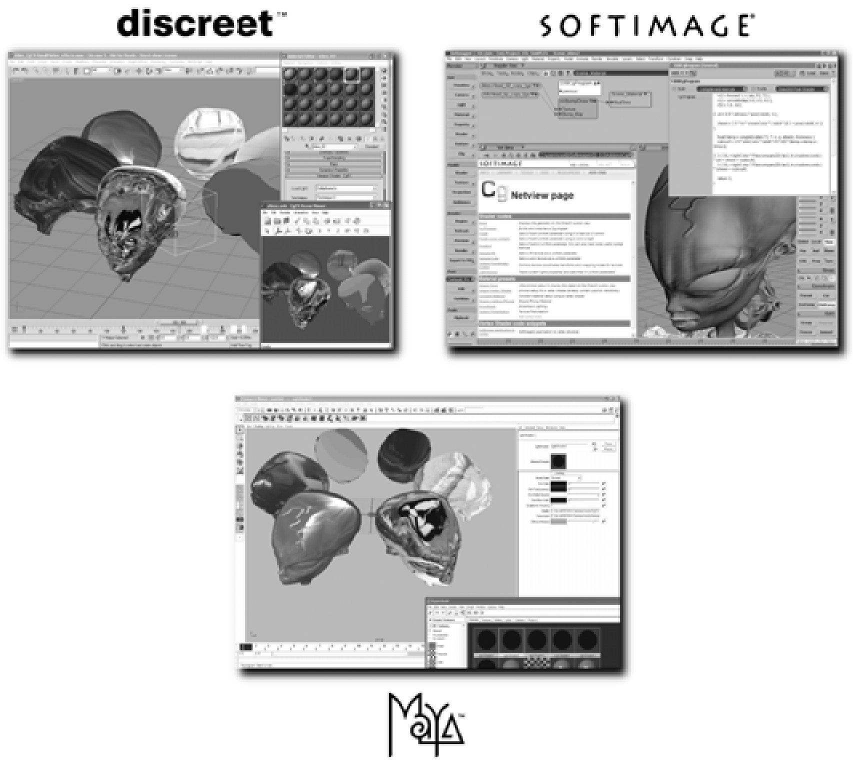

**Figure 1-14.** Digital Content Creation Applications That Use Cg and CgFX

## How CgFX Fits into Your Application

Figure 1-15 shows how a CgFX file containing multiple instanced shaders is used by an application in conjunction with the Cg runtime and your choice of rendering API. Most of this book focuses on the Cg language itself and on how to write Cg programs, rather than CgFX, but see Appendix C for more information about the CgFX file format and its associated runtime API.

## Onward to the Tutorial

Having finished this introduction to the Cg programming language, you are now ready to take on the tutorial chapters, which will teach you how to write Cg programs.

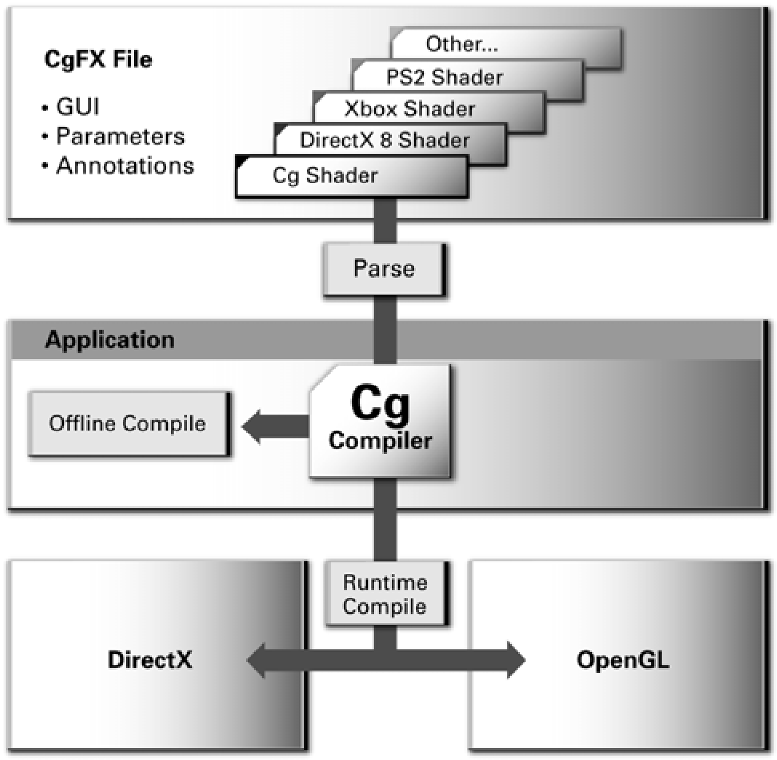

**Figure 1-15.** How CgFX Fits into a Standard Application

## 1.5 Exercises

The exercises at the end of each chapter help you review your knowledge and develop practical programming skills.

1. **Answer this:** Name two standard 3D programming interfaces for which you can compile Cg programs. What operating systems does each programming interface support?
2. **Answer this:** What are the major stages of the graphics pipeline? In what order are the stages arranged?
3. **Answer this:** Where do vertex and fragment programs fit into the pipeline?
4. **Answer this:** What is a vertex? What is a fragment? Distinguish a fragment from a pixel.
5. **Try this yourself:** We haven't begun writing Cg programs yet (we'll get there soon enough in the next chapter), so take a break and watch a good feature-length computer graphics animation such as *Monsters, Inc.*

## 1.6 Further Reading

Cg builds on a host of concepts in computer language design, computer hardware design, and computer graphics. Doing justice to all these contributions in the context of this tutorial is not always practical. What we attempt in the "Further Reading" section at the end of each chapter is to offer you pointers to learn more about the contributions that underlie the topics in each chapter.

There are plenty of books on C. *The C Programming Language*, *Third Edition* (Prentice Hall, 2000), by Brian Kernighan and Dennis Ritchie, is a classic; the authors invented the C language. Cg includes concepts from both C and C++. There now may actually be more books about C++ than about C. The classic C++ book is *The C++ Programming Language, Third Edition* (Addison-Wesley, 2000), by Bjarne Stroustrup, who invented the language.

To learn more about the RenderMan Shading Language, read *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics* (Addison-Wesley, 1989), by Steve Upstill. Pat Hanrahan and Jim Lawson published a SIGGRAPH paper about

RenderMan called "A Language for Shading and Lighting Calculations" (ACM Press) in 1990.

Robert Cook's 1984 SIGGRAPH paper titled "Shade Trees" (ACM Press) motivated the development of RenderMan.

The development of programmable graphics hardware and its associated languages has been an active and fruitful research area for almost a decade. Anselmo Lastra, Steven Molnar, Marc Olano, and Yulan Wang at UNC published an early research paper in 1995 titled "Real-Time Programmable Shading" (ACM Press). Researchers at UNC also published several papers about their programmable PixelFlow graphics architecture. Marc Olano and Anselmo Lastra published a SIGGRAPH paper titled "A Shading Language on Graphics Hardware: The PixelFlow Shading System" (ACM Press) in 1998.

Kekoa Proudfoot, Bill Mark, Svetoslav Tzvetkov, and Pat Hanrahan published a SIGGRAPH paper in 2001 titled "A Real-Time Procedural Shading System for Programmable Graphics Hardware" (ACM Press) that describes a GPU-oriented shading language developed at Stanford.

*Real-Time Rendering, Second Edition* (A. K. Peters, 2002), written by Eric Haines and Tomas Akenine-Möller, is an excellent resource for further information about graphics hardware and interactive techniques.

*The OpenGL Graphics System: A Specification* documents the OpenGL 3D programming interface. The best tutorial for learning OpenGL programming is the *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Third Edition* (Addison-Wesley, 1999), by Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. The **www.opengl.org** Web site serves up much more information about OpenGL.

Documentation for the Direct3D programming interface is available from Microsoft's **msdn.microsoft.com** Web site.

NVIDIA provides further information about the Cg runtime, CgFX, and Cg itself on its Developer Web site at **developer.nvidia.com/Cg**.