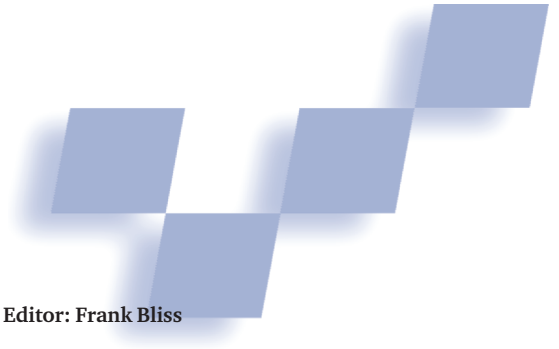
An Introductory Tour of Interactive Rendering



Editor: Frank Bliss

Eric Haines
Autodesk

This article provides an understanding of consumer-level personal graphics processors and a sense of how different algorithms are developed in response to their capabilities.

In the past decade, 3D graphics accelerators for personal computers have transformed from an expensive curiosity to a basic requirement. Along the way, the traditional interactive rendering pipeline has undergone a major transformation, from a fixed-function architecture to a programmable stream processor. Along with the faster speeds and new functionality have come new ways of thinking about how we can use graphics hardware for rendering and other tasks. While expanded capabilities have in some cases simply meant that old algorithms could now be run at interactive rates, the performance characteristics of the new hardware has also meant that novel approaches to traditional problems often yield superior results.

The term *interactive rendering* can have many different meanings. Certainly the user interfaces for operating systems, photo manipulation capabilities of paint programs, and line and text display attributes of CAD drafting programs are all graphical in nature. All of these types of 2D graphical elements have benefited, in speed and quality, from new graphics hardware developed over the years. However, this article will focus on 3D rendering, because that is where much of the recent interest, competition, and progress have been. With Apple's OS X and Microsoft's upcoming Avalon GUI for Longhorn, accelerated 3D features have become an integral part of the operating system's methods of communicating with the user. As an example, high-quality texture mapping used on 3D polygons lends itself well to interactive warping and zooming of 2D images.

Given the depth and breadth of the field of interactive rendering, an article summarizing all the significant algorithms and hardware developments in the past 10 years would become a long yet shallow litany of taxonomies and terms. Instead, here, I use a selection of topics from various areas to illustrate this central theme: the influence of 3D graphics hardware on algorithm design.

Basic principles

For interactive rendering, the goal is, above all, to present a responsive user experience. No technique, no matter how beautiful, can afford to slow the refresh rate lower than around 6 frames a second and still be considered interactive. Most video console games strive for a minimum of 30 frames or more. A requirement of 60 frames per second yields a budget of less than 17 milliseconds per frame. As such, algorithms have to be both fast overall and also avoid much variability in rendering time. As an example, two common methods for shadow generation are shadow buffers and shadow volumes. Each has its own strengths and weaknesses, but one serious flaw of the basic shadow volumes technique is that for some camera positions many large polygons must be rendered to compute the shadows, while in others a few small polygons are needed. This variability in performance can cause an uneven display rate, a problem that the shadow buffer technique usually avoids.

Three basic principles of interactive rendering for 3D are approximation, preparation, and amortization. Approximation is a basic principle of all computer graphics, as we cannot hope to track all photons in their full glory, nor the placement and reactions of all atoms to these photons (including those atoms in the eye and brain). The basic building blocks of modern interactive rendering are textured triangles with vertex data interpolation. This bias toward triangles shows the roots of development of modern PC graphics processors. Computer games are the major driving force for the sale of graphics cards, and for the most part game developers do not have needs for fast, high-quality line and point rendering versus, say, CAD or data visualization applications.

Preparation means computing in advance or on the fly various types of data and reusing these results. For example, one common technique used for static diffuse shaded environments with fixed lighting conditions is to bake in some or all of the lighting effects, storing a color per vertex in the model or the light's effect on a surface in a texture map. Doing so saves repeatedly computing the identical results frame after frame.

Hand in hand with preparation is the idea of amortization. If during interaction a computation can be per-

 Published by the IEEE Computer Society

formed once and reused, its initial cost can be justified by the savings it yields over a number of frames it has used. For example, for a given view the original model might be replaced by a simplified 3D model or even a 2D image (called an impostor). If the view does not change significantly for a series of frames and the simplified model can be used, the creation cost is recouped. The idea of amortization is a principle important to interactive rendering. In contrast, film rendering systems normally use a rendering farm, one processor per frame, and so must reload all data for each individual image generated.

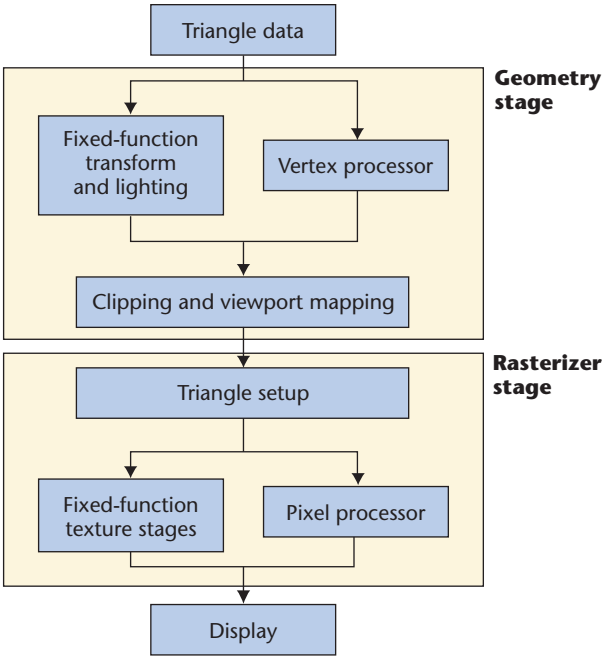
Evolving pipeline

The traditional 3D z-buffer pipeline starts with data from the application in the form of vertices of individual polygons (converted to triangles for simplicity) transformed, shaded, and clipped in the geometry stage. The surviving potentially visible triangles are then filled pixel by pixel in the rasterizer stage, using the data interpolated from the vertices.

A graphics processor implementing this pipeline obtains much of its performance by using both task and data parallelism. The hardware pipeline gains its speed from the same concept used on an assembly line. As a simple example, as one stage transforms a triangle, another stage rasterizes a different triangle, similar to how one automobile might have its engine installed while further down the line another has its doors attached. In actual graphics hardware, many triangles may be in the pipeline at one time. A single triangle might take a fair amount of time to work its way through the whole pipeline, but the overall rate at which triangles are processed is much higher, since (ideally) each part of the pipeline is active at the same moment.

It is a principle that at any given moment there is always some stage along the pipeline that is the bottleneck, one that is slower than every other stage. This bottleneck can change, depending on the input data. For example, rendering large polygons can cause the rasterizer to be the limiting factor, while rendering meshes of tiny polygons might make vertex transformation the bottleneck. We can apply parallelism in several ways to improve a pipeline stage's performance. One straightforward method is putting a number of units in parallel to transform vertices or to fill triangle pixels. First-in-first-out buffers are also commonly used between stages to allow temporary backlogs and so avoid stalling processing further up the pipeline.

We can see the evolution of PC graphics cards over the 1990s by traveling backward up this pipeline. At first it was a thrill to computer graphics researchers that they could even display a 24-bit color image on a PC at a reasonable price. The first cards to offer 3D acceleration provided a z-buffer and the ability to rasterize a pretransformed triangle. In 1999, the transform and lighting part of the pipeline moved onto the GPU (and this was when the term *graphics processing unit* was introduced). During this time, some game developers considered this a step backward, in terms of flexibility, as the CPU itself had no limits on what shading model was used to compute vertex colors. Graphics hardware



1 Vertex and fragment processor enhanced pipeline from the user's point of view. The fixed-function hardware no longer exists in newer GPUs, and the fixed-function API calls are translated into shader programs by the graphics driver.

offered a fixed-function shading model: ambient, diffuse, Phong specular, alpha-blending, fog, and whatever else the graphics API happened to support. New features and modes might be available through, say, OpenGL's extensions mechanism, but most developers disliked programming for specific cards. In addition, hardware vendors were finding they were devoting noticeable numbers of transistors to specific shading algorithms.

The response to these problems was the development of programmable vertex and fragment processors, arranged in the pipeline as an alternate path from the fixed-function calls available in the API (see Figure 1). These alternate paths were controlled by relatively short programs, commonly called vertex and fragment shaders (or for DirectX, pixel shaders).

Instead of fixed-function shading per vertex, we can program the vertex processor to perform all sorts of computations. Specifically, a vertex processor works on one vertex of a triangle at a time, independently of the other vertices. The inputs to a vertex processor are a set of coordinate values associated with a vertex and a set of constants, meant for per-surface properties. The vertex processor program itself consists of a number of operators that manipulate these coordinates, typically as vectors (for example, dot product, subtract, normalize). The output is a new vertex, one that can have a new format. At the minimum, this new vertex consists of an *XYZ* location, but can also have elements such as a normal, colors, and texture coordinates. So, in addition to computing a complex shading model, the vertex processor can also deform the model's geometry in world or view space. Among other operations, this functionality

Resources

I have purposely avoided referencing many older research papers and books in this article in the interest of brevity. The historical material in this article is discussed in depth in a book I coauthored, *Real-Time Rendering*.[1] This book's list of references is available at http://www.realtimerendering.com, a site that also includes links to a wide range of relevant resources. The Nvidia and ATI developer Web sites are particularly useful for understanding the latest developments in the field. In recent years, beyond the normal research publication channels such as conferences and journals, a number of edited collections of articles have appeared as books. In particular, the *GPU Gems* (Addison-Wesley), *ShaderX* (Charles River Media), and *Game Programming Gems* (Charles River Media) book series explain many new techniques and give code samples.

Reference

1. T. Akenine-Möller and E. Haines, *Real-Time Rendering*, 2nd ed., AK Peters, 2002; http://www.realtimerendering.com.

is commonly used for skinning, an animation technique for joints where a vertex is part of a skin that is affected by the matrices of two or more nearby bones. The costs of computing such vertices' locations for each frame is thus offloaded from the CPU to the GPU.

The introduction of the vertex processor was a fairly natural evolution of the pipeline. If a vertex processor is not available on the GPU, or the GPU does not support the length or command set of the vertex program provided, the CPU can perform the computations instead and then pass in the processed vertex data to the graphics card. In comparison, the fragment processor allows operations that are available only if the graphics hardware exists; a CPU-side software emulation of this functionality is much slower, if available at all. While the CPU can often keep up with transforming the thousands of vertices in a typical frame, it cannot keep up with processing the millions of pixel fragments generated.

The fragment (pixel) processor does for pixels what the vertex processor does for vertices, with a few differences. The vertex processor or fixed-function pipeline generates data for the three vertices of a triangle. After culling and clipping, triangle setup occurs, in which data is readied to interpolate across the triangle surface. After interpolation, the fragment processor comes into play. As each pixel that the triangle covers is evaluated, the fragment processor can access a set of values interpolated from the vertex data, as well as stored constant data. Similar to the vertex processor, the fragment processor's program manipulates these values, deciding whether to output a pixel fragment to the z-buffer, and if so, then computing the RGB and optionally the z-depth of this fragment. The fragment processor is evaluated at every pixel covered.

This power can directly lead to improved image quality. For example, instead of standard Gouraud shading—which interpolates the light's contribution at each vertex—a shader can use higher-quality per-pixel Phong shading—which directly computes the light's contribution at each pixel. The idea of approximation comes into play here. Some design decisions include whether the normal interpolated among the vertices should be renormalized per pixel, and whether the light's direction should be computed per pixel or approximated by interpolation. Shorter fragment shader programs run faster; fragment processors provide more options, but also bring up more speed versus quality questions.

Fragment processors have an additional resource available that (older) vertex processors do not: texture access. The fragment processor can use the texture coordinate values to access textures and use the retrieved values in succeeding steps of its program. Using the results of one texture access to influence another is called a dependent texture read. I discuss this idea in more depth in the next section. Older GPUs have limitations on the ability to perform dependent texture reads that have been eliminated in newer processors.

One goal for fragment and vertex processor designers has been to make these two separate languages converge into one. For example, originally vertex processors used floating-point numbers for their input and output, while fragment processors were limited to fixed-point values with few bits. Over time the fragment processor has evolved to the point where it can handle full floating-point math throughout, as well as a plethora of different fixed-point formats. Greater precision helps even simple shader programs avoid computation artifacts such as color banding. Also, high-dynamic-range environment textures can be represented, allowing more realistic reflections. In addition, the fragment processor can now render to more than just a single output frame buffer, instead sending results to up to four render targets. These in turn can be used as input textures for further computations.

It turns out that having dependent texture reads and high-precision pixels are two key capabilities needed to compute more complex shading equations. Every shading equation can be broken down into a series of terms that are multiplied or added together, and so each term can be computed, saved in an off-screen image, and these images then combined—a multipass render. Texture lookups can help replace particularly complex functions by using texture coordinates as inputs. Calling the additional vertex data *texturing coordinates* is almost archaic, as these values are currently used for much more than just texturing. They are good places to store any information that varies slowly over a surface and that is to be interpolated at each pixel. The standard today is to interpolate every value in a perspective-correct manner—that is, that interpolation is linear in object space.

Over the years, the instruction sets of both fragment and vertex processors have significantly increased and converged. The initial offerings did not allow any type of flow control such as static or dynamic if-statements, for-loops, or subroutines. These features have been introduced over time to both types of shaders. Also, the introduction of DirectX's Shader Model 3.0 in 2004 increased the number of instructions in both shaders to 65,536.

Because vertex and fragment processors are now more than powerful enough to do everything that the fixed-function pipeline used to do, graphics hardware no longer needs to have the fixed-function parts of the pipeline as physically separate components on the chip. The traditional shading-related calls in the DirectX and OpenGL APIs are now simply converted internally by the driver to corresponding vertex and fragment programs.

A huge number of shading methods have grown out of these qualitative changes in the pipeline, more than I can possibly cover here. For example, most shaders written for Pixar's RenderMan can be implemented directly on graphics hardware, given enough time per frame. The CPU and GPU are simply different types of Turing-complete, programmable hardware, each with its own strengths and weaknesses.

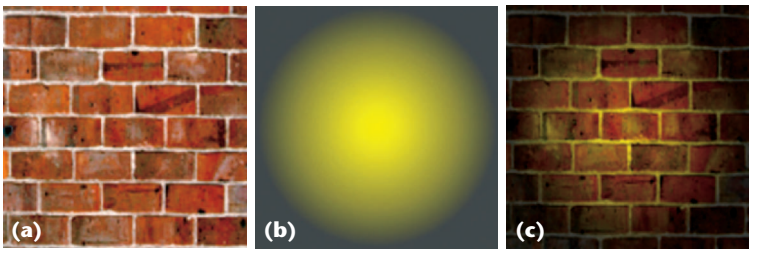
At first, elaborate sets of API calls controlled the new GPU flexibility. Assembly language commands that worked on scalars and vectors soon superseded this cumbersome interface. In recent years, more natural languages such as DirectX 9's high-level shading language have been developed. Developers take for granted coding support tools such as debuggers and profilers when programming a CPU. With the rise of the GPU, the creation of similar tools is an active area of research and development.[1] At the same time, specialized code-development tools have also been created for the GPU, such as shader designers like ATI's RenderMonkey and Nvidia's FX Composer.

Texturing advances

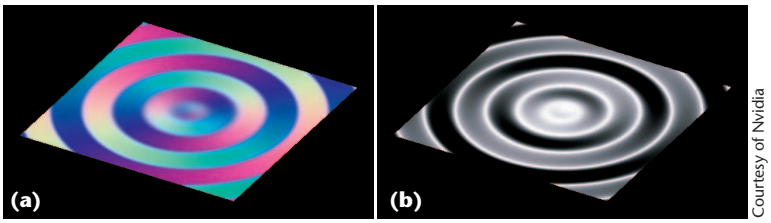
One of the first advantages graphics hardware offered over software was fast texturing. In late 1996, 3dfx brought out its Voodoo chipset. Many in the graphics community consider this offering the true start of 3D acceleration for the PC. There were earlier cards by other companies, but this was arguably the first PC graphics card that was more an accelerator than decelerator, being at least four times faster than its closest competitor.[2]

Around this time, new ideas also began to appear, the most significant being multipass texturing, in which two or more textures affect a single surface. Each texture is applied to the surface in a separate rendering pass, with successive passes adding or multiplying its texture's contents with the previous off-screen image produced, with the final result then displayed.

One common early use of this technique was for light mapping, where a high-resolution surface texture such as a brick wall pattern would have a low-resolution grayscale lighting texture applied to it. In this way, precomputed or artistically created lighting patterns could be added to repetitive wall patterns to give more visual interest and realism to a scene. The high-resolution brick pattern could be repeated without using additional memory, while the low-resolution lighting patterns could vary from wall to wall with little extra cost (see Figure 2).



2 Wall texture combined with a light map texture to give a lit wall. (Courtesy of J.L. Mitchell, M. Tatro, and I. Bullard.)

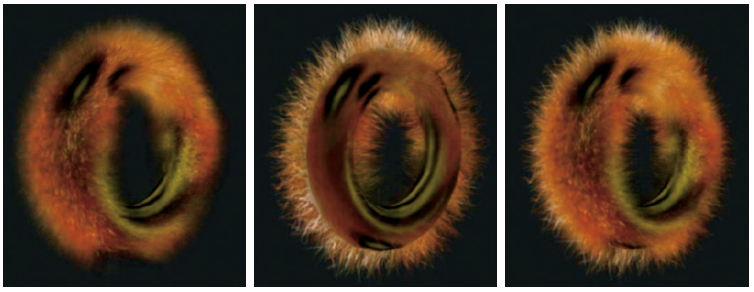


3 (a) A normal map's vector values shown as RGB and (b) the results.

The idea of applying two textures to the same surface has many other uses: adding decals to objects, adding visual detail to terrain, and so on. This concept was general enough that graphics hardware (for example, 3dfx's Voodoo2 in 1998) was developed that supported multitexturing, where two textures could be applied to a surface in a single rendering pass. Doing so is considerably faster than performing a multipass rendering, as the geometry has to be transformed only once. Over the years the number of textures that can be applied to a surface in a single pass has increased to three, four, six, eight, and higher; currently 16 is the maximum. Each texture can have its own set of coordinates defined and interpolated on a single surface, and the resulting values can be combined by addition, multiplication, compositing, and more. In fact, one of the driving forces for adding fragment processors was the desire for more flexibility in combining textures and interpolated data, once it was shown that even simple per-pixel expressions could make a major difference in image quality.

One new texturing method that arose purely from graphics hardware research is dot product bump mapping. The idea is that, instead of a texture holding color information, it actually holds the surface normal at each location. The red/green/blue values of 0 to 255 are mapped to correspond to normal vector $X/Y/Z$ directions of –1.0 to 1.0. So, for example, (58, 128, 235) would correspond to a vector (–0.65, 0.00, 0.84); see Figure 3.

Two tangent vectors are also stored for each vertex, specifying how the normal map relates to the surface. Traditional Gouraud interpolation computes the shade of each vertex, then these colors are interpolated across



4 Concentric shells of layered fur textures and silhouette edge fins produce a convincing simulation of fur. (Courtesy of John Isidoro, ATI Technologies.)

the surface. Dot product bump mapping computes the vector to the light at each vertex. This vector must be computed relative to the surface's orientation, and this is where the tangent vectors come into play. The two tangent vectors and the surface normal at the vertex form a frame of reference, a basis, into which the light vector is transformed. After this transformation, the light vector points toward the light relative to the surface's orientation.

The diffuse contribution for any location is computed using the light vector and the surface's shading normal. So, instead of an RGB per vertex, the differing light vectors, one per vertex, are interpolated. The texture coordinates are also interpolated across the surface, as with regular texture mapping. At each point on the surface we then have a texture location and a light vector. The texture coordinates help retrieve the surface's normal from the normal map. The dot product of the light vector and the bump map's normal is then the diffuse shade of the surface at that point.

This idea of interpolating or looking up values in textures that are not colors and combining them in different ways was an important developmental step. Programmers could achieve elaborate effects by combining multitexturing with more flexible ways of accessing data. For example, dot-product mapping can be extended to generate specular highlights by also transforming and interpolating the relative direction of the eye vector per vertex and factoring in its effect.

New algorithms

Perhaps one of the most exciting elements of computer graphics is that some of the ground rules change over time. Physical and perceptual laws are fixed, but what was once an optimal algorithm might fall by the wayside due to changes in hardware, both on the CPU and GPU. This gives the field vibrancy: we must constantly reexamine assumptions, exploit new capabilities, and discard old ways.

As an example, consider a relatively simple problem and two solutions. Say you have a lens flare effect you apply whenever the sun is visible in your scene. You want to attenuate the flare by how much of the sun is visible on the screen, with the sun itself fitting inside a 16×16 pixel area. The traditional solution would be to render the sun into the scene, then read back this screen area to the CPU and count the number of pixels that have the sun's color, then use this proportion to attenuate the lens flare's brightness. However, as Maughan[3] points out, reading the frame buffer causes a large time loss, as it stalls the graphics pipeline due to the cost of asynchronous communication of screen data back to the CPU. It's actually more efficient to keep everything on the GPU. So a much faster solution is to first render the scene as black and the sun as white to a separate 16×16 image. Use each of this texture's pixels in turn to color 256-point sprites that are rendered to a 1×1 image, so accumulating a level of intensity each time a white sun pixel is visible. This 1×1 image is then used to multiply the incoming lens flare, so attenuating it when rendered.

The effect of the pipeline's architecture and limitations is the major challenge for researchers attempting to use the GPU's incredible speed. One way to think of the CPU and GPU pipeline is as a canoe in a rushing river: Stay in the current by progressing down the pipeline in a normal way and you move along rapidly. Trying to go another less-used direction, for example, back to the CPU, is a much slower proposition.

Researchers have developed a huge number of new techniques due to the increase in newer graphics hardware functionality. With vertex and fragment processors now having an almost unlimited number of steps available, a fair number of traditional rendering algorithms can be done at interactive rates. Here, I discuss just a few of the new ways of using graphics hardware to render images to give a flavor of the way algorithms have changed in response to GPU improvements.

An excellent example of this phenomenon is fur rendering. Lengyel et al.[4] developed numerous techniques to aid in rendering short-haired fur on surfaces. Some methods are straightforward, such as using a level-of-detail algorithm to render a simple single-texture representation of the fur when the model is far away, blending with a more complex model as the object comes nearer.

When close to the furry object, imagine your model consists of not a single surface but a series of concentric shells, with the innermost shell being the original model (essentially the skin). Have the other shells capture the location of the hairs of the fur—that is, imagine each hair poking through the shells and record this hair's location on each shell it pierces. If you now rendered these shells as semitransparent textured polygons in a consistent back-to-front order, you get a furry effect.

Using the vertex processor, it's simple to make fairly reasonable concentric shells. For each shell, the model is sent through a vertex processor that moves all the polygon vertices outward a distance along each vertex normal. Doing so by regular steps gives a set of concentric models, all generated from a single model.

However, this technique does not look convincing around the silhouette edges, as the gaps between the textured shells make the fur strands break up and fade out. This problem is addressed by adding fins along the silhouette edges of the model and texturing these with a fur texture seen from the side. Finding the silhouette edge on the CPU can be an expensive operation, but can be solved by a clever use of the vertex processor: Send all edges of the mesh through the pipeline, with the two

neighboring polygon normals included in the data sent. If the vertex processor finds that edge is a silhouette edge (because one of its polygon normals faces the eye and the other faces away), that edge should be extruded into a quadrilateral fin and rendered.

Since vertex processors cannot create new vertices, each edge has to be sent as two coincident edges to form a zero-area quadrilateral and extruded as needed. Edges not extruded affect nothing, as these quadrilaterals cover no pixels. This all sounds complex, but because of the fantastically high vertex transform and fill rates of modern GPUs it all happens extremely rapidly (see Figure 4).

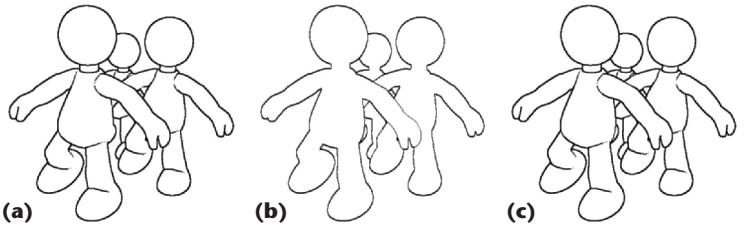
DirectX 10, which will be released in 2006, has a new element called the geometry shader, which can process data coming from the vertex processor. A geometry shader can create new vertices. So, instead of having to store degenerate quadrilaterals for every edge, just the edge itself will need to be sent down the pipeline. If the edge is a silhouette edge, the shader generates two more vertices and creates a quadrilateral.

One interactive application area that has opened up with graphics hardware advances is image processing. Since a fragment processor can access a single texture multiple times for a single fragment and process the results, we can perform operations such as filtering and edge detection rapidly—for example, see Figure 5. What has been done here is to take a traditionally shaded scene and render it with a nonphotorealistic cel-shading effect. This effect results from rendering different versions of the scene, one with a different color per object, another that stores the normal at each pixel, along with a z-buffer depth image. Painting any object, normal, or z-depth discontinuity black creates a cel-shaded version of the scene. An additional pass that samples each pixel's neighbors and outputs the darkest value found can increase line thickness.

More interesting still is the realization that we can use textures to store 3D geometry. One area of research has been to ray-trace scenes by using the GPU to test sets of rays—each represented by an origin and direction vector, each of which in turn is a pixel's worth of data in a texture—against triangles, spheres, or other primitives—which again can each be stored as a few pixels' worth of data. In 2005 two open source projects have implemented ray tracing on the GPU (visit http://gpgpu.org). This research pushes the limits of what is possible on a GPU, and the performance is often at best comparable to that of CPU-side ray tracers. Recent work by Woop et al.[5] explores how some relatively small additions to current GPU hardware could make it work better as a ray-tracing engine.

Lighting and shading advances

Vertex and fragment processing, dependent texture reads, floating-point math, and other graphics hardware advances have made a plethora of new techniques available for use at interactive rates. In particular, the research area of soft shadows has seen a large amount of activity. A thorough survey article by Hasenfratz et al.[6] summarizes work in this field up through 2003, though there has been noticeable activity since then. Other rendering effects have been developed, such as



5 Cel-shaded cartoon style of a model by using edge detection: (a) image has edges found from a normal map, (b) image from z-depth difference, and (c) the thickened composite.
(Courtesy of Jason Mitchell, ATI Technologies.)

glows, volumetric fog, various kinds of reflection and refractions, as well as a wide range of nonphotorealistic effects. Algorithms for more realistic depictions of materials such as skin, water, and woven cloth have been tailored for interactive computation. Exhaustive coverage of all of these effects is well beyond the space limits of this magazine. The goal here is to show how translating algorithms from the CPU to the GPU can result in new ways of rendering images.

Toward this end, it's worth examining the evolution of basic ideas behind the new field of precomputed radiance transfer.[7] PRT attacks a difficult problem in the area of interactive global illumination: how to shade an object lit with arbitrarily complex lighting, properly accounting for shadows and interreflections among surfaces, with the object and lighting changing over time, and all updating at interactive rates. Previous work had attacked combinations of these elements—for example, various soft-shadow algorithms simulate area light sources, but not lighting from, say, a surrounding environment map.

PRT builds on a few different concepts. These include environment mapping, irradiance mapping, spherical harmonics, and ambient occlusion. I'll explain each of these in turn, as this progression follows how theory and practice have evolved over the past few years.

Another way that we can apply a texture to a surface is as an environment or reflection map. Technically, a reflection map is an environment map modified by the surface's attributes, but the two terms are often used interchangeably. The texture itself is a representation of the environment surrounding the objects in the scene. This texture is accessed by computing a reflection vector from each point on the surface and mapping that direction to a location on the texture, making the surface appear shiny. Using an environment map is a good approximation of mirror reflection if the shiny object is far enough away from the surrounding environment, so that the reflector's location has relatively little effect on what is reflected and only the surface normal is needed.

Originally, spherical texture mapping was the only environment mapping type supported in graphics hardware. It could also be supported on the CPU by computing and converting reflection vectors into vertex texture coordinates and interpolating these. A spherical environment map is equivalent to a view of the environment as seen in a mirrored ball. In fact, many real-world environment maps are generated by photographing the

Courtesy of Paul Debevec

6 Spherical environment map of St. Peter's Basilica.

Courtesy of Kasper Høy Nielsen

7 Environment map on the sphere, along with flipping the scene "through the looking-glass," used recursively to create reflections of reflections.

equivalent of a shiny silver ball such as one found on a Christmas tree (see Figure 6).

In 1999, Nvidia introduced the GeForce 256, the first consumer-level (that is, game) card to include cubic environment mapping. Instead of a single texture capturing the environment, the company used six, one on each face of a cube. Imagine an observer looking around from some location. To capture exactly what the world looks like from this single point, he or she can take six photos with a 90-degree field of view: up/down/right/left/front/back. To access this set of images, use the reflection direction vector to choose the correct texture among these six and find the pixel data in the corresponding direction.

While we could use this method with sets of real-world photographs, its greater strengths come from its symmetry and its use in synthetic environments. First, the method works well for any view direction. In contrast, sphere mapping normally forces the sphere texture to face the eye, to avoid distortion and other artifacts. The other great advantage is that the graphics hardware itself can easily create, on the fly, the cube map's six faces. For example, in an automobile simulation, as the car moves, we can render the surrounding environment into a cubic texture and then use the resulting map for reflections off the car's body. Even reflections of reflections can be simulated, using environment maps on nearby shiny objects when generating a new cube map. See Figure 7 for an example, which is combined with the technique of creating planar reflections by mirroring the scene and rendering it through each reflective plane.

An interesting interaction between cubic reflection maps and dot product maps illustrates the beginning of a trend: the use of textures for function computation. One problem encountered with interpolating data between vertices is that vector length is not maintained. That is, if you linearly interpolate between the coordinates of two unit vectors, such as the light vectors used in dot-product bump mapping, any interpolated vector will be shorter than the original vectors. Interpolating in this way can give noticeable shading artifacts. A clever technique to renormalize the vector is to use a special cubic map, one in which each location's data stores a normalized version of the direction vector pointing from the viewer to that location. Feeding this cubic texture an unnormalized vector will yield a normalized vector in the same direction, which we can then use with the dot-product normal retrieved to perform shading correctly. This renormalization method is no longer necessary in newer GPUs, as renormalization can now be done efficiently inside a fragment processor. However, the underlying concept is still valid, that texture lookup can approximate many functions that the graphics hardware might not directly support.

The idea behind an environment map is that each pixel in it captures the radiance coming from a specific direction—what would be seen reflected off a perfect mirror. By filtering the environment map in advance, essentially making it blurry, we can then use the same map to give a surface a roughened, glossy appearance instead of a mirrored one. This method is not particularly physically accurate, but can be perceptually convincing.

We can access a reflection map by using the direction from the eye to the surface and the surface normal itself. Shininess is view dependent. The diffuse component of shading is view independent. So, given a set of lights sufficiently far away, the light contribution depends entirely on the diffuse surface's normal. Imagine an object surrounded by an environment map. Finding the direct lighting at each point on the object is a straightforward, though somewhat arduous, task: Treat each texel on the environment map as a light source. For each vertex, the effect of each texel is weighted by the cosine of the angle between the vertex normal and the direction and solid angle of the texel. Summed up, this is the contribution of the environment map to a surface location with a particular normal direction. By doing so, all possible diffuse illumination values from a given environment map can be precomputed and stored in another environment map, accessed by the surface normal direction (see Figure 8).

This type of texture is called an irradiance map. It's essentially a filtered environment map that records the amount of light coming from a hemisphere over the surface, weighted by the angle away from the normal. Dur-

ing rendering, the surface's normal accesses this map and retrieves how much each light in the environment affects the surface at this orientation. This is an extremely useful technique, as a single map can represent elaborate lighting effects from the surrounding environment.

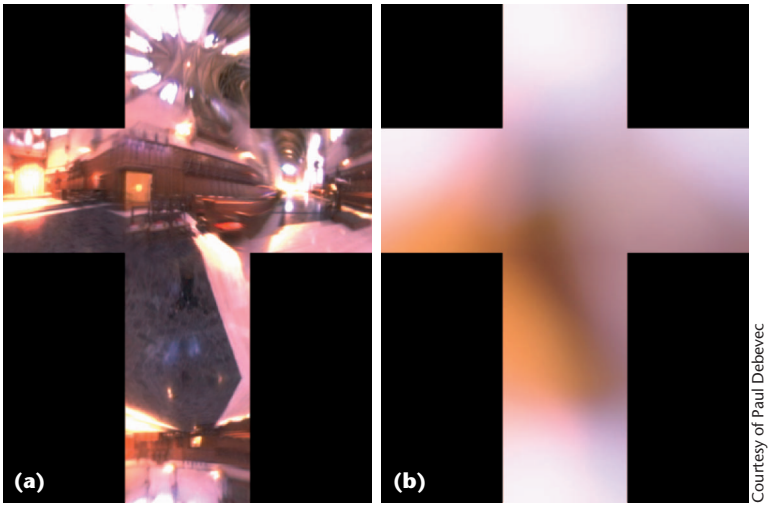
Ramamoorthi and Hanrahan[8] realized in 2001 that spherical harmonics could represent irradiance maps with fairly high fidelity. An entire irradiance map could be represented by 9 RGB triplets and be accurate to within an average of 1 percent error of the original. This technique gives a huge (though lossy) compression factor over storing the original irradiance map as a texture. Determining the light contribution for a particular surface normal is done by evaluating a quadratic polynomial in the normal's coordinates.

This approximation works because the original irradiance map is normally a slowly varying function with few high frequencies; in other words, it looks extremely blurry. Spherical harmonics are something of a spherical analogue of a Fourier series. Just as a series of scaled and shifted sine waves of different frequencies can reconstruct a 1D signal, a series of spherical harmonics can approximate an image on a sphere. Also similar to Fourier series, if the image has no sharp changes, that is, no high frequencies, then it can be represented by fewer terms in the sequence of spherical harmonics.

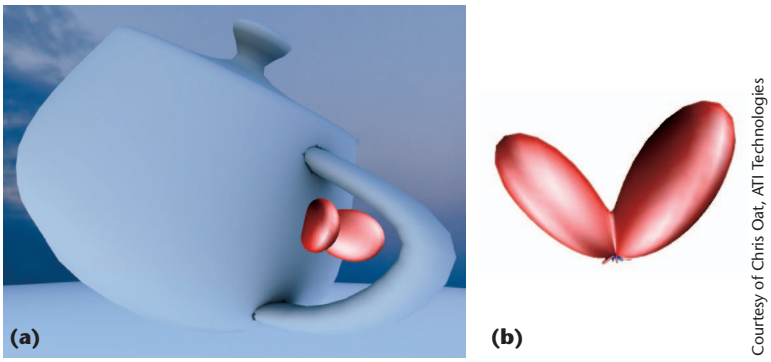
Another concept related to precomputed radiance transfer is the idea of ambient occlusion. In its simplest form, ambient occlusion uses a ray-tracing preprocess to imitate the effect of self-shadowing on a surface. Specifically, a set of rays representing a hemisphere is sent out from some point on the surface. If a ray hits something nearby, light from that general direction is likely to be blocked; if the ray hits nothing on the object, light is likely to be unblocked. We can sum up the rays' contributions in some fashion and obtain a rough estimate of how much light is blocked from the environment. For interactive rendering, this attenuation factor is stored per vertex and used to lessen the effect of lighting as needed. The net effect is to give more definition and realism to cracks and crevices. For example, areas around the ears and nose of a head model will become darker.

However, for the most part this technique performs a wild guess. Areas with some self-shadowing are darker, but they are darkened the same amount regardless of changing lighting conditions, even when a single light is directly illuminating them. At best, the effect is a weak approximation of reality.

PRT draws its strength from all of these ideas and more. As a preprocess, a sampling of each vertex on a model with rays is done similarly to ambient occlusion, but with a different goal in mind. Instead of a single number, what is created for each vertex is a spherical harmonic representation. This representation approximates the way that radiance would be transferred to the surface. It's independent of the actual lighting conditions. A vertex that is entirely unoccluded would have a spherical harmonic representation that approximates a cosine distribution for a standard diffuse surface—that is, the classic Lambertian diffuse term. Occluded vertices have a different set of spherical harmonic values, representing the fact that parts of their view of the hemisphere above them is occluded. Figure 9 shows an example.



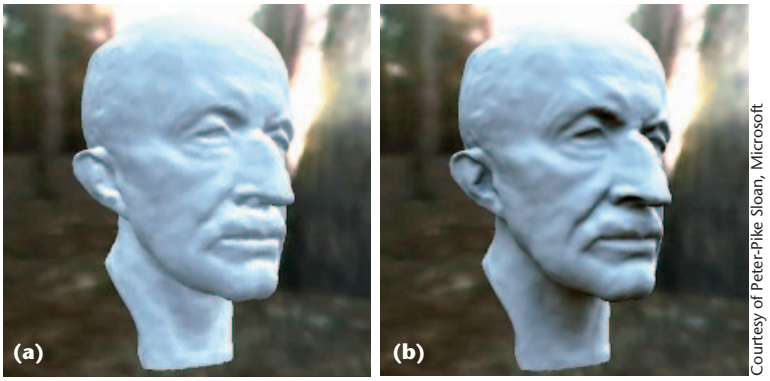
8 (a) Cubic environment map of Grace Cathedral and (b) its corresponding irradiance map.



9 Spherical harmonic radiance transfer function created for (a) a single point under the handle. (b) The crease is due to occlusion from the handle.

Having spherical harmonics represent purely direct illumination would be a poor approximation since there are sharp discontinuities in the function due to self-occlusion. By computing the effect of interreflection among surfaces, these discontinuities are reduced and so spherical harmonics becomes a more reasonable approximation. We can compute interreflection by shooting more rays from each vertex and computing the radiance transfer for the places hit on the surface, by interpolating their transfer values. This result is then factored into each vertex, modifying its spherical harmonic values to now include how radiance is transferred by bouncing off occluding surfaces. Repeat this process any number of times to compute light from two, three, or more interreflections.

With a set of spherical harmonics representing the radiance transfer at each vertex, and a spherical harmonic representation of the environment map itself, we can compute the lighting of the object instantly (see Figure 10 on the next page for an example). Combining the two spherical harmonic sets at each vertex by a simple series



10 **Head rendered (a) with standard diffuse shading and (b) with precomputed radiance transfer.**

of dot products yields the vertex color. This basic property, that we can combine the two spherical harmonic representations rapidly on the fly on the GPU, gives rise to a number of extensions. For example, rotating the object is relatively quick, as the spherical harmonics do not have to be recomputed from scratch but rather can be transformed to a new set of spherical harmonics.

Object translation can also be handled. For example, instead of a single environment map representing the lights far away, environment maps are generated for specific locations in space. Each environment map is then more of a light field, a representation of all the light passing through a point in space. We can convert these representations to irradiance maps, which can then be converted to spherical harmonic sets. As the object moves through space, a spherical harmonic set representing the approximate irradiance map is interpolated from the surrounding sets and used as before.[9]

Research in this area has proceeded apace. Shiny objects can also be dealt with interactively by extending the basic idea, but because of view dependency the storage and precomputation requirements go up considerably. Local light sources can be made to work with the method, as well as deformable surfaces, again with higher costs. Subsurface scattering, a phenomenon where photons travel some distance through the material (examples include skin and marble), can also be simulated using PRT techniques. In fact, any light-transfer function, such as dispersion or caustics, that can be computed can then be compressed and stored using some set of basis functions.

Using a spherical harmonic basis for illumination is just one way to store radiance transfer. For example, while Bungie Studio's game *Halo 2* used a spherical harmonic basis, Valve's *Half-Life 2* uses a more standard Cartesian basis using a method called the ambient cube.[10] What is exciting about all this work is that it's a different way of thinking about the problem, one that leverages the capabilities of the GPU.

Improved bump mapping

To conclude this brief tour of new shading algorithms, I will revisit the problem of making a surface appear bumpy. Dot-product bump mapping is a technique that was first described around 1998 and which started appearing in games in 2004. The delay is understandable as it took awhile for the graphics hardware to become relatively common and fast enough for software providers to create tools to deal with such textures, and for developers to integrate this functionality into their production processes. In the meantime, several new schemes for creating more realistic bumps on surfaces have emerged. The two I will discuss here are parallax-occlusion mapping and vertex textures.

A problem with bump mapping in general is that the bumps never block each other. If you look along a real brick wall, for example, at some angle you will not see the mortar between the bricks. A bump map of the wall will never show this type of occlusion, as it merely varies the normal. It would be better to have the bumps actually affect which location on the surface is rendered at each pixel.

A traditional solution used in high-end, offline rendering is displacement mapping. Imagine that a square with a bump map is tessellated into a grid of triangles and each vertex is varied in height, instead of being geometrically flat. This mesh represents a truly geometrically bumpy surface. However, this method is a poor fit for GPUs, as part of their design results in a sweet spot of pixels per triangle. A huge number of tiny triangles will force some part of the geometry section of the pipeline to be the bottleneck, and the enormous speed of the rasterizing section will then be mostly unused and hence wasted. As such, other approaches have been explored to render self-occlusion more efficiently. The first GPU-oriented algorithm attacking this problem was from Kaneko et al. in 2001,[11] and a number of others have been developed since then.

Brawley and Tatarchuk's parallax-occlusion-mapping approach is relatively straightforward to describe and gives a flavor of how fragment shaders and texture access can be used to improve image appearance.[12] Consider a square with a height field applied as a texture. Say the lowest altitude of this height field corresponded to the location of the original, flat square. Any points at this lowest altitude are on the original square, regardless of viewing angle.

Now imagine looking at the original square at a shallow angle through a single pixel. There is a lowest altitude point that is visible on this square. A real, geometric height field usually has the effect of occluding this lowest point with some geometry that is closer to the eye. In parallax-occlusion mapping, the square (or any other surface) has a texture applied to it that holds a height field. The eye direction is projected onto this height-field texture. Where this projected vector falls forms a line on the height-field texture. By walking this line and checking the height of the height field at each step, we can determine the height-field location that actually blocks the pixel (see Figure 11).

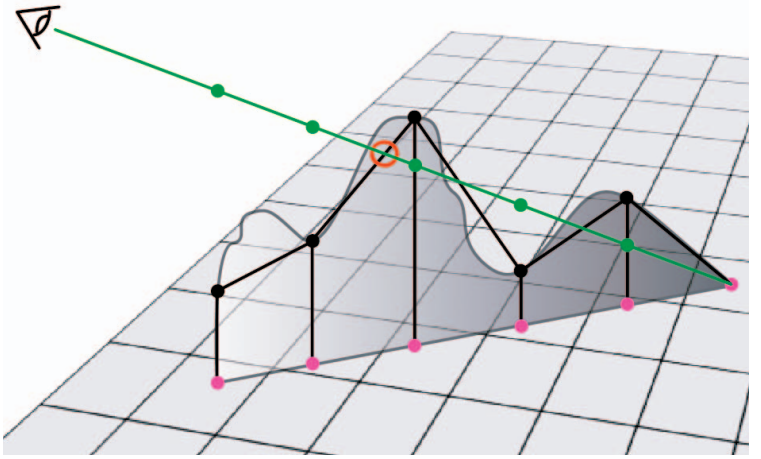
The angle between the eye and the surface (as well as the relative height range of the height field) determines the length of the projected eye vector. That is, at shallow angles to the surface, the number of height-field texture locations that could influence the result is increased; at steep angles (looking down on the square), fewer height-field locations are needed along this line.

In a traditional ray tracer, a ray would walk through the grid forming the height field, checking intersection with the relevant geometry. Instead, this GPU-based algorithm tests a fixed number of texture samples along the projected vector. Each texture location is retrieved and processed to determine if it's visible, and the interpolated location and corresponding information is used to shade the surface instead of the original texture location. This algorithm is a clever mix of ideas, playing to the strengths of current GPUs by using more fragment processing on a single surface. The length of the vector usually limits the number of height-field locations that could influence the result to something manageable. The texturing capabilities of the GPU are then used to sample these height-field locations. This same technique can make the bumpy surface cast shadows onto itself (see Figure 12).

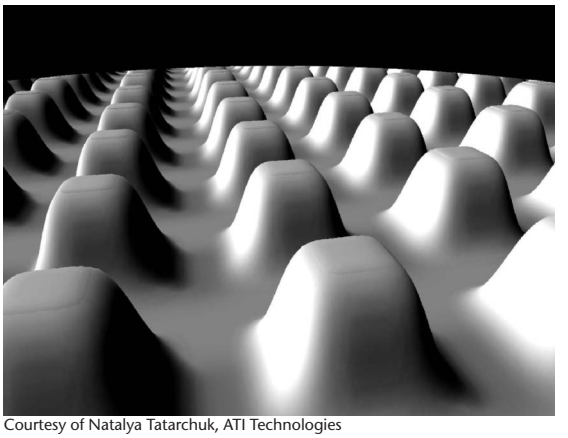
This technique has some limitations, though. At shallow angles, the number of height-field locations that should be sampled might exceed the number of texture retrievals that the fragment shader can handle. However, we can extend the algorithm to adaptively sample the surface in these cases.[13] Also, as there is no real displacement of the surface itself, the illusion breaks down along the silhouette edges of objects, which will show the original surface's smooth outlines. Oliveira and Policarpo attack this problem by warping the ray's sample path by the local curvature of the surface and discarding any fragments found to be off of the surface.[14]

A 2004 graphics hardware development enables vertex processors to access texture data. This functionality is suited for true displacement mapping: each vertex on a mesh can be shifted by a height-field texture. Doing so solves the silhouette edge problem by brute force. Data transfer and storage costs are also minimized by using a simple height-field texture. However, this functionality comes at the expense of needing state-of-the-art graphics hardware to perform this algorithm, as well as the need for more vertex processing power.

This new hardware feature also means that now fragment processors can render to textures that the vertex processors can then access. Collisions between objects can truly deform the objects simply by painting the colliding areas onto the texture.[15] Also, fragment processors can generate *XYZ* coordinate values, which the vertex processor then accesses to generate sprites for particle systems.[16] Another idea is to use the vertex texture as a guide for how and where to place actual geometric elements, such as grass and leaves.[17] There are undoubtedly more applications that will arise from this new functionality as programmers come to understand its capabilities. The key insight is that this feature closes the loop between the high computational power of the fragment units and the addressing capability of the rasterizer, and allows more complex operations. For instance, the GPU can compute something in the fragment unit, and then in another pass write that data into an arbitrary pixel using a vertex program. This type of scatter operation is important in building GPU-based data structures. For example, Purcell et al. use the scatter operation to perform photon mapping on the GPU.[18]



11 Parallax-occlusion mapping. The green eye ray is projected onto the surface plane, which is sampled at regular intervals (the pink dots) and heights retrieved. The algorithm finds the first intersection of the eye ray with the black line segments approximating the height field.



Courtesy of Natalya Tatarchuk, ATI Technologies

12 Height-field rendering using parallax-occlusion mapping.

Further up the pipeline

The major focus of this tour is how vertex and fragment processors, combined with rapid texture access and filtering, have brought about changes in the way we think about and program interactive applications. It's worth revisiting the evolution of the pipeline at this point to discuss a few graphics hardware developments in other areas.

As noted earlier, the evolution of the GPU started from the end of the pipeline and went backward. First display, then rasterization, then triangle setup, then geometric operations on incoming vertex data were each moved to specialized graphics hardware. The question is always what makes sense to leave on the flexible but relatively slow CPU, and what is worth committing to fixed but fast GPU functionality. Adding programmability to the GPU changes this balance, with specialized processor functions suited for use with a wider range of tasks.

Additional memory also affects this mix. Memory was once simply for the final image that would be displayed. Usability improved with the addition of double-buffering, but so did memory requirements. Then z-buffer memory was added for 3D computation, along with stencil buffer memory (today usually 8 bits, with the

z-buffer taking the other 24 bits in the 32-bit word). By the late 1990s, memory became a more flexible resource, something that could be doled out to provide various rendering modes, trading off screen resolution, color and z-buffer precision, and so on. Eventually there was enough memory available that all current cards can have a double-buffered display with z-buffer and stencil buffer at full 24-bit color at any resolution.

Interactive graphics is sometimes considered a bottomless pit when it comes to resources, and memory is not exempted. Beyond the memory needed for the z-buffer, memory is also required for storing textures. To save texture storage space (and, more importantly, bandwidth), a graphics hardware-friendly class of compression algorithms, called DXTC in DirectX, have been developed. Finally, geometry data can also be moved to the GPU's memory and be reused. Vertex and fragment processors can help vary the appearance of this data from frame to frame or object to object.

For display, the only drawback of using more memory is that higher resolution and color depth cost performance. While processors have continued to increase in speed at a rate in keeping with Moore's law, and graphics pipelines have actually exceeded this rate,[19] memory bandwidth and latency have not kept up. Bandwidth is the rate at which data can be transferred, and latency is the time between request and retrieval. While the processor capability has been going up 71 percent per year, DRAM bandwidth is increasing about 25 percent, and latency a mere 5 percent.[20] What this means is that with the latest GPU processors, you can do about 12 floating-point instructions in the time you can access a single word of floating-point texture data. Processing power is becoming less and less the bottleneck as these trends continue.

Beyond improving the efficiency of memory access itself, a number of techniques can avoid or minimize memory access when possible. A z-buffer operation on a single fragment causes the pixel's z-depth memory to be read, compared to the incoming z-depth value, and then possibly replaced by this new value, as well as affecting the color and stencil buffers. If the incoming z-depth value was not closer—in other words, the triangle being rendered is not visible—the operation of testing the z-depth has no actual effect on the image.

The ideal situation would be one in which the pipeline processes only those triangles visible in a scene. On a triangle-by-triangle level the pipeline has long provided clipping (always needed) and backface culling (used to throw away polygons in solid objects that face away from the viewer). The third area, not exploited until recently, is occlusion culling.

One passive example of this sort of occlusion is ATI's HyperZ technology (Nvidia has its own version), which avoids touching memory by treating sets of pixels as tiles. For example, call an 8×8 set of pixels a tile, and think of the screen as being made of a grid of tiles. If the rasterizer determines that a tile overlapping the polygon to be drawn is already entirely covered by nearer objects, then the part of the polygon in this tile does not have to be rasterized but can be rejected as a whole, saving unneeded fragment processing and memory costs. Bandwidth is also saved by losslessly compressing the z values in the tile itself. More actively, if a developer renders the scene roughly sorted from front to back, he or she will further avoid performing pixel operations unnecessarily, as the closer objects will help hide those further away. This technology also circumvents wasting time spent performing a clear of each pixel at the beginning of rendering a frame, instead marking tiles as cleared and treating them appropriately when accessed. Screen clearing might sound like a trivial operation, but avoiding it saves considerable bandwidth.

HyperZ culling gains efficiency by dealing with each triangle just before it is set up to be filled. However, the triangle still has to be sent to the GPU. The fastest triangle to render is the one never sent down the pipeline. Toward this end, techniques have been developed to deal with groups of triangles. On the CPU it's common to use frustum culling, in which each object is given a bounding box or sphere. If this bounding volume is outside the view, none of the objects inside the volume need to be sent down the pipeline. The technique is further improved by nesting volumes inside larger volumes in a hierarchy, so that such culling can potentially ignore vast amounts of data with a few simple tests.

However, such CPU processing does not remove any objects inside the frustum that are hidden by other objects. GPUs have support for higher-level occlusion culling in which an object such as a bounding box can be sent through the pipeline to check whether any part of the box is visible. If the box is not visible, all the objects inside the box do not have to be sent down the pipeline. The problem is that reading back object status from the GPU to the CPU is expensive. Methods such as batching results of some tests into a single query, and the faster transfer speed of PCI Express, make such techniques more feasible, especially for very large data sets.[21] This sort of processing is at the limits of how much of the pipeline the current GPU can handle, and it needs to do so in conjunction with the CPU's guidance.

The future

Moving model data onto the GPU for reuse recalls how graphics acceleration worked decades ago, in which a vector display system was loaded with a list of lines and dots to display, and the user would control the transforms applied and the visibility of objects. It will be interesting to see whether the principle of the wheel of reincarnation will apply someday to current systems. This idea, first noted back in 1968 by Myer and Sutherland,[22] can be seen when a new piece of peripheral hardware is developed and elaborated. This system's functionality is eventually folded back into the controller's domain, and the cycle begins anew with another piece of peripheral hardware.

That said, the GPU, because of its speed advantages, is often thought of in different terms than the CPU. A GPU is not a serial processor, but is rather a data flow or stream processor. The appearance of data at the beginning of the pipeline causes computation to occur and requires a limited set of inputs to perform the computation. This different processing model lends itself to problems where each datum is affected by only nearby data. One active area of research is how to apply the GPU to nonrendering prob-

lems of this type, such as fluid-flow computation and collision detection (visit http://gpgpu.org).

Another area of graphics hardware design that looks likely to progress further is surface representation. The current hardware pipeline deals with vertices as disconnected entities, but the new geometry shader changes the rules. Some experimentation has been done with on-chip tessellation, such as ATI's N-Patches, in which a single triangle with vertex normals can generate a curved surface. DirectX 10 includes an interface to perform tessellation. However, as yet, no generalized geometry tessellation has been added to the graphics hardware pipeline for surfaces with connectivity among triangles, such as subdivision surfaces. The main question, as with any new feature, is whether it is worth the cost.

The amount of functionality in the GPU has grown enormously in just a few years and understanding what is fast and what operations to avoid will lead to new, efficient ways to solve old problems. As an example, a fixed platform such as the Sony PlayStation took developers a few years to fully comprehend and to take advantage of its architecture. Games produced four years after this console's introduction were considerably more advanced graphically than the original offerings. Unexpected and wonderful new algorithms and improvements are on the way, and anyone with a graphics card and a compiler can help discover them. ■

Acknowledgments

Thanks to Matt Pharr for his extensive comments on two drafts of this article; Peter-Pike Sloan for checking the PRT section; Tomas Akenine-Möller, Chris Seitz, and the four anonymous reviewers for their comments of the final draft; and to all the individuals and companies that granted permission to reprint their images.

References

1. N. Duca, "A Relational Debugging Engine for the Graphics Pipeline," *ACM Trans. Graphics*, vol. 24, no. 3, 2005, pp. 453-463.
2. T. Hudon, *Timeline: 3DFX Revisited*, 2003; http://www.aceshardware.com/read.jsp?id=60000292.
3. C. Maughan, "Texture Masking for Faster Lens Flare," M. DeLoura, ed., *Game Programming Gems 2*, Charles River Media, 2001, pp. 474-480.
4. J. Lengyel et al., "Real-Time Fur over Arbitrary Surfaces," *Proc. Symp. Interactive 3D Graphics*, 2001, pp. 227-232; http://people.csail.mit.edu/ericchan/bib/pdf/p227-lengyel.pdf.
5. S. Woop, J. Schmittler, and P. Slusallek, "RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing," *ACM Trans. Graphics*, vol. 24, no. 3, 2005, pp. 434-444.
6. J.-M. Hasenfratz, "A Survey of Real-Time Soft Shadows Algorithms," *Computer Graphics Forum*, vol. 22, no. 4, 2003, pp. 753-774.
7. P.-P. Sloan, J. Kautz, and J. Snyder, "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments," *ACM Trans. Graphics*, vol. 21, no. 3, 2002, pp. 527-536.
8. R. Ramamoorthi and P. Hanrahan, "An Efficient Representation for Irradiance Environment Maps," *Proc. Siggraph*, ACM Press, 2001, pp. 497-500.
9. C. Oat, "Irradiance Volumes for Games," *Proc. Game Developers Conf.*; http://www.ati.com/developer/gdc/GDC2005_PracticalPRT.pdf.
10. G. McTaggart, "Half-Life 2/Valve Source Shading," *Proc. Game Developers Conf.*, 2004; http://www2.ati.com/developer/gdc/D3DTutorial10_Half-Life2_Shading.pdf.
11. T. Kaneko et al., "Detailed Shape Representation with Parallax Mapping," *Proc. 11th Int'l Conf. Artificial Reality and Telexistence* (ICAT), 2001, pp. 205-208; http://vrsj.t.u-tokyo.ac.jp/ic-at/icat2003/papers/01205.pdf.
12. Z. Brawley and N. Tatarchuk, "Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing," *ShaderX^3*, W. Engel, ed., Charles River Media, 2004, pp. 135-154.
13. N. Tatarchuk, *Dynamic Image-Space Displacement Mapping with Silhouette Antialiasing via Parallax Occlusion Mapping*, 2005; http://www.ati.com/developer/techpapers.html.
14. M.M. Oliveira and F. Policarpo, *An Efficient Representation for Surface Details*, tech. report RP-351, Universidade Federal do Rio Grande do Sul (UFRGS), 2005.
15. P. Wrotek, A. Rice, and M. McGuire, "Real-Time Collision Deformations Using Graphics Hardware," *J. Graphics Tools*, to be published.
16. P. Kipfer, M. Segal, and R. Westermann, "UberFlow: A GPU-Based Particle Engine," *Proc. Siggraph/Eurographics Workshop Graphics Hardware*, 2004, pp. 115-122; http://wwwcg.in.tum.de/research/data/publications/eghw04.pdf.
17. O. Tresniak, "Rendering Detailed Outdoor Ground Surfaces on the GPU," *Siggraph Sketch*, 2005; DVD ROM.
18. T.J. Purcell et al., "Photon Mapping on Programmable Graphics Hardware," *Proc. Siggraph/Eurographics Workshop on Graphics Hardware*, 2003, pp. 41-50; http://graphics.lcs.mit.edu/~ericchan/bib/pdf/p41-purcell.pdf.
19. A. Lastra, "All the Triangles in the World," *Proc. Cornell Workshop on Rendering, Perception, and Measurement*, 1999; http://www.graphics.cornell.edu/workshop/talks/complexity/lastra.pdf.
20. J. Owens, "Streaming Architectures and Technology Trends," *GPU Gems 2*, Matt Pharr, ed., Addison-Wesley, 2005, pp. 457-470.
21. M. Wimmer and J. Bittner, "Hard Occlusion Queries Made Useful," *GPU Gems 2*, M. Pharr, ed., Addison-Wesley, 2005, pp. 91-108.
22. T.H. Myer and I.E. Sutherland, "On the Design of Display Processors," *Comm. ACM*, vol. 11, no. 6, 1968, pp. 410-414.

Eric Haines *is a lead software engineer at Autodesk. His research interests include ray tracing, shadow generation, and efficient geometric computations. Haines has an MS from the Program of Computer Graphics at Cornell. He is an editor of the* Journal of Graphics Tools*, online editor for* ACM Transactions on Graphics*, and a member of the IEEE. Contact him at erich@acm.org.*