

# **An Introductory Tour of Interactive Rendering**

by Eric Haines, [erich@acm.org](mailto:erich@acm.org)

[near-final draft, 9/23/05, for IEEE CG&A January/February 2006, p. 76-87]

**Abstract:** The past decade has seen major improvements, in both speed and quality, in the area of interactive rendering. The focus of this article is on the evolution of the consumer-level personal graphics processor, since this is now the standard platform for most researchers developing new algorithms in the field. Instead of a survey approach covering all topics, this article covers the basics and then provides a tour of some parts the field. The goals are a firm understanding of what newer graphics processors provide and a sense of how different algorithms are developed in response to these capabilities.

**Keywords:** graphics processors, GPU, interactive, real-time, shading, texturing

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. With faster speeds and new functionality have come new ways of thinking about how graphics hardware can be used 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 a number of 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 ten years would become a long yet shallow litany of taxonomies and terms. Instead, a selection of topics from various areas is used to illustrate this central theme: the influence of 3D graphics hardware on algorithm design. To begin, the main algorithmic principles used for interactive rendering are outlined. The evolution of the graphics processor pipeline and texturing capabilities are then presented. A few examples of how new techniques have been developed in response to these improved capabilities are given. The focus is then on two particular areas, environmental lighting and bump mapping, showing how algorithms

have developed to give higher quality results while still maintaining interactive rendering rates. Finally, the overall behavior of current graphics hardware is discussed and future possibilities explored.

## **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 towards 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 vs., 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 performed once and reused, its initial cost can be justified by the savings it yields over a number of frames it is 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.

## The Evolving Pipeline

A high-level view of the traditional 3D Z-buffer pipeline is shown in Figure 1. Data from the application in the form of vertices of individual polygons (converted to triangles for simplicity) is first 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.

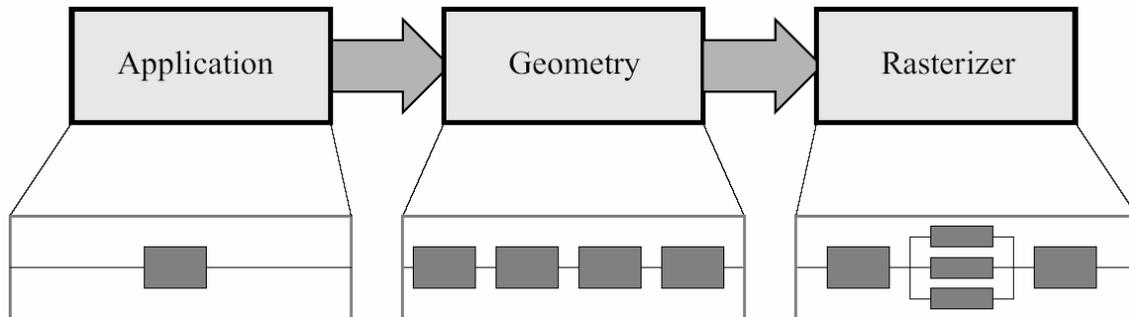


Figure 1. The traditional pipeline. Each of the three main sections of the pipeline can then be further broken down into separate tasks, some of which can be performed in parallel in a SIMD or MIMD fashion.

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 GPU hardware many triangles may be in the pipeline at one time. A single triangle may 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 may make vertex transformation the bottleneck. Parallelism can be applied in a number of ways to improve the performance of a stage in the pipeline. One straightforward method is putting a number of units in parallel to transform vertices or to fill triangle pixels. First-in-first-out (FIFO) buffers are also commonly used between stages to allow temporary backlogs and so avoid stalling processing further up the pipeline.

The evolution of PC graphics cards over the 1990s can be seen in terms of going backwards 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 pre-transformed triangle. In 1999 the transform and lighting part of the pipeline moved onto the GPU (and this was when the term GPU, Graphics Processing Unit, was introduced). During this time some game developers considered this a step backwards, in terms of flexibility, as the CPU itself had no limits on what shading model was used to compute vertex colors. Graphics hardware 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 specialized capabilities.

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 2. These alternate paths were controlled by relatively short programs, commonly called *vertex* and *fragment shaders* (or for DirectX, *pixel shaders*).

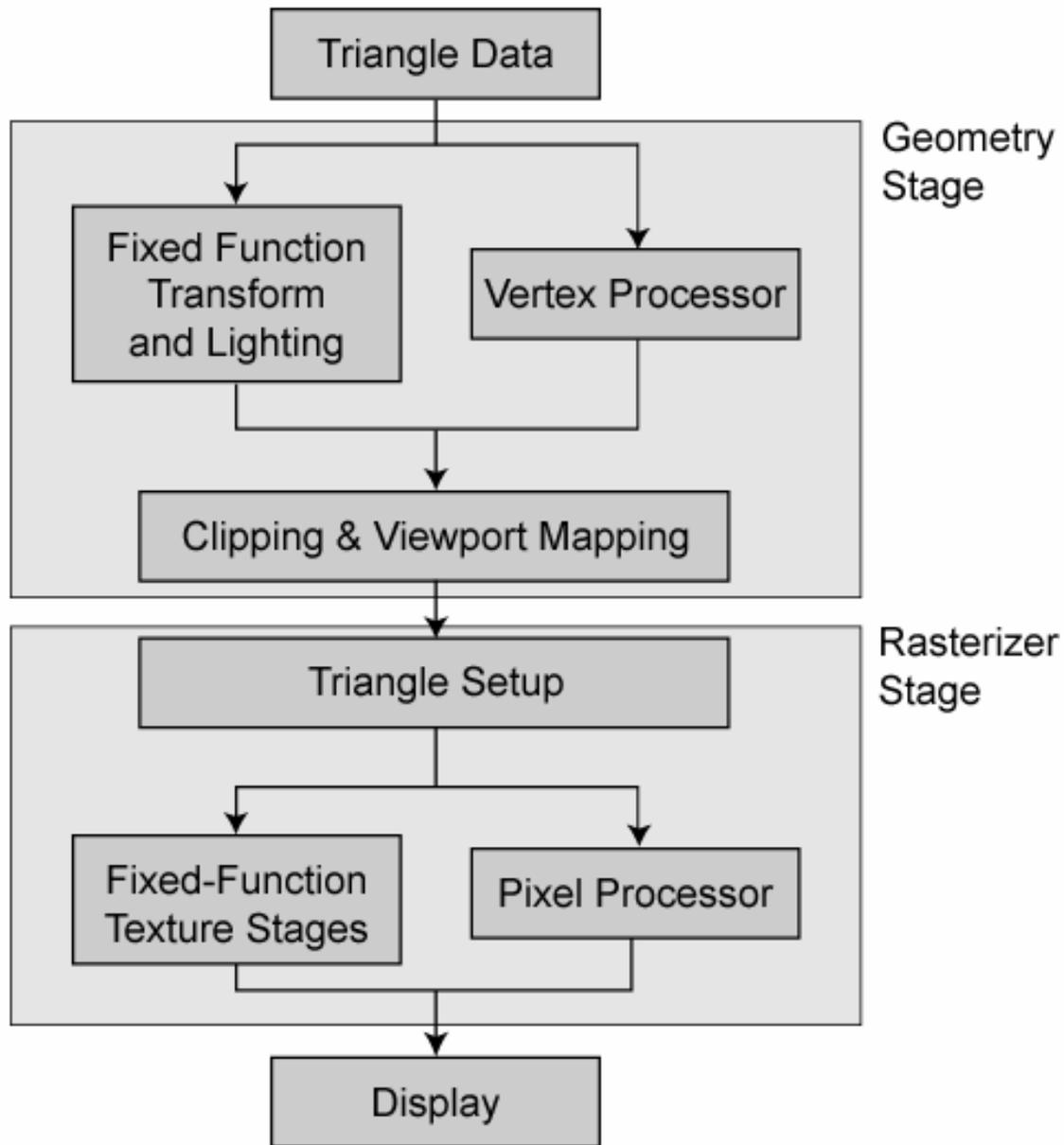


Figure 2. The vertex and fragment processor enhanced pipeline, from the user's point of view. The fixed-function hardware no longer exists in newer GPUs, and the fixed-function API calls are translated into shader programs by the graphics driver.

Instead of fixed-function shading per vertex, the vertex processor can be programmed to perform all sorts of computations. Specifically, a vertex processor works on one vertex of a triangle at a time, independent 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 (e.g. 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 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 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, 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 (a.k.a. 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 surface of the triangle. 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 vs. 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*. The next section discusses this idea in more depth. Older GPUs have limitations on the ability to perform dependent texture reads that have been eliminated in newer processors.

One goal for the designers of fragment and vertex processors 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 (HDR) environment textures can be represented, so 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 multi-pass render. Texture lookups can be used to 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, i.e. that interpolation is linear in object space.

Over the years, the instruction sets of both fragment and vertex processors have been significantly increased and have 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 number of instructions in both shaders has been increased to 65,536 with the introduction of DirectX’s Shader Model 3.0 in 2004.

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 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 can possibly be covered here. For example, most shaders written for 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 the new flexibility of the GPU was controlled by elaborate sets of API calls. This cumbersome interface was soon superseded by assembly language commands that worked on scalars and vectors. In recent years more natural languages such as DirectX 9’s HLSL (high-level shading language) have been developed. Coding support tools such as debuggers and profilers are taken for granted when programming a CPU. With the rise

of the GPU the creation of similar tools is an active area of research and development [Duca05]. 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. This offering is considered by many the 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 4 times faster than its closest competitor [Hudon03].

Around this time new ideas also began to appear, the most significant being multi-pass 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 3.

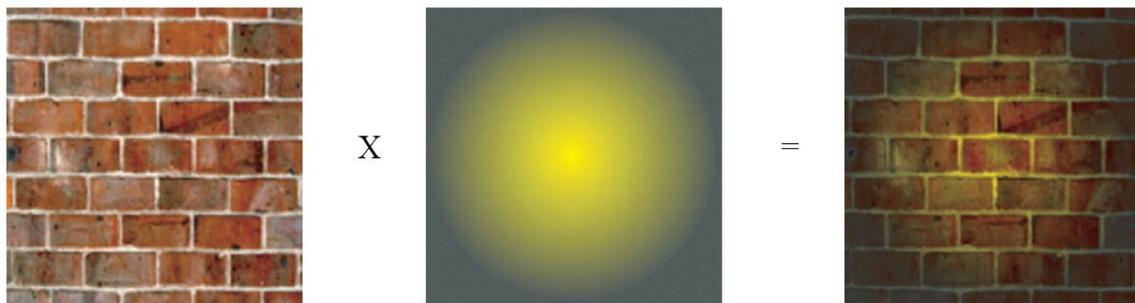


Figure 3. A wall texture is combined with a light map texture to give a lit wall. (Courtesy of J.L. Mitchell, M. Tatro, and I. Bullard)

The idea of applying two textures to the same surface has many other uses: adding decals to objects, adding visual detail to terrain, etc. This concept was general enough that graphics hardware (e.g. 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 3, 4, 6, 8, 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 0-255 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 4.

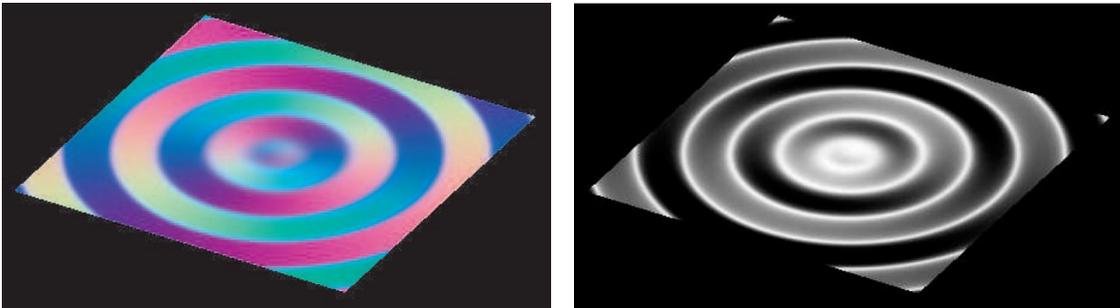


Figure 4. On the left, a normal map's vector values shown as RGB. On right, the results. (Courtesy of NVIDIA Corp.)

Two tangent vectors are also stored for each vertex, specifying how the normal map relates to the surface. In traditional Gouraud interpolation the shade of each vertex is computed, then these colors are interpolated across the surface. For dot product bump mapping the vector to the light is computed 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 towards 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 are used to 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. Elaborate effects

could be achieved 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 may fall by the wayside due to changes in hardware, both on the CPU and GPU. This gives the field vibrancy: assumptions must be constantly reexamined, new capabilities exploited, and old ways discarded.

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 16x16 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 which have the sun's color, then use this proportion to attenuate the lens flare's brightness. However, as Maughan [Maughan2001] 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 is actually much faster 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 16x16 image. Use each of this texture's pixels in turn to color 256 point sprites that are rendered to a 1x1 image, so accumulating a level of intensity each time a white sun pixel is visible. This 1x1 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 incredible speed of the GPU. One way to think of the CPU/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, e.g. back to the CPU, is a much slower proposition.

There are a huge number of new techniques that have been developed due the increase in functionality of newer graphics hardware. 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. Just a very few of the new ways of using graphics hardware to render images are discussed here, to give a flavor of the way algorithms have changed in response to improvements on the GPU.

An excellent example of this phenomenon is fur rendering. Lengyel et al. [Lengyel2000] developed a number of techniques to aid in rendering short-haired fur on surfaces. Some 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 where the hairs of the fur are located; that is, imagine each hair poking through the shells and record the location of this hair 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 is simple to make fairly reasonable concentric shells. For each shell, the model is sent through a vertex processor that moves all the polygon vertices outwards 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 an edge is found by the vertex processor to be 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 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 5.

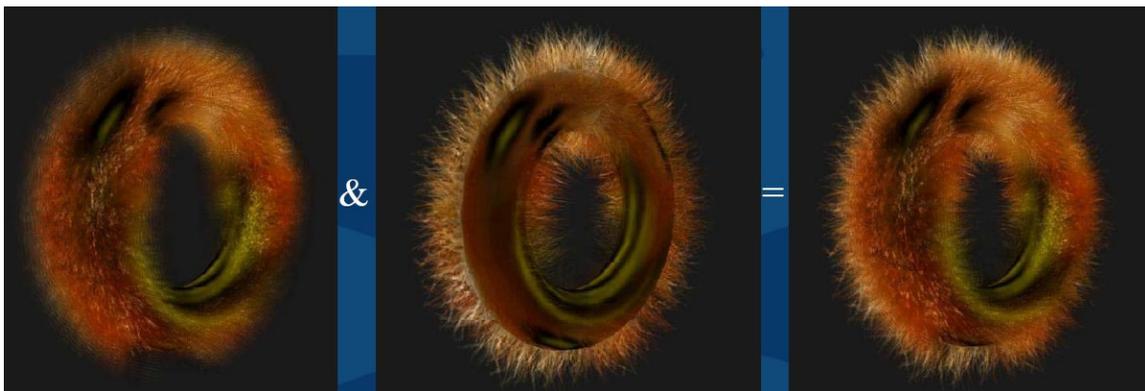


Figure 5. Concentric shells of layered fur textures and silhouette edge fins produce a convincing simulation of fur. (Courtesy of John Isidoro, ATI Technologies Inc.)

DirectX 10, which will be released by 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 found to be a silhouette edge, two more vertices are generated and a quadrilateral is created.

One interactive application area that has opened up with graphics hardware advances is image processing. Since a single texture can be accessed multiple times for a single fragment and the results can be processed with a fragment processor, operations such as filtering and edge detection can be performed rapidly. As an example, see Figure 6. What has been done here is to take a traditionally shaded scene and render it with a non-photorealistic cel-shading effect. This effect is performed by rendering different versions of the scene, one with a different color per object, another which stores the normal at each pixel, along with a Z-buffer depth “image”. By painting any object, normal, or Z-depth discontinuity black, a cel shaded version of the scene is created. Line thickness can be increased by an additional pass that samples each pixel’s neighbors and outputs the darkest value found.

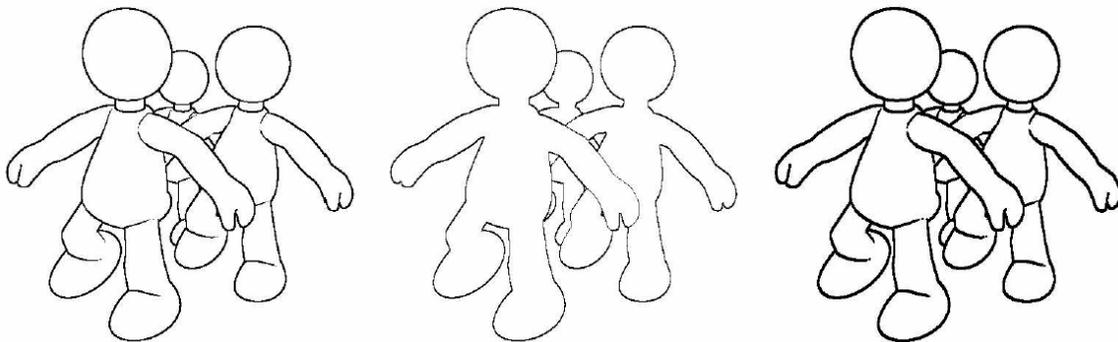


Figure 6. Cel-shaded cartoon style of a model by using edge detection. The left image has edges found from a normal map, the middle image from z-depth difference, and the right is the thickened composite (*Courtesy of Jason Mitchell, ATI Technologies Inc.*)

More interesting still is the realization that textures can be used 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 the past year two different open source projects have implemented ray tracing on the GPU (visit [gpgpu.org](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. [Woop05] 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. [Hasenfratz03] summarizes work in this field up through 2003, though there has been noticeable activity since then. Other rendering effects have been developed, such as glows, volumetric fog, various kinds of reflection and refractions, as well as a wide range of non-photorealistic effects. Algorithms for more realistic depictions of a wide range 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 journal. The goal here is to show how translation of algorithms from the CPU to the GPU can result in new ways of rendering images.

Towards this end, it is worth examining the evolution of basic ideas behind the new field of precomputed radiance transfer (PRT) [Sloan2002]. 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 various combinations of these elements, e.g. various soft-shadow algorithms have been developed for area light sources, but not for lighting from, say, a surrounding environment map.

There are a few different concepts that PRT builds upon. These include environment mapping, irradiance mapping, spherical harmonics, and ambient occlusion. Each of these will be explained in turn, as this progression follows how theory and practice have evolved over the past few years.

Another way that a texture can be applied 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, so 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 equivalent of a shiny Christmas-tree silver ball. See Figure 7.



Figure 7: Spherical environment map of St. Peter's Basilica. (Courtesy of Paul Debevec)

In 1999 NVIDIA introduced the GeForce 256, the first consumer-level (i.e. game) card to include cubic environment mapping. Instead of a single texture capturing the environment, six were used, 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 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 this method could be used 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 six faces of the cube map can easily be created on the fly by the graphics hardware itself. For example, in an automobile simulation, as the car moves, the surrounding environment can be rendered into a cubic texture and the resulting map then used 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 8 for an example, which is combined with the technique of creating planar reflections by mirroring the scene and rendering it through each reflective plane.



Figure 8. Environment map on the sphere, along with flipping the scene “through the looking-glass,” used recursively to create reflections of reflections (*Courtesy of Kasper Høy Nielsen*)

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 can then be used with the dot product normal retrieved to perform shading correctly. This renormalization method is no longer necessary in newer GPUs, as this process can now be done efficiently inside a fragment processor. However, the underlying concept is still valid, that texture lookup can approximate many functions that might not be directly supported by the graphics hardware.

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, the same map can then be used 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.

A reflection map is accessed 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 is entirely dependent 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 9.

This type of texture is called an *irradiance map*. It is 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. During 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.

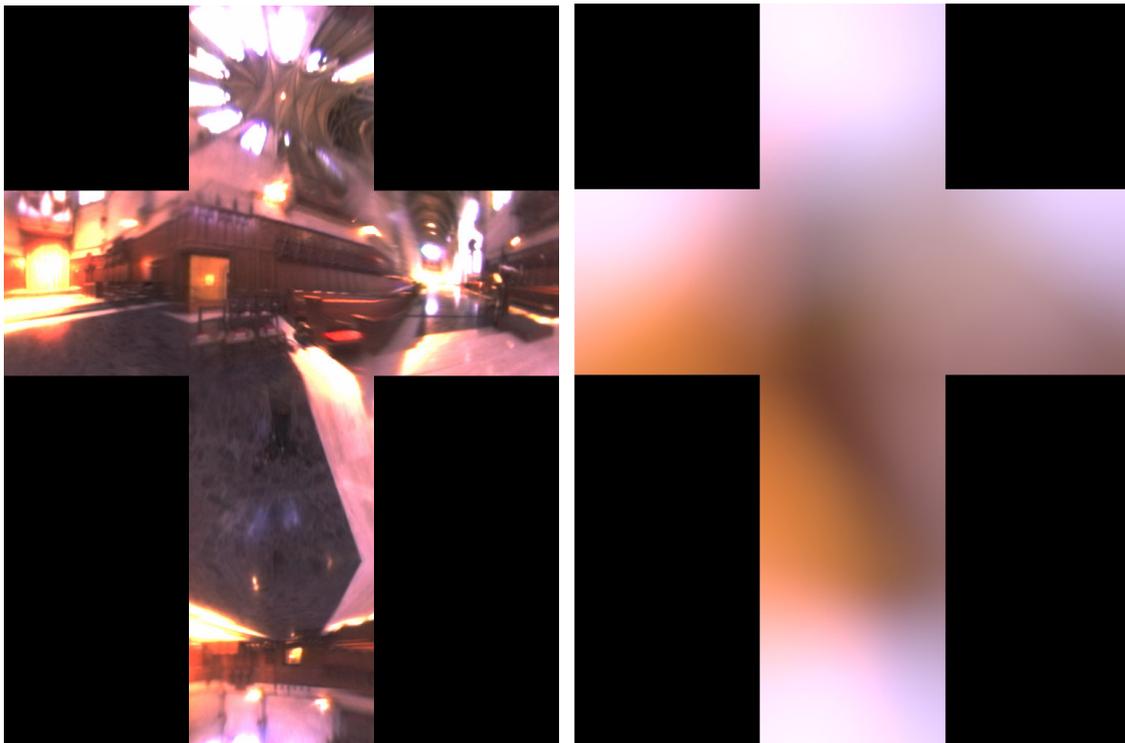


Figure 9. A cubic environment map of Grace Cathedral and its corresponding irradiance map. (Courtesy of Paul Debevec)

Ramamoorthi and Hanrahan [Ramamoorthi01] realized in 2001 that spherical harmonics could be used to 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% 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 coordinates of the normal.

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 1D signal can be reconstructed by a series of scaled and shifted sine waves of different frequencies, an image on a sphere can be approximated by a series of spherical harmonics. Also similar to Fourier series, if the image has no sharp changes, i.e. 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. The rays' contributions are then summed in some fashion and a rough estimate of how much light is blocked from the environment is obtained. 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 is performing a wild guess. Areas with some self-shadowing are darker, but they are darkened the same regardless of changing lighting conditions, even for when a single light is directly illuminating them. At best the effect is a weak approximation to 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 is 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, i.e.

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 10 shows an example.

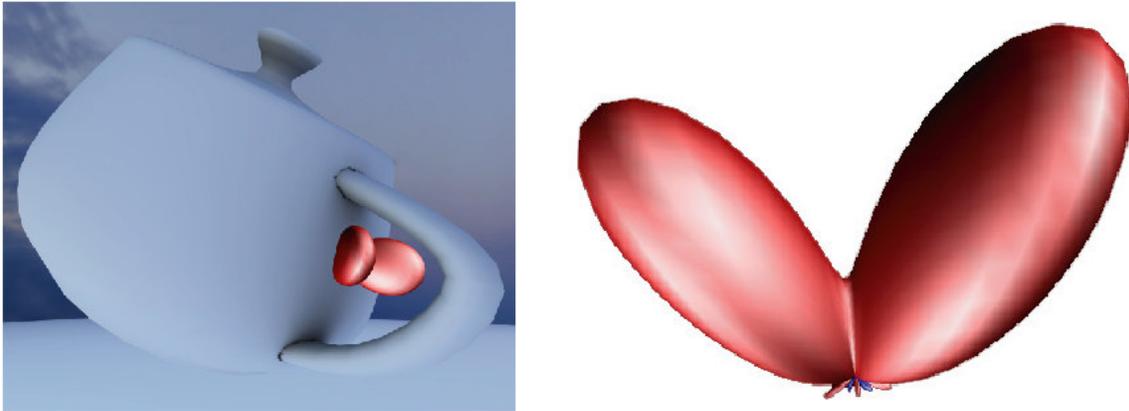


Figure 10. The spherical harmonic radiance transfer function created for a single point under the handle. The crease is due to occlusion from the handle. (*Courtesy of Chris Oat, ATI Technologies Inc.*)

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. Interreflection is computed 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, so modify its spherical harmonic values to now include how radiance is transferred by bouncing off occluding surfaces. This process can be repeated 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, the lighting of the object can be computed instantly. See Figure 11 for an example. The two spherical harmonic sets can be combined at each vertex by a simple series of dot products, yielding the color of the vertex. This basic property, that the two spherical harmonic representations can be combined 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.



Figure 11. Head rendered with standard diffuse shading and with PRT. (Courtesy of Peter-Pike Sloan, Microsoft Inc.)

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. These representations can be converted to irradiance maps, which are in turn 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 [Oat2005].

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 [McTaggart04]. What is exciting about all this work is that it is a different way of thinking about the problem, one that leverages the capabilities of the GPU.

### **Improved Bump Mapping**

To conclude this brief survey of new shading algorithms, we 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 making it into 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, there have been a number of new schemes developed for creating more realistic bumps on surfaces. The two we 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, off-line 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 so be 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 [Kaneko01], and a number of others have been developed since then.

Brawley and Tatarchuk's parallax occlusion mapping approach [Brawley04] is relatively straightforward to describe and gives a flavor of how fragment shaders and texture access can be used to improve image appearance. 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, the height field location that actually blocks the pixel can be determined. See Figure 12.

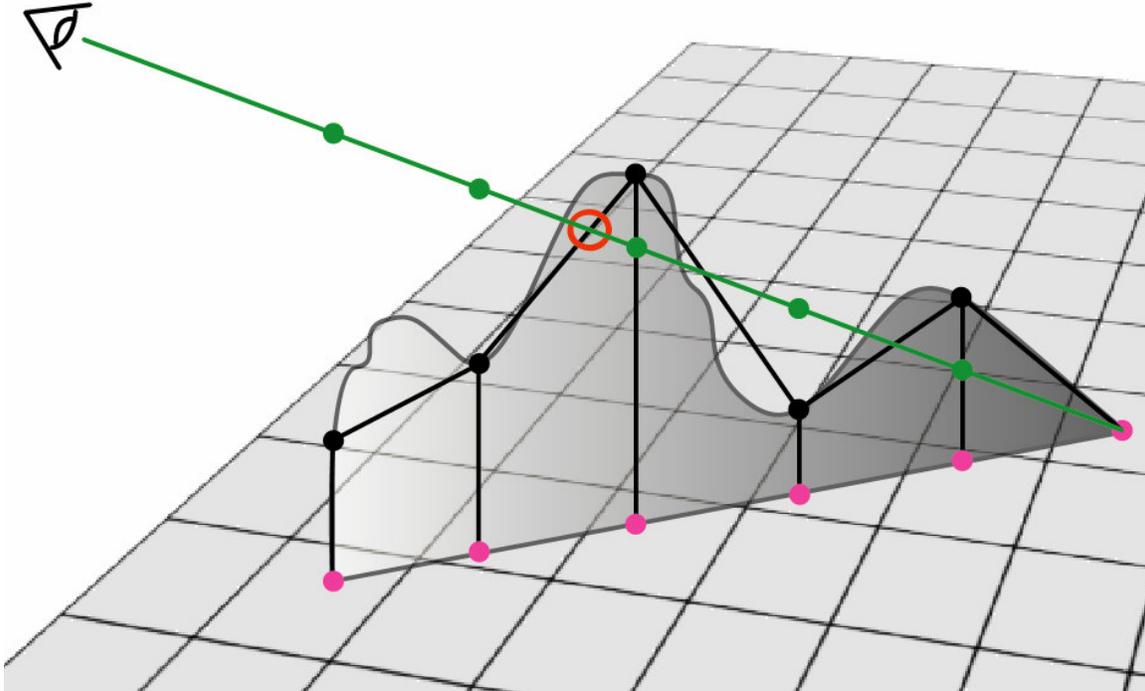


Figure 12. Parallax occlusion mapping. The green eye ray is projected onto the surface plane, which is sampled at regular intervals (the pink dots) and heights retrieved. The algorithm finds the first intersection of the eye ray with the black line segments approximating the height field.

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 is 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 be used to have the bumpy surface cast shadows onto itself. See Figure 13.

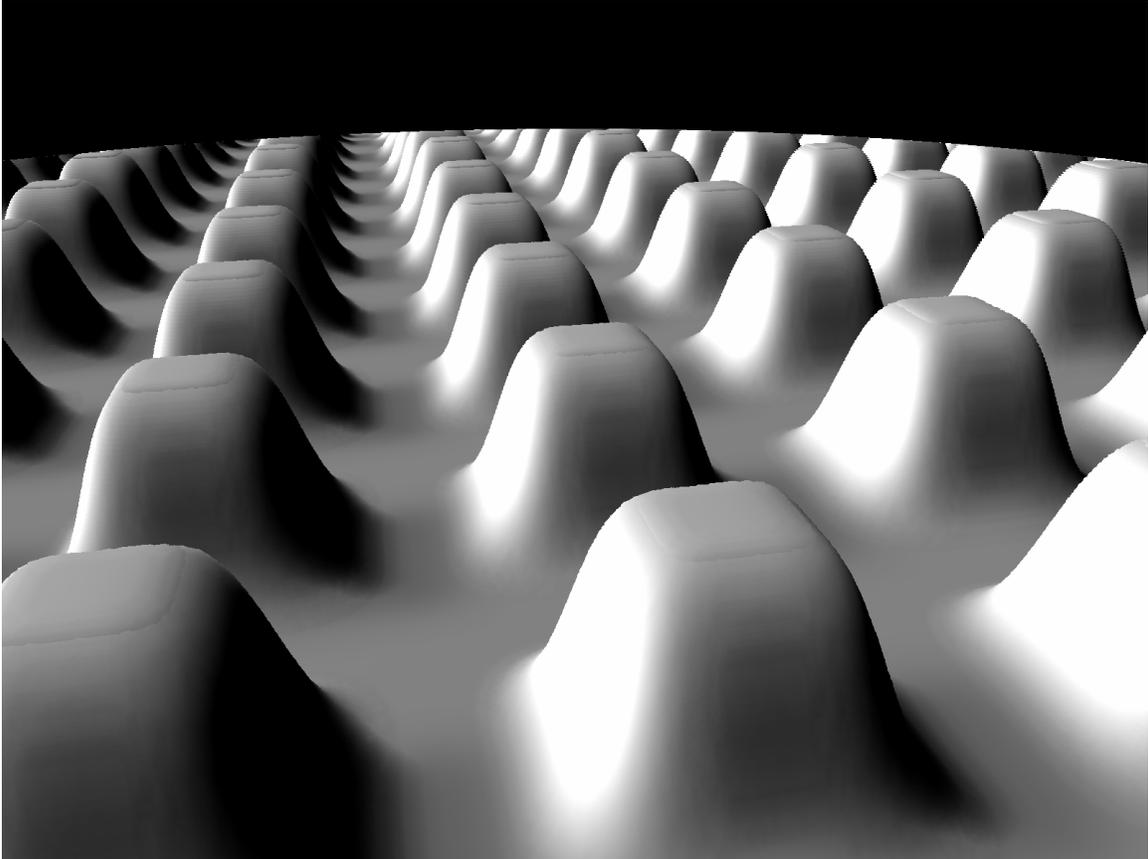


Figure 13. Height field rendering using parallax occlusion mapping (*Courtesy of Natalya Tatarchuk, ATI Technologies Inc.*)

There are some limitations to this technique. At shallow angles the number of height field locations that should be sampled may exceed the number of texture retrievals that can be done by the fragment shader. However, the algorithm can be extended to adaptively sample the surface in these cases [Tatarchuk05]. 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 [Oliveira05] 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.

One graphics hardware development in 2004 was that vertex processors gained the ability to access texture data. This functionality can be used for true displacement mapping: each vertex on a mesh can be shifted by a heightfield 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 which can then be accessed by vertex processors. Collisions between objects can

truly deform the objects simply by painting the colliding areas onto the texture [Wrotek]. Also, fragment processors can generate XYZ coordinate values, which the vertex processor then accesses to generate sprites for particle systems [Kipfer04]. 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 [Tresniak05]. 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 to be performed. 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 is called a *scatter operation*, and is important in building GPU-based data structures. For example, Purcell et al. [Purcell03] make use of the scatter operation to perform photon mapping on the GPU.

### **Further Up the Pipeline**

The major focus of this tour is about 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 is 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 backwards. 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 being able to be used for 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 1990's, 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, etc. Eventually there was enough memory available that current cards have plenty to 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 be used to 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 [Lastra99], 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 capability of a processor has been going up 71% per year, DRAM bandwidth is increasing about 25% and latency a mere 5% [Owens05]. What this means is that with the latest GPU processors you can do about 12 FLOPS 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, there are a number of techniques that have been developed to 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 only those triangles visible in a scene are processed by the pipeline. 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 HyperZ technology (ATI's name; NVIDIA has its own version), which avoids touching memory by treating sets of pixels as tiles. For example, call an 8x8 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, so 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 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 may sound like a trivial operation, but avoiding it as much as possible 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 had to be sent to the GPU. The fastest triangle to render is the one never sent down the pipeline at all. Towards this end, techniques have been

developed to deal with groups of triangles. On the CPU it is common to use frustum culling, in which each object is given a bounding box or sphere. If this bounding volume is found to be 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 it 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 a number of tests into a single query, and the faster transfer speed of PCI Express, make such techniques more feasible, especially for very large data sets [Wimmer05]. 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 [Myer68], can be seen when a new piece of peripheral hardware is developed and elaborated. The functionality of this system 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 *dataflow* or *stream* processor. It is the appearance of data at the beginning of the pipeline that causes computation to occur, and a limited set of inputs are needed to perform the computation. This different processing model lends itself in particular to problems where each datum is affected by only nearby data. One active area of research is how to apply the GPU to non-rendering problems of this type, such as fluid flow computation and collision detection (visit [ggpu.org](http://ggpu.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 be used to generate a curved surface. DirectX 10 includes an interface to perform tessellation. However, as yet there has been no generalized geometry tessellation 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 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 (including you!).

## 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* [Akenine-Möller02]. 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. NVIDIA's and ATI's 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*, *ShaderX*, and *Game Programming Gems* book series explain many new techniques and give code samples.

## Acknowledgements

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

[Akenine-Möller02] Akenine-Möller, Tomas, and Eric Haines, *Real-Time Rendering*, Second Edition, A.K. Peters, 2002. <http://www.realtimerendering.com>.

[Brawley04] Brawley, Zoe, and Natalya Tatarchuk, "Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing," in Wolfgang Engel, ed., *ShaderX<sup>3</sup>*, Charles River Media, pp. 135-154, 2004.

[Duca05] Duca, Nathaniel, Krzysztof Niski, Jonathan Bilodeau, Matthew Bolitho, Yuan Chen, and Jonathan Cohen, "A Relational Debugging Engine for the Graphics Pipeline," *ACM TOG 24 (3) (SIGGRAPH 2005)*, Aug. 2005, pp. 453-463.

[Hasenfratz03] Hasenfratz, Jean-Marc, Marc Lapierre, Nicolas Holzschuch, and François Sillion, "A Survey of Real-Time Soft Shadows Algorithms," *Computer Graphics Forum*, Volume 22, Number 4, December 2003, pp. 753-774.

[Hudon03] Hudon, Toby, "Timeline: 3DFX Revisited," <http://www.aceshardware.com/read.jsp?id=60000292>, 2003.

[Kaneko01] Kaneko, Tomomichi, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi, "Detailed Shape Representation with Parallax Mapping," *ICAT 2001*, Tokyo, pp. 205-208, Dec. 2001.

[Kipfer04] Kipfer, Peter, Mark Segal, and Rüdiger Westermann, "UberFlow: A GPU-based Particle Engine," *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 115-122, 2004.

[Lastra99] Lastra, Anselmo, "All the Triangles in the World," *Cornell Workshop on Rendering, Perception, and Measurement*, April 1999.

[Lengyel00] Lengyel, Jerome, "Real-Time Fur," *11th Eurographics Workshop on Rendering*, pp. 243-256, June 2000.

[Maughan01] Maughan, Chris, "Texture Masking for Faster Lens Flare," in Mark DeLoura, ed., *Game Programming Gems 2*, Charles River Media, pp. 474-480, 2001.

[McTaggart04], Gary, "Half-Life 2/Valve Source Shading", *Game Developers Conference 2004*, March 2004, [http://www2.ati.com/developer/gdc/D3DTutorial10\\_Half-Life2\\_Shading.pdf](http://www2.ati.com/developer/gdc/D3DTutorial10_Half-Life2_Shading.pdf).

[Myer68] Myer, T.H., and I.E. Sutherland, "On the Design of Display Processors," *Communications of the ACM*, vol. 11, no. 6, pp. 410-414, June 1968.

[Oat01] Oat, Chris, "Irradiance Volumes for Games", *Game Developers Conference 2001*, March 2001, [http://www.ati.com/developer/gdc/GDC2005\\_PracticalPRT.pdf](http://www.ati.com/developer/gdc/GDC2005_PracticalPRT.pdf).

[Oliveira05] Oliveira, Manuel M., and Fabio Policarpo, "An Efficient Representation for Surface Details," *UFRGS Technical Report RP-351*, January 26, 2005.

[Owens05] Owens, John, "Streaming Architectures and Technology Trends," *GPU Gems 2*, ed. Matt Pharr, Addison-Wesley, pp. 457-470, 2005.

[Purcell03] Purcell, Timothy J., Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan, "Photon Mapping on Programmable Graphics Hardware," *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 41-50, 2003.

[Ramamoorthi01] Ramamoorthi, Ravi, and Pat Hanrahan, "An Efficient Representation for Irradiance Environment Maps," *Computer Graphics (SIGGRAPH 2001)*, pp. 497-500, August 2001.

[Sloan02] Sloan, Peter-Pike, Jan Kautz, and John Snyder, "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments," *ACM TOG 21(3) (SIGGRAPH 2002)*, pp. 527-536, July, 2002.

[Tatarchuk05] Tatarchuk, Natalya, "Dynamic Image-Space Displacement Mapping with Silhouette Antialiasing via Parallax Occlusion Mapping," <http://www.ati.com/developer/techpapers.html>, 2005.

[Tresniak05] Tresniak, Orin, "Rendering Detailed Outdoor Ground Surfaces on the GPU," *SIGGRAPH 2005 Sketch*, 2005.

[Wimmer05] Wimmer, Michael and Bittner, Jiri, "Hard Occlusion Queries Made Useful," *GPU Gems 2*, ed. Matt Pharr, Addison-Wesley, pp. 91-108, 2005.

[Woop05] Woop, Sven, Jörg Schmittler, and Philipp Slusallek, "RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing," *ACM TOG 24(3) (SIGGRAPH 2005)*, pp. 434-444, August, 2005.

[Wrotek] Wrotek, Pawel, Alexander Rice, Morgan McGuire, "Real-Time Collision Deformations using Graphics Hardware", *journal of graphics tools*, to appear.

Bio: Eric Haines is a Lead Software Engineer at Autodesk, Inc. He is currently an editor of the *journal of graphics tools*, online editor for *ACM TOG*, and maintainer of the *Graphics Gems* code repository, among other activities. He received an MS from the Program of Computer Graphics at Cornell in 1985. He is a member of the IEEE.

