SMARTSHADER™ TECHNOLOGY WHITE PAPER

Introduction

Developers of 3D graphics applications have always had difficulty creating realistic computer generated characters, objects and environments that can be interacted with in real time. The limitation has been a lack of available processing power combined with the restricted flexibility afforded by existing graphics hardware. There has always been a trade-off between performing operations on the CPU, which allows more flexibility due to its general and programmable nature, and the graphics processor, which allows better performance due to its hard-wired and heavily optimized architecture. While the rapidly increasing speed of graphics processors has enabled a significant amount of progress, and while they have been steadily taking over many of the tasks formerly handled by the CPU, there are many interesting and useful graphical techniques that have remained out of reach because they require a combination of speed and flexibility that neither existing CPUs or graphics processors could adequately provide.

What is needed is a technology that combines the speed and optimizations of a dedicated graphics processor with the flexibility and programmability of a CPU, allowing a virtually infinite range of visual effects at interactive frame rates. The first attempts at introducing this kind of technology were successful in increasing the number of effects available to developers, but still suffered from a number of limitations in terms of both versatility and performance. SMARTSHADER™ technology, developed by ATI, removes these limitations to free developers' imaginations.

SMARTSHADER™ Technology

SMARTSHADER™ technology brings a new level of graphical effects to personal computers. It allows software developers to use techniques that, until recently, were only available to the creators of non-interactive computer generated movies and special effects, and bring them to interactive computer games, the world wide web, and digital content creation applications.

ATI's SMARTSHADER™ technology represents a new generation of programmable, hardware-accelerated graphics pipelines. The technology was developed with a keen eye toward maximizing efficiency and minimizing common performance bottlenecks, especially memory bandwidth. SMARTSHADER™ technology is an extension of the Vertex Shader and Pixel Shader programming languages first introduced by Microsoft in DirectX® 8.0. While these shader languages were a good first attempt at bringing programmability to graphics hardware, experimentation revealed that they had a number of limitations that offered many opportunities for improvement.

Incorporating suggestions from leading 3D application developers and researchers, ATI worked closely with Microsoft to improve the flexibility, efficiency, and usability of the DirectX® 8.0 shader languages. The results are being made available to developers for the first time with the release of DirectX® 8.1, which reveals the full capabilities of SMARTSHADER™ technology. For developers that prefer to work with OpenGL®, the full SMARTSHADER™ feature set is also accessible using custom extensions provided by ATI.

The key improvements offered by ATI's SMARTSHADER™ technology over existing hardware vertex and pixel shader implementations are:

- Support for up to six textures in a single rendering pass, allowing more complex effects to be achieved without the heavy memory bandwidth requirements and severe performance impact of multi-pass rendering
- A simplified yet more powerful instruction set that lets developers design a much wider ranger of graphical effects with fewer operations
- Pixel shaders up to 22 instructions in length (compared to 12 instructions in DirectX® 8.0 Pixel Shaders) allow more accurate simulation of the visual properties of materials
- Ability to perform mathematical operations on texture addresses as well as color values, enabling new types and combinations of lighting and texturing effects that were previously impossible

Programmable Shaders and Fixed Function Effects

Graphics APIs like DirectX® and OpenGL® have historically consisted of a set of fixed functions. By calling a function and passing it a set of parameters, a developer could cause things to be drawn on the screen in a certain way. Graphics processors could accelerate these functions using hard-wired circuitry, which can perform the calculations for fixed functions very quickly. As graphics hardware became more complex and powerful, new functions were added to the APIs to give developers access to new features and visual effects.

This process worked reasonably well, but it caused a major gap to develop between the time new features were conceived and the time an end-user could see the result. For example, consider an imaginary new 3D graphics technique called Feature X. Before an end-user can experience the benefits of Feature X, someone first has to design the algorithm, and perhaps publish a research paper on the subject. If Feature X looks promising, then a graphics chip designer has to think of an efficient way to implement support for it in hardware. Once the hardware design is near completion, someone has to implement support for Feature X in the major APIs to make it accessible to application developers. Finally, the developers have to determine how to use Feature X in their application, test it, optimize it, and ship a product with the feature enabled. This process can take several years, and slows the pace of innovation (see Figure 1).

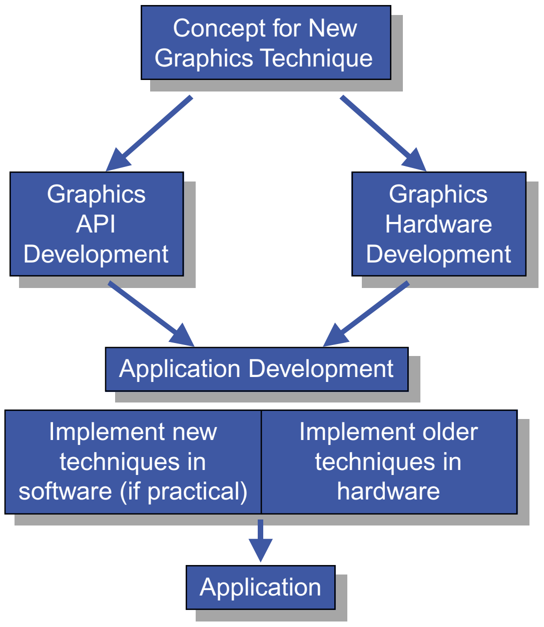
Concept for New Graphics Technique
Graphics API Development
Graphics Hardware Development
Application Development
Implement new techniques in software (if practical)
Implement older techniques in hardware
Application

***Figure 1:** The development and implementation cycle for new graphics techniques, before the advent of programmable shaders*

Programmable shaders represent a new way of doing things that can substantially decrease the time it takes to bring innovative new graphics ideas to the end-user. Shaders are small programs that application developers can write and use to determine how things are drawn on the screen. This means that immediately after the idea for Feature X is thought of, a developer can write a shader to add that feature into their application without having to wait for new graphics hardware or new versions of their preferred graphics API. Bypassing these steps can cut years off of the time it takes to develop new features and deliver them to end-users (see Figure 2).

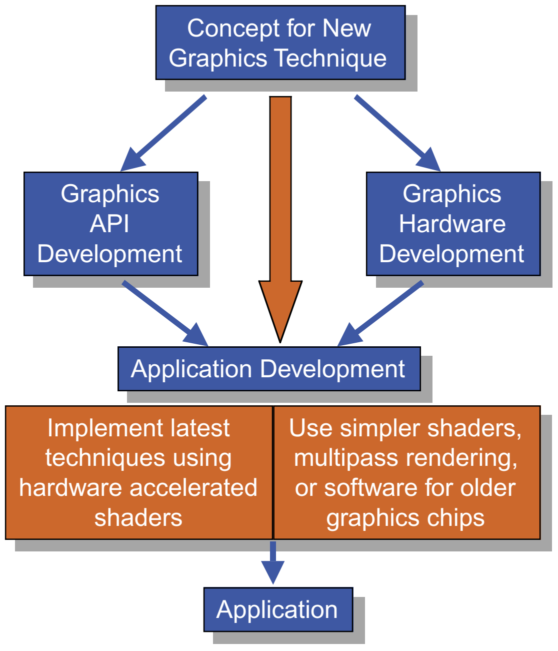
Concept for New Graphics Technique
Graphics API Development
Graphics Hardware Development
Application Development
Implement latest techniques using hardware accelerated shaders
Use simpler shaders, multipass rendering, or software for older graphics chips
Application

Figure 2: *Programmable shaders reduce the time required to bring new graphics techniques to applications*

It is also possible to write shaders that duplicate or replace existing fixed functions, such as Environment Mapped Bump Mapping or Matrix Palette Skinning. Although the shader versions of these functions can sometimes run slightly slower than the fixed versions, the ability to tweak and optimize the shaders to fit each particular application can provide many other benefits. For example, fixed function matrix palette skinning is limited to four matrices per vertex in DirectX® 8.0, whereas matrix palette skinning using vertex shaders can handle dozens of matrices per vertex if necessary. The result is that skeletally animated characters with more than four bones can be animated as a whole rather than one piece at a time, letting animators work more quickly and easily.

The 3D Graphics Pipeline

Drawing 3D graphics involves numerous steps that are performed one after the other. These steps can be thought of like an assembly line; or a pipeline. The 3D graphics pipeline can be broken down into two main stages: the geometry processing stage, and the rendering stage.

All 3D graphics start as a set of models created by an artist, which are represented using a series of interconnected points known as vertices. These vertices are stored as chunks of data that contain several pieces of information about themselves, such as their position, color, texture, reflectivity, etc. Once a software application determines what objects will be present in the 3D scene being drawn and how they are arranged, the appropriate vertex data is sent to the geometry processing stage, as shown in Figure 3.

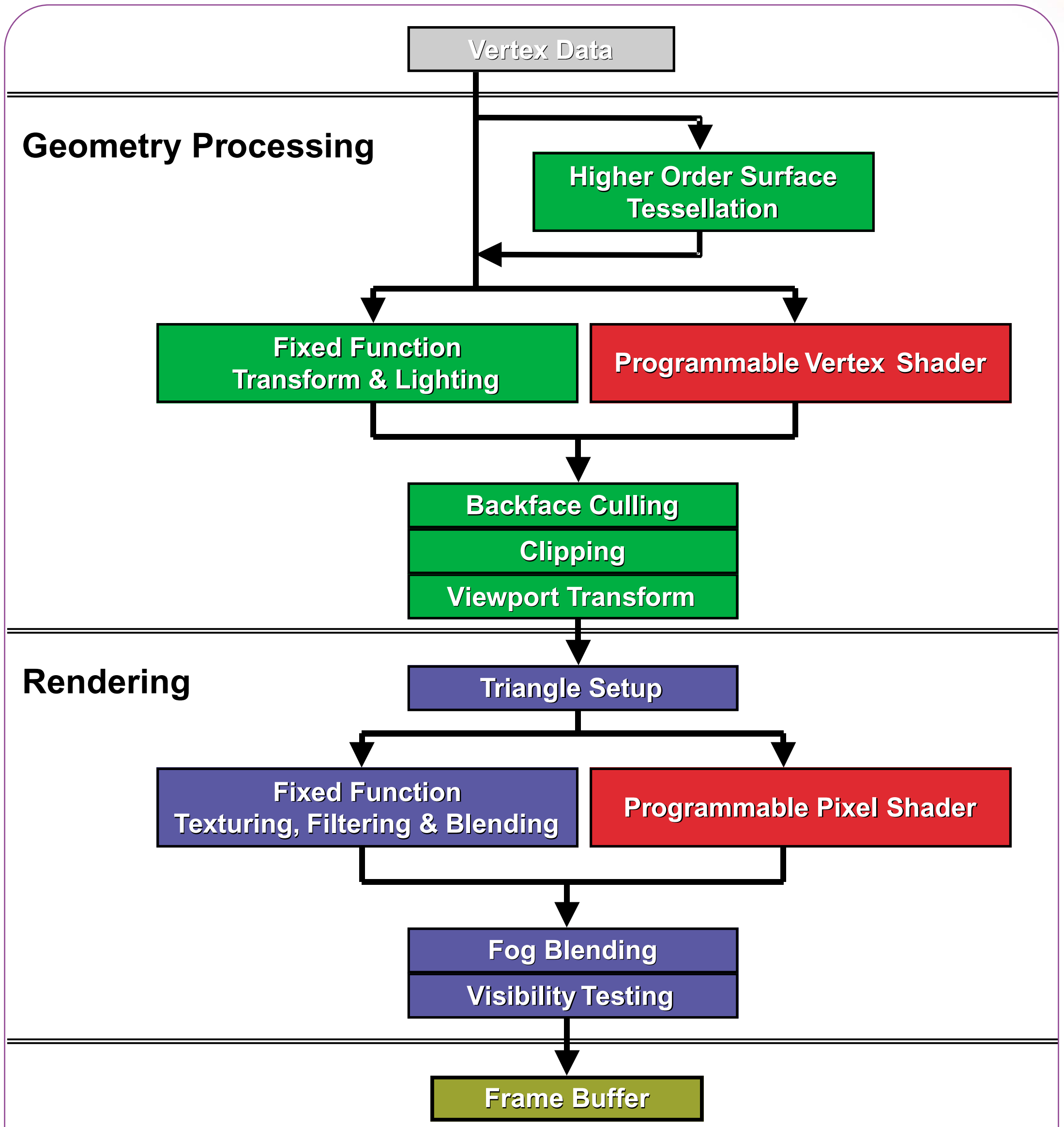
Vertex Data
Geometry Processing
Higher Order Surface Tessellation
Fixed Function Transform & Lighting
Programmable Vertex Shader
Backface Culling
Clipping
Viewport Transform
Rendering
Triangle Setup
Fixed Function Texturing, Filtering & Blending
Programmable Pixel Shader
Fog Blending
Visibility Testing
Frame Buffer

Figure 3: *The 3D Graphics Pipeline*

When using a graphics chip that supports hardware accelerated transform and lighting (T&L), the geometry processing stage takes place on the GPU; otherwise, it is handled by the CPU. Geometry processing involves calculations that modify or, in some cases, create new data for vertices. At this point, either fixed function transform and lighting operations or programmable vertex shaders can be used to perform a specified set of operations on every incoming vertex. These operations can be executed on the CPU if they are not supported by the graphics hardware, but this almost always results in greatly reduced performance.

Once the vertices have passed through the geometry processing stage, they must then be passed on to the rendering stage for conversion into pixels. The image that appears on a display is composed of a two-dimensional array of pixels, each of which is assigned a color. The job of the rendering stage is to convert three-dimensional vertex data into two-dimensional pixel data. This is accomplished by first linking the vertices together to form triangles, then filling in each triangle with pixels of the appropriate color. The color of each pixel has to be chosen carefully to give the 2D image the appearance of depth and texture. This can be accomplished by using either fixed function texturing, filtering and blending instructions, or by using programmable pixel shaders. The rendering stage also blends in fog if desired, and performs visibility testing to determine if each pixel is hidden, partially visible, or fully visible.

After the rendering stage is completed the final colors of each pixel in the rendered image are written to a block of memory called the frame buffer. From this point, the image can either be sent back through the pipeline again for further processing, or sent to a display device for viewing.

Now that we have described how vertex and pixel shaders fit into the 3D graphics pipeline, we can look at how each of them works in more detail.

Vertex Shaders

Vertex shaders are small programs or sets of instructions that are performed on vertex data as it passes through the geometry processing pipeline. With ATI's SMARTSHADER™ technology, each vertex can consist of up to 16 distinct pieces of data, which are read by the vertex shader as individual streams. These pieces of data can contain positional co-ordinates, texture co-ordinates, lighting values, weighting values for matrix skinning, or anything else the developer desires. A vertex shader program can have a maximum length of 128 instructions, and make use of up to 96 constant values and 12 temporary data registers. These specifications provide ample ability to perform a huge range of transformations on incoming vertex data. The actual instructions are very similar to those found in CPU assembly language and allow for easy manipulation of vertex data. Figure 4 gives a detailed description of the SMARTSHADER™ vertex shader architecture.

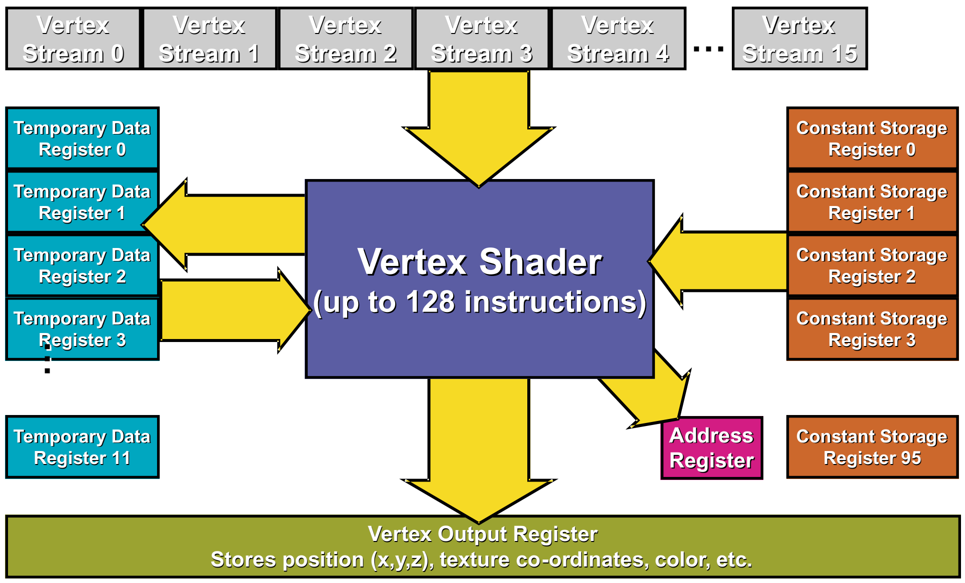
Vertex Stream 0
Vertex Stream 1
Vertex Stream 2
Vertex Stream 3
Vertex Stream 4
...
Vertex Stream 15
Temporary Data Register 0
Temporary Data Register 1
Temporary Data Register 2
Temporary Data Register 3
Temporary Data Register 11
Vertex Shader (up to 128 instructions)
Constant Storage Register 0
Constant Storage Register 1
Constant Storage Register 2
Constant Storage Register 3
Constant Storage Register 95
Address Register
Vertex Output Register
Stores position (x,y,z), texture co-ordinates, color, etc.

Figure 4: *SMARTSHADER™ Vertex Shader Architecture*

Vertex Shader Effects

Through the implementation of vertex shaders a huge variety of new graphics effects are now possible. Most of these effects involve modifications to the shape or position of objects, although they can also be used to modify the appearance of surfaces in conjunction with pixel shaders. Vertex shaders can also be optimized in many cases to provide superior performance to fixed function T&L operations. A few examples of vertex shader effects are described below.

Procedural Deformation

Many objects in the real world change shape according to simple mathematical functions. These functions can be modeled using vertex shaders and applied to the position of each vertex in an object to create highly realistic, procedural animation. This has a wide range of applications including water waves, flags and capes that blow in the wind, and wobbling objects like soap bubbles (see Figures 5 & 6).

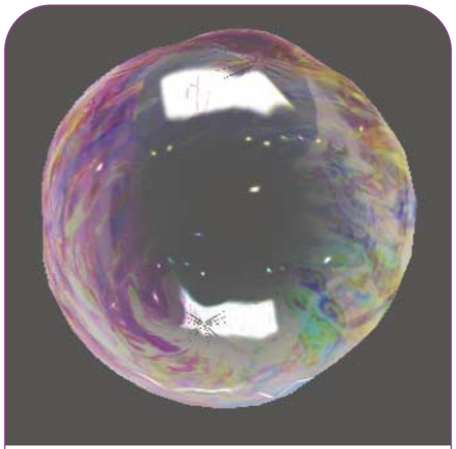

Figure 5: *Soap bubble animated using vertex shaders for procedural deformation*

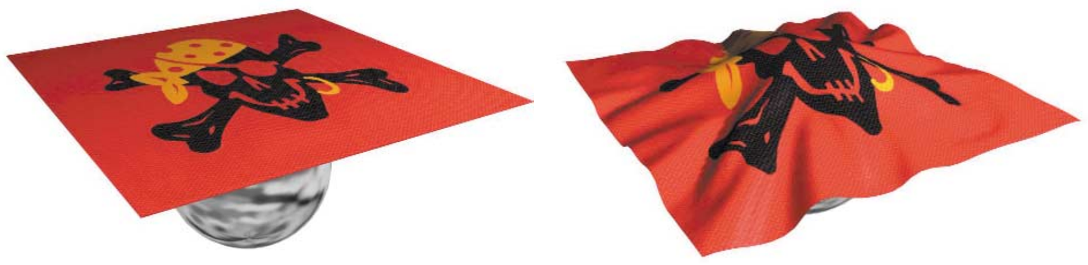

Figure 6: *Cloth simulation using vertex shaders*

Shadow Volumes

Shadows are a very important part of any scene, since they help convey a sense of depth and atmosphere. Vertex shaders provide a simple way of generating convincing shadows that can be fully animated and extended to multiple light sources. The shader is used to create transparent volumes that extend behind objects away from any light sources, creating shadows where the volumes contact other surfaces. The closer the light source is to the shadow-casting object, the darker the shadow is. See Figure 7 below.

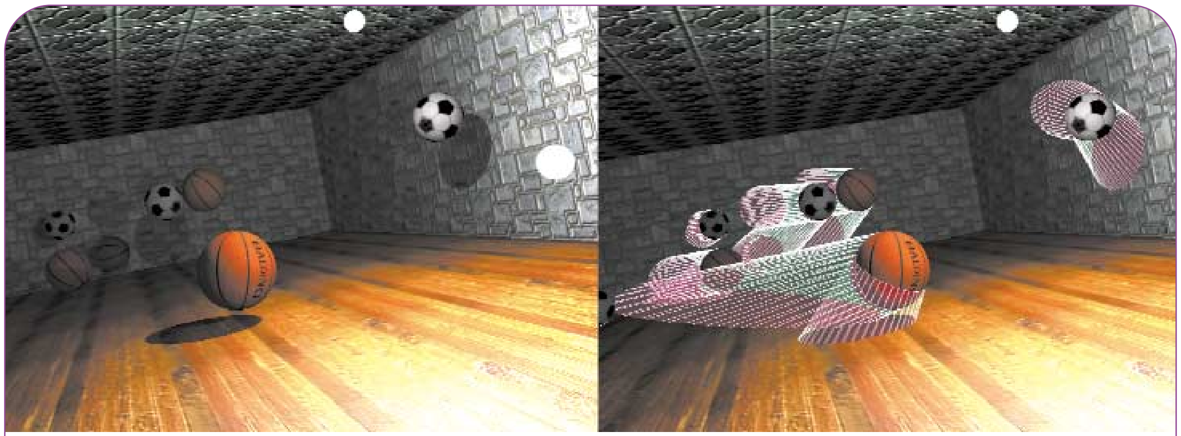

Figure 7: *Shadow generation using vertex shaders. The image on the right shows how shadow volumes are extended behind each object away from the light sources.*

Other vertex shader effects include, but are not limited to:

- Fur Rendering – grow realistic fur from any object that can be made to curl or blow in the wind if desired.
- Particle Systems – animate large numbers of small particles by assigning and modifying physical properties such as mass, velocity, acceleration, etc. Useful for modeling fire, sparks, explosions, smoke, rain, snow, and more.
- Lens Effects – distort an image as if it were being viewed through a lens.
- Matrix Palette Skinning – allow skeletally animated characters with many bones to move freely and bend naturally.
- Advanced Keyframe Interpolation – perform animation and morphing effects using non-linear blending between two or more keyframes. Useful for modeling complex facial expressions and speech.

Pixel Shaders

Pixel shaders are small programs that are executed on individual pixels as they pass through the rendering pipeline. With SMARTSHADER™ technology, up to six different textures can be sampled and manipulated in a single rendering pass to determine the color of a pixel. Textures can be one-dimensional, two-dimensional, or three-dimensional arrays of values stored in memory. Each value in a texture is called a texel, and although they are most commonly used to store color values, they can contain any kind of data desired including normal vectors (for bump maps), shadow values, or look-up tables. A pixel shader program can have a maximum length of 22 instructions, and make use of up to eight constant values and six temporary data registers. These specifications provide ample ability to perform a huge range of transformations on incoming vertex data. The actual instructions are very similar to those found in CPU assembly language and allow for easy manipulation of pixel data. Figure 8 gives a detailed description of the pixel shader architecture.

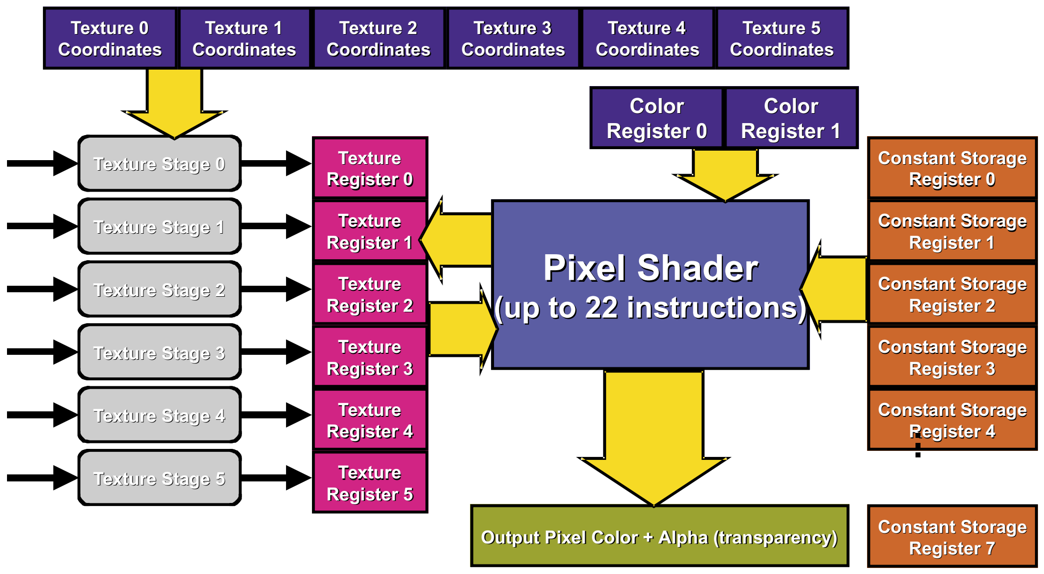

Figure 8: *SMARTSHADER™ Pixel Shader Architecture*

Pixel shader programs can be divided into two parts, as shown in Figure 9. The first part is called the address shader, which performs up to eight mathematical operations (addition, multiplication, dot product, etc.) on texture co-ordinates or addresses. Up to six textures can be sampled either before or after the address shader is executed. The ability to sample a texture value, modify that value in the address shader, and use the modified value as an address to sample a different texture, allow pixel shaders to perform what are known as dependent texture reads. This technique is necessary to accomplish environment mapped bump mapping, anisotropic lighting, and many other important effects. The second part of a pixel shader is known as the color shader, which consists of up to eight instructions that blend and modify the values (usually colors) of previously sampled textures to give the final pixel color. Whereas DirectX® 8.0 used different instruction sets for the color shader and the address shader, DirectX® 8.1 simplifies things by using the same instruction set for both parts of the pixel shader.

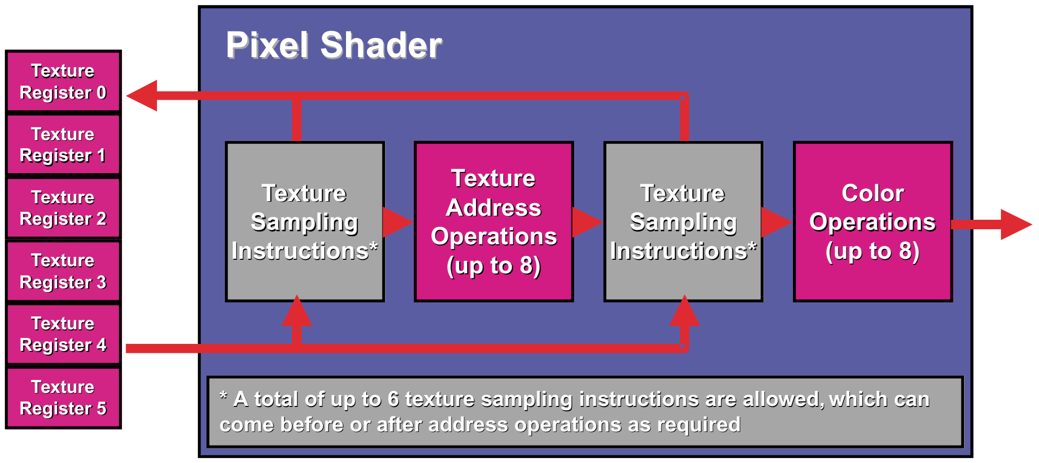
Pixel Shader
Texture Register 0
Texture Register 1
Texture Register 2
Texture Register 3
Texture Register 4
Texture Register 5
Texture Sampling Instructions*
Texture Address Operations (up to 8)
Texture Sampling Instructions*
Color Operations (up to 8)
* A total of up to 6 texture sampling instructions are allowed, which can come before or after address operations as required

Figure 9: *SMARTSHADER™ Pixel Shader Program Flow*

SMARTSHADER™ Advantages

Programmable shaders are intended to give 3D application developers the ability to devise and implement a practically infinite range of visual effects. However, as mentioned earlier, existing programmable shader implementations place significant limits on what developers can actually do. These constraints can be narrowed down into the following categories:

- Number of Input Variables
- Maximum Program Length
- Instruction Set
- Performance

The improvements offered by SMARTSHADER™ technology provide more variables to work with, allow longer shader programs, employ a more versatile instruction set, and open up more opportunities to save memory bandwidth and increase performance. These improvements were designed to release the constraints on 3D application developers by allowing them to use more sophisticated rendering techniques.

The table below compares some of the key specifications of the existing DirectX® 8.0 programmable shader standards:

	DirectX® 8.0 Vertex Shaders	**DirectX® 8.0 Pixel Shaders**
Max. Number of Inputs	16 vertex streams	4 textures
Max. Program Length	128 instructions	12 instructions
Constant Registers	96	8

This comparison highlights the imbalance between DirectX® 8.0 vertex and pixel shaders. Although vertex shaders are already quite flexible, pixel shaders are severely restricted in the number of effects they can be programmed to implement. The main reason for this is performance. A typical 3D scene might contain dozens or even hundreds of times more pixels than vertices, and so on existing graphics processors pixel shader programs must be less complex in order to maintain interactive frame rates. SMARTSHADER™ technology is designed for the next generation of high performance graphics processors, where executing complex pixel shaders will become more feasible. Thus, most of the SMARTSHADER™ improvements involve pixel shaders, with only minor instruction set enhancements being added to vertex shaders. These are summarized in the following table:

	DirectX® 8.0 Pixel Shaders	**SMARTSHADER Pixel Shaders (DirectX® 8.1)**
Max. Texture Inputs	4	6
Max. Program Length	12 instructions (up to 4 texture sampling, 8 color blending)	22 instructions (up to 6 texture sampling, 8 texture addressing, 8 color blending)
Instruction Set	13 address operations 8 color operations	12 address / color operations
Texture Addressing Modes	40	Virtually unlimited

In order to run at smooth frame rates, pixel shaders must be executed tens of millions of times per second, making performance a key concern. Every time a pixel passes through the rendering pipeline, it consumes precious memory bandwidth as data is read from and written to texture memory, the depth buffer, and the frame buffer. By decreasing the number of times each pixel on the screen has to pass through the rendering pipeline, memory bandwidth consumption can be reduced and the performance impact of using pixel shaders can be minimized. DirectX® 8.1 pixel shaders allow up to six textures to be sampled and blended in a single rendering pass. This means effects that required multiple rendering passes in earlier versions of DirectX® can now be processed in fewer passes, and effects that were previously too slow to be useful can become more practical to implement. Numerous textures are needed to create truly realistic scenes that contain bumpy surfaces combined with diffuse, specular and anisotropic lighting effects from multiple sources, as well as realistic reflections and shadows.

DirectX® 8.0 pixel shaders can use a maximum of four texture addressing (address shader) instructions and eight texture blending (color shader) instructions. While this allows much more complex effects than can be provided by a fixed function rendering pipeline, there are still many effects that are impossible to achieve with this limited number instructions. The most powerful capabilities of programmable pixel shaders are provided by the texture sampling and addressing instructions. With only four such instructions, DirectX® 8.0 pixel shaders are restricted to just 40 meaningful ways in which textures can be addressed. DirectX® 8.1 pixel shaders separate the sampling and addressing instructions, giving a total of up to 14 instructions to work with in the address shader. This change allows textures to be addressed and sampled in a virtually unlimited number of ways, and enables a wide range effects not possible with DirectX® 8.0.

Another area in which DirectX® 8.0 pixel shaders had room for improvement was their instruction set. Two different sets of operations were used for address shaders and color shaders. Whereas the color shader instructions were fairly simple mathematical operations, the address shader instructions were relatively complex and inflexible. DirectX® 8.1 simplifies the pixel shader language by allowing the color shader instruction set to be used for address shaders as well. In addition to making pixel shaders easier to write, this new approach removes many of the data type incompatibilities that arose from using two different languages in a single shader, meaning that many effects can now be accomplished with fewer instructions than before.

Pixel Shader Effects

The following examples provide just a glimpse at the many types of pixel effects that are now possible only with SMARTSHADER™ technology. While most vertex shader effects involve altering the shape and position of objects, most pixel shader effects are focused on controlling the material properties and texture of surfaces. Many of these effects are accomplished using texture addressing instructions to perform mathematical operations on the texture co-ordinates passed into the pixel shader, then using these modified co-ordinates to perform dependent texture reads.

Single Pass, Per-Pixel Rendering of Multiple Light Sources

Some of the most important effects made possible by SMARTSHADER™ technology involves per-pixel lighting from multiple light sources. Making use of all six texture inputs to the pixel shaders, bumpy objects can be made to accurately reflect light from multiple sources in a single rendering pass. Light sources can include environment maps (providing realistic reflections), point lights, spotlights, glowing objects, and more. See Figure 10.

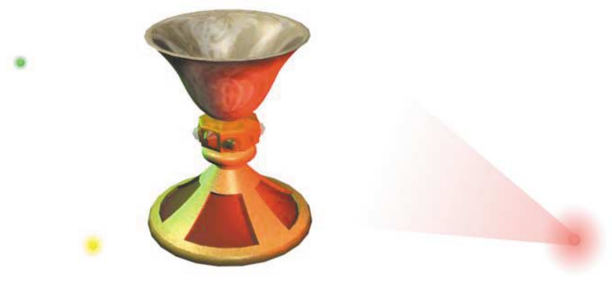

Figure 10: *An object lit by three spotlights of different colors. SMARTSHADER technology can be used to render images like this with a fraction of the number of rendering passes that were required before.*

True Phong Shading

Phong shading is a mathematical model that forms the basis of most lighting calculations in 3D applications today. It divides incoming light into three components (ambient, diffuse, and specular), and given the properties of the surface it is shining on, determines the final color the viewer will see at any given point. The equation used is as follows:

$$\text{Light} = \mathrm{D}_m * \sum_{1}^{\text{lights}} \mathrm{A}_i + \sum_{1}^{\text{lights}} (\mathrm{DC}_i * \mathrm{Dist}_i * \mathrm{N}\cdot\mathrm{L}_i)$$

$$+ \ \mathrm{S}_m * \sum_{1}^{\text{lights}} (\mathrm{DS}_i * \mathrm{Dist}_i * (\mathrm{N}\cdot\mathrm{H})^{k}{}_{i}) + \mathrm{E}_m$$

where

D_m is the Diffuse material property (RGB color)
S_m is the Specular material property (usually scalar “gloss”)
E_m is the Emissive material property
A_i is the Ambient contribution from light i
DC_i is the Diffuse Color of light i
SC_i is the Specular Color of light i
Dist_i is the Distance Attenuation term for light i

Because this equation is so complex and has so many variables, most graphics applications can only use a simplified approximation. SMARTSHADER™ technology allows the full Phong shading equation to be implemented, even for multiple light sources. Any variable in the equation, such as *k* (which controls “shininess”), can be varied across any surface using specialized texture maps. The result is that any kind of material can be made to look extremely realistic (see Figure 11).

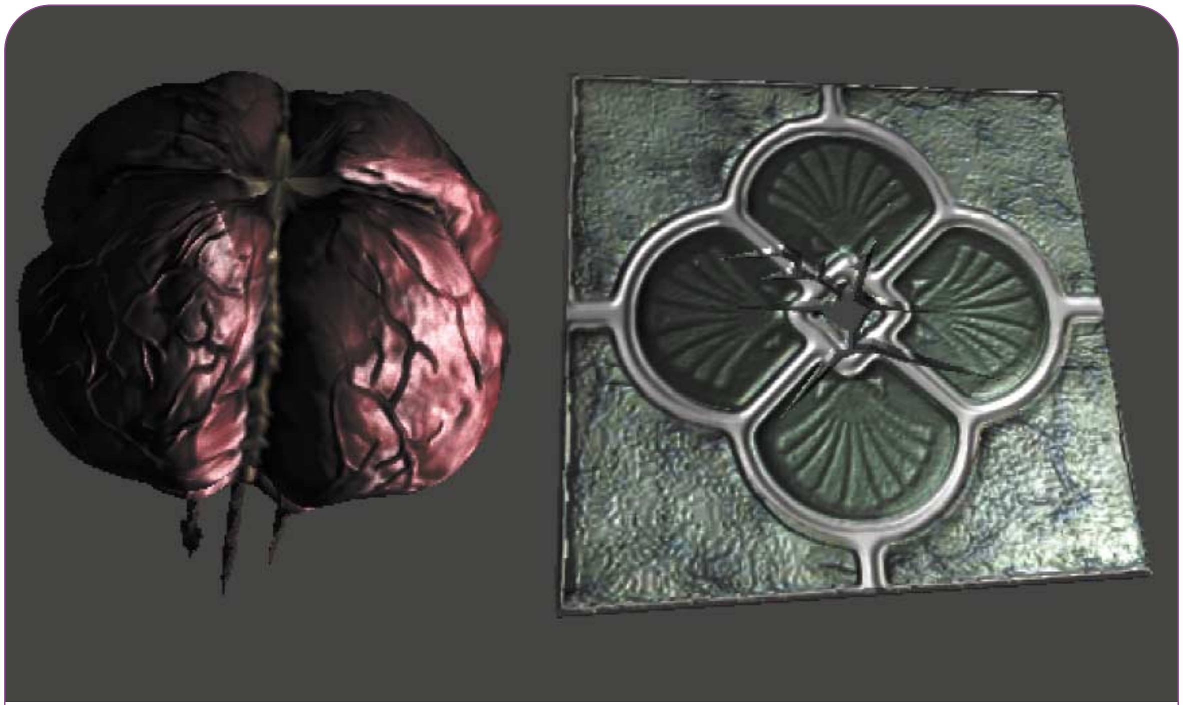

Figure 11: *Examples of per-pixel Phong shading used to render both real and imaginary materials.*

Anisotropic Lighting

While Phong shading works well for materials that are basically smooth, many surfaces in the real world like cloth, hair, and sand are made up of many small strands or particles. This characteristic causes them to reflect light in unusual ways, and it can only be modeled properly on a computer screen using a technique known as anisotropic lighting. Anisotropic lighting uses complex functions known as BRDFs (Bidirectional Reflectance Distribution Functions) to determine how light bounces off a surface as a function of the viewing angle. The results of these functions are stored in special lookup table textures that are unique to each particular type of surface. An example of this is shown in Figure 12.

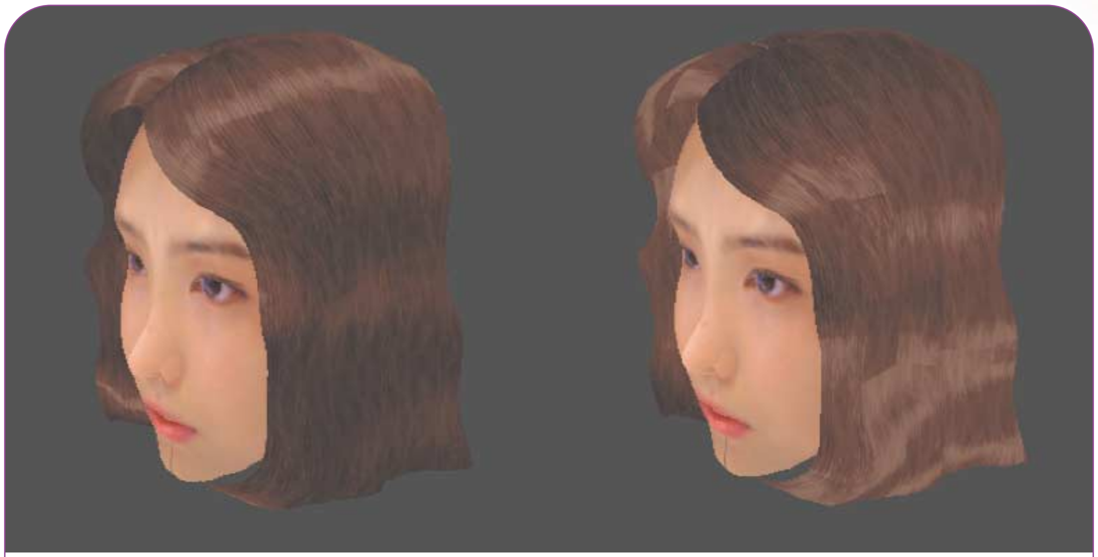

Figure 12: *Hair modeled using anisotropic lighting. The two images show how the reflected light changes as the light source is moved.*

Advanced Bump Mapping

Bump mapping has been demonstrated as a useful technique for improving the realism of surfaces without requiring a huge amount of additional geometry data. Both Environment Mapped Bump Mapping (which works best on shiny surfaces) and Dot Product 3 Bump Mapping (which works best on matte surfaces) can be implemented using pixel shader. ATI's SMARTSHADER™ technology makes it possible to go a step further with more advanced bump mapping effects, such as self-shadowing bump maps (also known as Horizon Mapping) and the ability to combine multiple bump maps (see Figure 13 below).

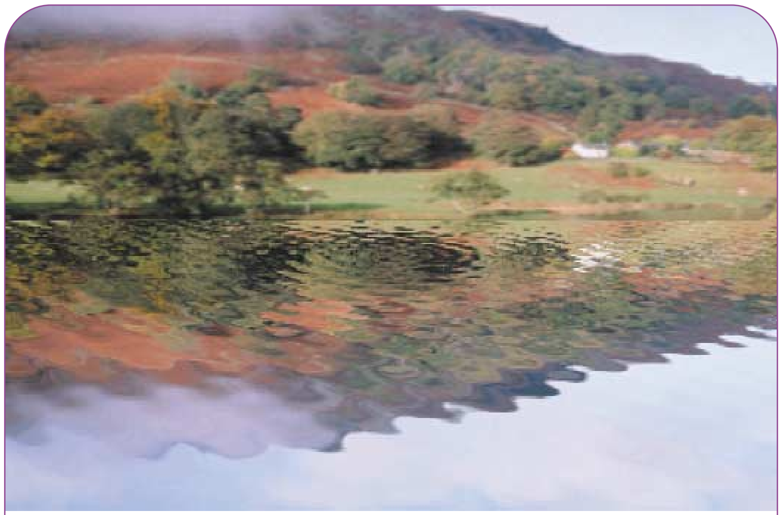

Figure 13: *Rippling water rendered using superimposed bump maps (the ripples spreading from the center have been added to the ripples moving across the lake). Just four triangles were needed to render this image.*

Procedural Textures

Detailed textures normally require large amounts of graphics memory for storage, which limits the variety of textures and amount of detail that can be used. However, with SMARTSHADER™ technology it has become possible to create a wide variety of interesting textures using mathematical functions, which require little or no memory storage because they are generated on the graphics chip. One example of this technique is shown in Figure 14.

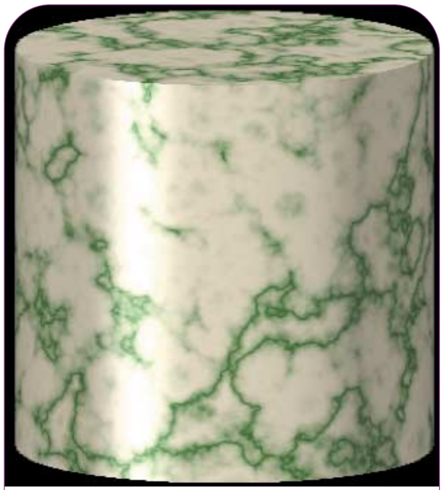
Figure 14: *A marble pattern generated using a procedural texture.*

Conclusion

ATI's new SMARTSHADER™ technology allows developers of 3D games and other applications to unleash their creativity with a fully programmable graphics pipeline for both geometry processing and rendering. Vertex shaders enable custom transform and lighting effects, giving complete control over the shape and position of 3D characters, objects, and environments. Pixel shaders enable a huge number of different per-pixel effects, allowing accurate simulations of the natural properties of materials such as hair, cloth, metal, glass, and water that were previously difficult to achieve in real-time applications. Important visual cues such as reflections, highlights, and shadows from multiple light sources can also be rendered with unprecedented speed and accuracy. Both pixel and vertex shaders are accessible using the DirectX® 8.1 API from Microsoft as well as OpenGL® extensions from ATI.

SMARTSHADER™ technology takes a major step beyond existing hardware shader implementations with support for up to six textures per rendering pass, nearly double the number of instructions, and a new and improved shading language that provides greater flexibility and ease of use. These advancements greatly increase the number of graphical effects that can be created, improve performance by conserving memory bandwidth, and allow more games and other applications to take advantage of the technology. With SMARTSHADER™ technology, the pace of graphical innovation will accelerate, and the gap in visual quality between what has been seen in movies and what appears on the computer screen will become narrower than ever before.