Technical Brief

NVIDIA nfiniteFX™ Engine
Programmable Vertex Shaders

The NVIDIA® nfiniteFX? Engine:

The NVIDIA nfiniteFX engine gives developers the ability to program a virtually infinite number of special effects and custom looks. Instead of every developer choosing from the same hard-coded palette of effects and ending up with the same generic look and feel, developers can specify personalized combinations of graphics operations and create their own custom effects. Games and other graphics-intensive applications are differentiated and offer more exciting and stylized visual effects. Two patented architectural advancements enable the nfiniteFX engine's programmability and its multitude of effects: Vertex Shaders and Pixel Shaders. This paper covers the engine's programmable Vertex Shaders.

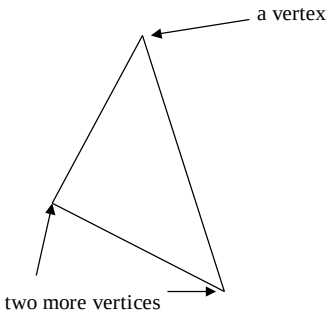
Programmable Vertex Shaders:

The Next Generation of Geometry Processing

The advancement of graphics processing technology continues to provide users with increasingly realistic and detailed real-time 3D graphics. With the introduction of NVIDIA's groundbreaking nfiniteFX engine, programmable Vertex and Pixel shaders were unleashed on the graphics community. Programmable Vertex Shaders are a prime example of the new functionality in graphics processor units (GPUs) that enables a virtually unlimited palette of real-time visual effects. Requiring complex computations, these effects were previously possible only with "offline" rendering using server farms. The addition of programmable Vertex Shaders to consumer graphics processors ushers in stunning PC graphics.

What is a Vertex Shader?

A *Vertex Shader*—a graphics processing function—adds special effects to objects in a 3D graphics scene. A *programmable Vertex Shader* lets developers adjust effects by loading new software instructions into the Vertex Shader memory. Vertex Shaders perform mathematical operations on the vertex data for objects. Each vertex is defined by a variety of data variables. At a minimum, each vertex has associated with it x, y, and z coordinates that define its location. A vertex may also include data for color, alpha-channel, texture, and lighting characteristics such as specular color. See Figure 1 for a visual example of a vertex and some of its associated data.



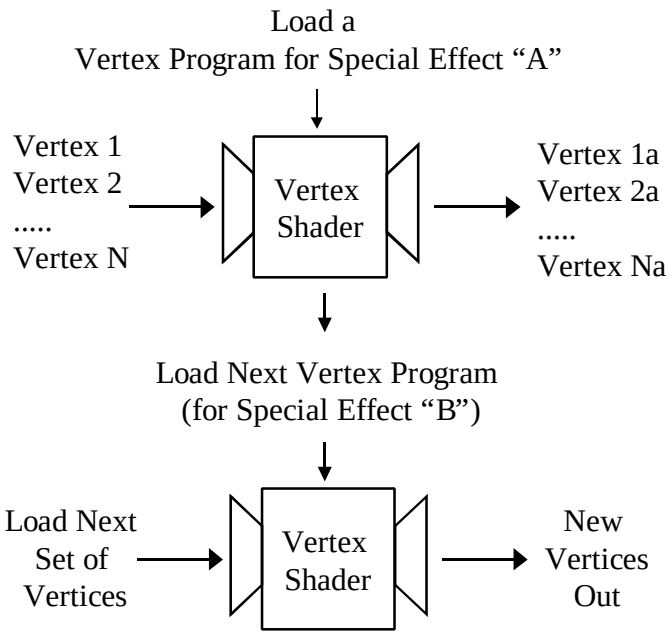
Example: Vertex Data

position: {X, Y, Z, W}
color: {Red, Green, Blue, Alpha}
texture1: {S, T, R, Q}
texture2: {S, T, R, Q}
.....
texture-n: {S, T, R, Q}
fog: {F}
specularity: {P}

A Vertex Shader can be considered a magic box—vertex data is fed into the box and different vertex data comes out. Every vertex that goes in comes out, but it may have changed while it was in the Vertex Shader box. Vertex Shaders do not create or delete vertices, but simply operate on, and change the values of, the data that describe each vertex. The vertex that emerges has a different position in space, a different color, is more or less transparent than it was before, or has different texture coordinates. All of these vertex changes, computed one vertex at a time, create the special effect for the overall object.

A specific vertex doesn't have to change as it goes through the Vertex Shader. A Vertex Shader may be programmed to change only vertices that have certain properties or it may be programmed to change every vertex in the same way. Programmers have the flexibility to use a Vertex Shader in one way on one object (creating one effect), and then reprogram the same Vertex Shader to apply an entirely different effect on the next object. (See Figure 2.) This programmability is discussed in more depth later in this paper.

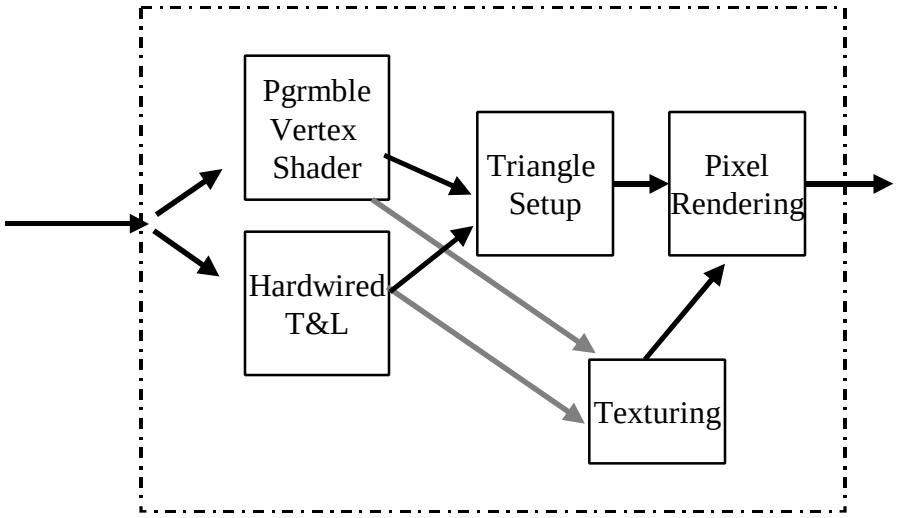
Figure 2 A Programmable Vertex Shader



Why Haven't Vertex Shaders Been Available Before Now?

Until now, Vertex Shaders were too complex to be included in GPUs. However, with the introduction of the NVIDIA nfiniteFX engine, programmable Vertex Shaders are now available, for the first time, in a GPU. Vertex Shaders are most powerful when combined with a programmable Pixel Shader. Pixel Shaders are discussed in a separate NVIDIA technical brief, *NVIDIA nfiniteFX Engine: Programmable Pixel Shaders*. For more information on the 3D graphics pipeline, readers may refer to Appendix A in this paper for a brief overview. Together, Vertex Shaders and Pixel Shaders offer software developers an unprecedented level of control and flexibility over the entire graphics pipeline.

Figure 3 The Graphics Pipeline Greatly Simplified



Mathematically, Vertex Shaders are a natural extension of the hardwired transformation and lighting engines of previous GPUs. The output of the Vertex Shader is a fully transformed and lit vertex. If a specific GPU has a hardwired transformation and lighting engine in addition to the programmable Vertex Shader, the hardwired engine will be idle when the Vertex Shader is used. This may seem extravagant, but it ensures better compatibility for legacy applications on the new hardware. GPUs that do not combine the fixed-function transformation and lighting engine with the Vertex Shader may suffer from incompatibility problems. Figure 3 shows the parallel relationship of the Vertex Shader and the traditional, hardwired transformation and lighting (T&L) engine.

The addition of Vertex Shaders to high-volume consumer and professional GPUs gives users access to this functionality for the first time. Vertex Shader operations are computationally complex and require such specific hardware structures that it is inefficient to use a microprocessor for these functions. If attempted on a microprocessor, performance is several times slower than on a GPU with a programmable Vertex Shader. That performance difference relegates Vertex Shader operations on microprocessors to offline rendering applications such as movie special effects. For users who demand fluid, interactive frame rates, the use of Vertex Shader operations requires a GPU with that capability.

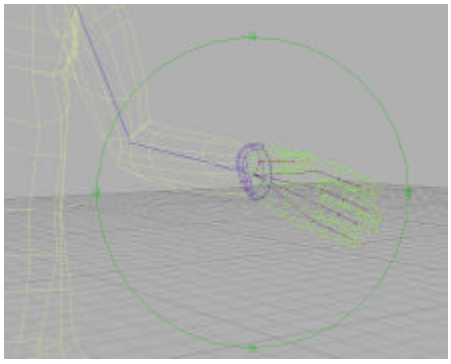
User Benefits and Effects Possible From a Programmable Vertex Shader

Programmable Vertex Shaders enable a virtually infinite list of visual effects without sacrificing real-time frame rates. While general-purpose microprocessors avoid specialization, GPUs with Vertex Shaders are architected to optimally process graphics functions. Specialized graphics processing units are superior to general purpose CPUs for graphics operations, measured in terms of pure performance, or architectural efficiency. The key categories of effects made practical by Vertex Shaders are described in the following sections, with many specific picture examples, grouped by the high-level user benefits.

Complex Character Animation

Vertex Shaders create skin and clothing more realistically. They stretch and crease properly at the joints like elbows and shoulders.

Facial animation can now include dimples or wrinkles that appear when the character smiles, and disappear when the smile disappears.



Lots of bones and muscles can be used to model characters as programmable Vertex Shaders allow up to 32 control matrices. That means up to 32 individual bones and muscles can be used to define individual components of a character's skeleton, with the possibility of hundreds or even thousands of total bones per character.

Keyframe animation uses a collection of control points whose locations are specified at various time intervals. The *frame* of animation at those specified moments in time are the *key*frames. Vertex Shaders calculate all of the animation in between the keyframes in real time—the animation is smooth and blends seamlessly with the pre-defined keyframes. Faster hardware will render more interim frames, creating smoother animation, but won't finish the animation sequence any sooner than slower hardware. This differentiates keyframe animation from morphing, which is described later in this paper.

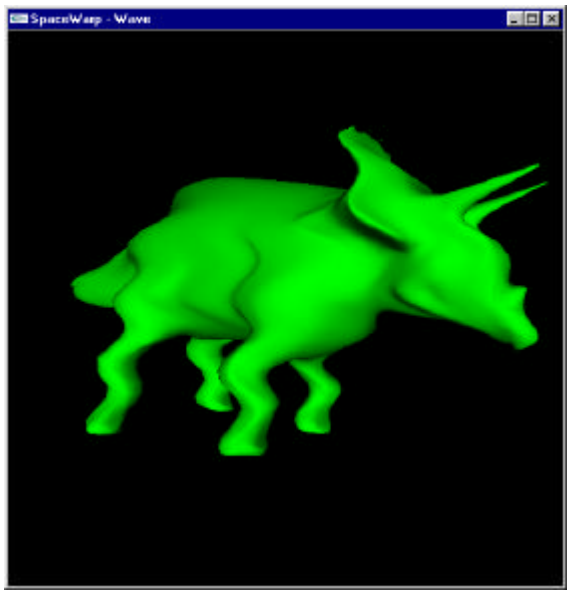
Environmental Effects

Elevation fog or smoke can mimic the thick fog that forms in valleys and other low-lying areas or even heavy smoke in a room. Large objects like hills, or smaller objects like a table and chairs, can stick up out of the fog/smoke because a Vertex Shader can apply effects selectively based on the height or elevation of each vertex in the object.

Image courtesy of Microsoft

Whether your scene includes a pond or an ocean, caustics and other refraction effects are essential for creating realistic water in a 3D scene. Note the pattern of light refracted from the unseen surface of the water above. Vertex Shaders can model the light refraction and/or project a texture in 3D space so that the light pattern falls on the objects realistically.

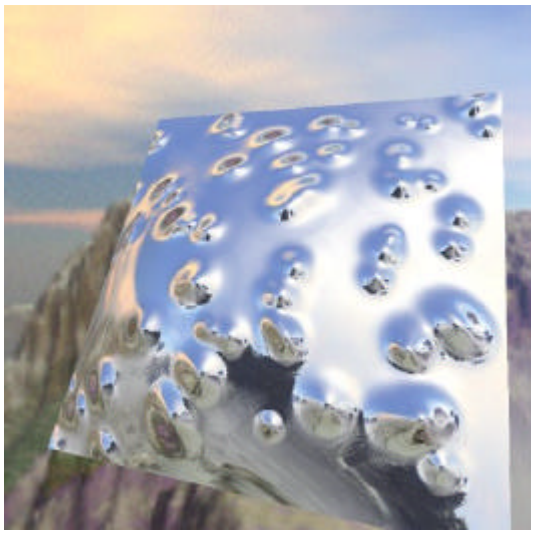
Heat-wave effects are another example of environmental effects enabled by Vertex Shaders. Imagine being in Death Valley, Arizona in August—you might hallucinate a stegosaurus like the picture at right.



Procedural Deformation

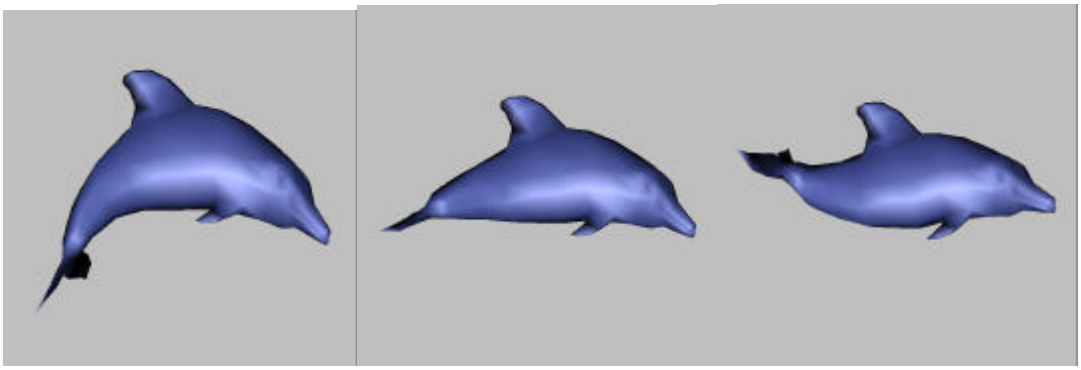
Procedural deformation, calculated by Vertex Shaders, can add movement to otherwise static objects. For example, a flag can wave in the breeze, or an animal's chest can expand and contract to simulate breathing.

The effect of procedural deformation can also be static rather than dynamic, like the bumps formed in a metal object from the impact of high-caliber bullets. Imagine a shooting game where the bullets leave lasting impressions, or a driving simulation that models realistic damage to your car after a collision.



Morphing

Morphing is another animation technique similar to keyframe animation. Using different versions of an object, the Vertex Shader blends the positions of each vertex on two images. For example, with the dolphins below, the Vertex Shader blends the position of each vertex in dolphin #1 with the positioning of the same vertices in dolphin #2 to create the middle, morphed dolphin. The middle dolphin exists only as a temporary blend of the other two. The morphed dolphin's geometry is never stored permanently. It is recreated immediately before it is needed. The result is a smooth animation sequence as the dolphin morphs from one tail position to the next. A morphing animation is usually not keyed to real-time events, so faster graphics hardware will result in more dolphin kicks per minute rather than more rendered frames per kick. Changing the blending parameters will affect the number of intermediate dolphins that are created between the tail-down and the tail-up versions. The speed of the animation can be controlled to make the dolphin's pace appear natural.

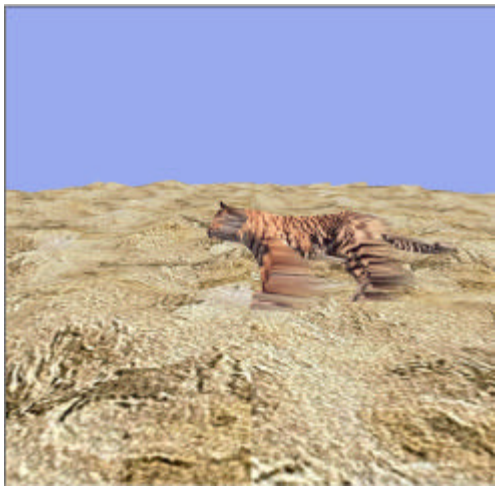


Dolphin #1 **Morphed Dolphin** **Dolphin #2**

Images courtesy of Microsoft

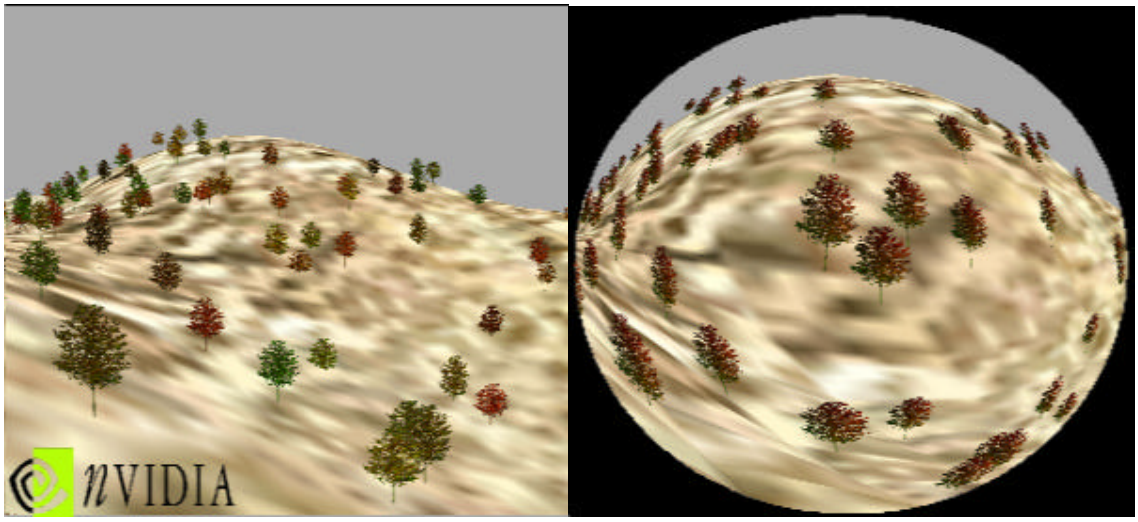
Motion Blur

Vertex Shaders can be used to create a variety of motion effects. Blurring an object creates an impression of super-speed like an action hero that moves at the speed of light or a space ship that is accelerating to warp speed.



Lens Effects

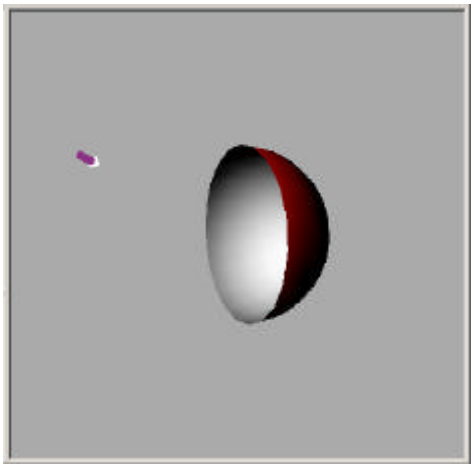
Custom transforms can be programmed into Vertex Shaders to produce the effects associated with optical lenses. Note the effect below: a normal transform on the left, and the fish-eye lens transform on the right. Whether simulating a view through a virtual security peephole in a front door, or the viewfinder for an international spy scooping out an enemy compound, a fish eye lens effect can heighten the realism of the 3D scene.



Other custom transform operations could simulate the thick glass of a World War II bomber's cockpit, the lens characteristics of a pool of water, and a variety of other visual effects.

Custom lighting effects

Two-sided lighting is another effect realistically delivered by Vertex Shaders. Not all GPUs are capable of lighting the back side of a triangle; the inside of any hollow object just doesn't show up on the screen. To solve this problem, artists must model the inside of the object to be displayed. That requires twice as many triangles for the same object. Two-sided lighting provides a better solution by allowing a single surface to have different lighting characteristics on either side



Programmability

The most powerful Vertex Shaders are programmable, with a complete instruction set and registers that a programmer can tailor to specific needs. Programmability is required to meet the Microsoft® DirectX™ 8 application programming interface (API) specification, and it enhances a Vertex Shader by enabling extensibility and re-configurability.

A programmable Vertex Shader is extensible because a programmer can write a new vertex program and achieve a new and unique result. An analogy would be ordering a meal in restaurant—do you choose your meal from a menu of standard fare or do you ask the chef to create something unique? A programmable Vertex Shader does not have specific effects hard-coded, like a fixed menu. It does have the proper ingredients—instructions and registers—so the chef (or programmer) can create a unique experience for the user.

Programmability in a Vertex Shader also increases the total computational power that can be applied to any specific task, because it is re-configurable. A programmable Vertex Shader can be programmed to process multiple tasks in parallel or to focus all of its computational capability on a single task for a fraction of a second before moving on to the next task. This is like having a team of people that can be divided up in two ways. Either each person performs a specific task, or the entire team can be applied to a single task in order to achieve something impossible to be done by one person. For example, imagine several people unloading several trucks. Each person might be responsible for unloading a single truck, but if there are objects that require multiple people to lift them, a single person could never unload a whole truck without help from others. The total amount of work done doesn't depend on how many people are applied to the task, but having enough people is essential. Likewise, some graphics operations such as 32-matrix skinning are so computationally complex that they are impractical to accomplish with dedicated hardware. It takes so many transistors to process the function, that if those same transistors can't be reconfigured (or reprogrammed) for other functions, the feature won't be implemented and won't be available. If your Vertex Shader is not programmable, you won't be able to do 32-matrix skinning. If your Vertex Shader is programmable, it can be dedicated entirely to a 32-matrix skinning operation because a split second later it can be re-configured for a different vertex processing function. This versatility is crucial for enabling the most complex special effects while keeping the overall graphics solution affordable.

Conclusion

The addition of programmable Vertex Shaders to consumer graphics processors shakes up the PC graphics market--visual quality takes a quantum leap forward. The NVIDIA nfiniteFX engine delivers a fully programmable vertex (and pixel) shading solution for real time 3D graphics. With the introduction of programmable Vertex Shaders, real-time 3D graphics content takes a major step towards cinematic realism, by offering content developers the ability to create their own programs, essentially special effects, to define new realities, and push the boundaries of image quality. Game players and other application users can enjoy stunning visual effects that were previously limited to pre-rendered video clips or movie screens, further blurring the line between the linear world of the movie, and the dynamic and interactive world of the interactive experience. Graphics and gaming will never be the same.

Appendix: The 3D Graphics Pipeline

The mathematical functions that must be performed to display 3D graphics are referred to as the 3D graphics pipeline. The complete 3D graphics pipeline is complex, but the major steps include:

Step 1: Scene Database Management

This step is not part of the 3D graphics pipeline, but it is mentioned here since it must be performed before all the other steps. Scene database management includes many application-level tasks (meaning they are done by the 3D application) such as knowing which objects should be in the scene and where they should be relative to other objects. The application is responsible for sending the necessary information about objects on the screen to the software driver for the GPU. Once the information has been sent to the GPU's driver, it can be thought of as having entered the 3D graphics pipeline and will proceed through the following steps. The driver then sends the information to the graphics hardware itself.

Step 2: Higher-order Surface Tessellation

Most objects in a 3D scene are constructed of triangles because triangles are easy for GPUs to process. Some other polygon types, such as straight lines or quadrilaterals, can be used but triangles are the most common. Some objects are defined using curved lines. These curved lines can be very complex mathematically because they require high-order formulas to describe them. A high-order formula is one that has a variable that is raised to a power, such as x^2. Examples of linear formulas would be $y = x+1$ or $y = 2x+1$. A similar example of a high-order formula would be $y = x^2+1$. Objects that are defined by high-order surfaces must be broken down into triangles before they can be sent to the next functional unit in the GPU. For this reason, the Surface Engine in a GPU is the first hardware function. Its purpose is to break higher-order lines and surfaces down into triangles.

Step 3: Vertex Shading including Transform and Lighting

Once an object is defined as a set of triangles (triangles are defined by specifying their vertices or the corners), the Vertex Shader function of the GPU is ready to do its job by applying custom transform and lighting operations. Because vertex shading is the subject of this paper, it is not described any further in this section other than the following general descriptions of transform and lighting.

Transform. As objects move through the 3D pipeline, they often need to be scaled, rotated or moved (translated) to make them easier to process or simply put them in the right place relative to other objects. The transform engine mathematically performs these scaling, rotation and translation chores using matrix multiplication.

Lighting. The lighting step is the calculation of lighting effects at each vertex of each triangle. This includes the color and brightness of each light in the scene and how it reacts with the color and specularity (glossiness) of the objects in the scene. These calculations are performed for every vertex in the 3D scene, so they are sometimes referred to as "vertex lighting."

Step 4: Triangle Setup

Triangle setup involves taking vertices and triangles and breaking them down mathematically into pixels or fragments. Note that fragments can be pixels or can be smaller than pixels. The sole purpose of this function is to take data as it comes out of the transform and lighting engine and convert it mathematically so the pixel shading engine can understand it.

Step 5: Pixel Shading and Rendering (including texturing)

Pixel shading and rendering include all of the complex pixel-level calculations to determine what the final color of each pixel should be. Information from the transform and lighting engine is used to determine what the pixel color should be based on the object color and the various lights in the scene. Next, the pixel shading and rendering functions must consider the additional changes to the pixel color based on what textures should be applied. These textures can describe color changes, lighting changes, reflections from other objects in the scene, material properties, and lots of other changes. For more detail on pixel shading technology, see the NVIDIA technical brief on the subject. The final task of the pixel rendering engine is to store the pixel in the frame buffer memory.

Step 6: Display

For the last stage in the 3D graphics pipeline, the display controller reads the information out of the frame buffer and sends it to the driver for the selected display (CRT, television display, flat-panel display, etc.).