ExtremeTech 3D Pipeline Tutorial

June 13, 2001
By: Dave Salvator

Introduction

From the movie special effects that captivate us, to medical imaging to games and beyond, the impact that 3D graphics have made is nothing short of revolutionary. This technology, which took consumer PCs by storm about five years ago, has its roots in academia, and in the military. In fact, in some sense, the 3D graphics we enjoy on our PCs today is a sort of "Peace Dividend," since many professionals now working in the 3D chip, film and game development industries cut their teeth designing military simulators.

Entertainment aside, 3D graphics in Computer Assisted Design (CAD) has also brought industrial design a quantum leap forward. Manufacturers can not only design and "build" their products without using a single piece of material, but can also leverage that 3D model by interfacing with manufacturing machinery using Computer Assisted Manufacturing (CAM). But for most of us, 3D graphics has made its biggest impact in the entertainment industries, both in film and in games, and its here that most of us became acquainted with the technology and jargon surrounding 3D graphics.

It is useful to note that not all rendering systems have the same goals in mind. Offline rendering systems, such as those used in CAD applications, stress accuracy over frame rate. Such models might be used to manufacture airplane parts, for instance. Real-time renderers, like game engines and simulators, tend to emphasize constant frame rate to keep animations smooth and fluid, and are willing to sacrifice both geometric and texture detail in order to do this.

Some renderers are a kind of hybrid, like those used in the *Toy Story* movies. Artists and programmers using Pixar's Renderman technology create visually stunning scenes, but each frame of animation might take hours to render on a server "farm"--a group of computers to which the rendering work is distributed. These offline-rendered frames are then sequenced together at 24 frames per second (fps), the standard film rate, to produce the final cut of the film.

With the rapid growth of consumer 3D chips' rendering power, the line between "consumer 3D" and "workstation 3D" has blurred considerably. However, real-time game engines do make trade-offs to maintain frame rate, and some even have the ability to "throttle" features on and off if the frame rate dips below a certain level. In contrast, workstation 3D can require advanced features not needed in today's 3D games.

The field of 3D graphics is expansive and complex. Our goal is to present a series of fairly technical yet approachable articles on 3D graphics technology. We'll start with the way 3D graphics are created using a multi-step process called a 3D pipeline. We'll walk down the 3D pipeline, from the first triangle in a scene to the last pixel drawn. We will present some of the math involved in rendering a 3D scene, though the treatment of 3D algorithms will be introductory. (We list a good number of references at the end of this story if you want to dive deeper).

In the Pipe, Five By Five

Because of the sequential nature of 3D graphics rendering, and because there are so many calculations to be done and volumes of data to be handled, the entire process is broken down into component steps, sometimes called stages. These stages are serialized into the aforementioned 3D graphics pipeline.

The huge amount of work involved in creating a scene has led 3D rendering system designers (both

hardware and software) to look for all possible ways to avoid doing unnecessary work. One designer quipped, "3D graphics is the art of cheating without getting caught." Translated, this means that one of the art-forms in 3D graphics is to elegantly reduce visual detail in a scene so as to gain better performance, but do it in such a way that the viewer doesn't notice the loss of quality. Processor and memory bandwidth are precious commodities, so anything designers can do to conserve them benefits performance greatly. One quick example of this is culling, which tells the renderer, "If the view camera (the viewer's eye) can't see it, don't bother processing it and only worry about what the view camera can see."

With the number of steps involved and their complexity, the ordering of these stages of the pipeline can vary between implementations. While we'll soon inspect the operations within these stages in much more detail, broadly described, the general ordering of a 3D pipeline breaks down into four sections: Application/Scene, Geometry, Triangle Setup, and Rasterization/Rendering. While the following outline of these sections may look daunting, by the time you're done reading this story, you'll be among an elite few who really understand how 3D graphics works, and we think you'll want to get even deeper!

3D Pipeline - High-Level Overview

1. Application/Scene

- Scene/Geometry database traversal
- Movement of objects, and aiming and movement of view camera
- Animated movement of object models
- Description of the contents of the 3D world
- Object Visibility Check including possible Occlusion Culling
- Select Level of Detail (LOD)

2. Geometry

- Transforms (rotation, translation, scaling)
- Transform from Model Space to World Space (Direct3D)
- Transform from World Space to View Space
- View Projection
- Trivial Accept/Reject Culling
- Back-Face Culling (can also be done later in Screen Space)
  Lighting
- Perspective Divide - Transform to Clip Space
- Clipping
- Transform to Screen Space

3. Triangle Setup

- Back-face Culling (or can be done in view space before lighting)
- Slope/Delta Calculations
- Scan-Line Conversion

4. Rendering / Rasterization

- Shading
- Texturing
- Fog
- Alpha Translucency Tests
- Depth Buffering
- Antialiasing (optional)
- Display

Where the Work Gets Done

While numerous high-level aspects of the 3D world are managed by the application software at the application stage of the pipeline (which some argue isn't technically part of the 3D pipeline), the last three major stages of the pipeline are often managed by an Application Programming Interface (API), such as SGI's OpenGL, Microsoft's Direct3D, or Pixar's Renderman. And graphics drivers and hardware are called

by the APIs to performing many of the graphics operations in hardware.

Graphics APIs actually abstract the application from the hardware, and vice versa, providing the application with true device independence. Thus, such APIs are often called Hardware Abstraction Layers (HALs). The goal of this design is pretty straightforward-- application makers can write their program once to an API, and it will (should) run on any hardware whose drivers support that API. Conversely, hardware makers write their drivers up to the API, and in this way applications written to this API will (should) run on their hardware (See Figure 1). The parenthetical "should's" are added because there are sometimes compatibility issues as a result of incorrect usages of the API (called violations), that might cause application dependence on particular hardware features, or incorrect implementation of API features in a hardware driver, resulting in incorrect or unexpected results.

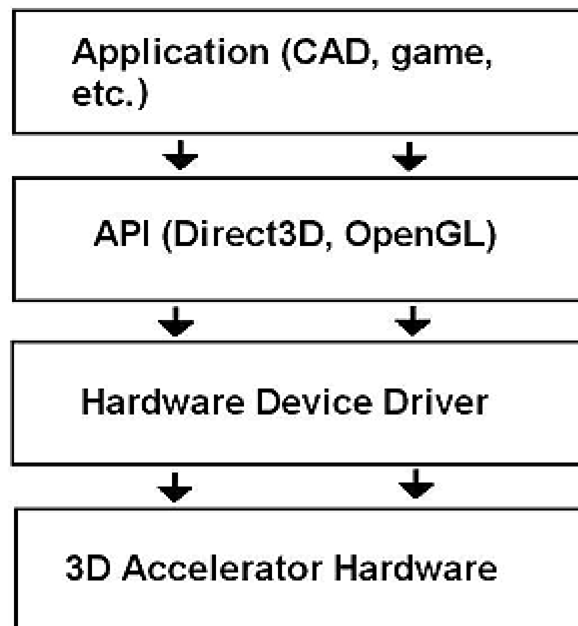
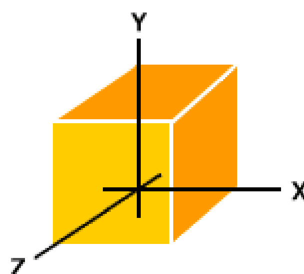


Figure 1 - 3D API stack

Space Invaders

Before we dive into pipeline details, we need to first understand the high level view of how 3D objects and 3D worlds are defined and how objects are defined, placed, located, and manipulated in larger three-dimensional spaces, or even within their own boundaries. In a 3D rendering system, multiple Cartesian coordinate systems (x- (left/right), y- (up/down) and z-axis (near/far)) are used at different stages of the pipeline. While used for different though related purposes, each coordinate system provides a precise mathematical method to locate and represent objects in the space. And not surprisingly, each of these coordinate systems is referred to as a "space."



Model of X-Y-Z Cartesian Coordinate system

Objects in the 3D scene and the scene itself are sequentially converted, or transformed, through five

spaces when proceeding through the 3D pipeline. A brief overview of these spaces follows:

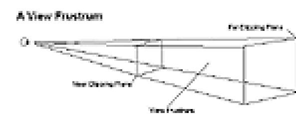
Model Space: where each model is in its own coordinate system, whose origin is some point on the model, such as the right foot of a soccer player model. Also, the model will typically have a control point or "handle". To move the model, the 3D renderer only has to move the control point, because model space coordinates of the object remain constant relative to its control point. Additionally, by using that same "handle", the object can be rotated.

World Space: where models are placed in the actual 3D world, in a unified world coordinate system. It turns out that many 3D programs skip past world space and instead go directly to clip or view space. The OpenGL API doesn't really have a world space.

View Space (also called Camera Space): in this space, the view camera is positioned by the application (through the graphics API) at some point in the 3D world coordinate system, if it is being used. The world space coordinate system is then transformed (using matrix math that we'll explore later), such that the camera (your eye point) is now at the origin of the coordinate system, looking straight down the z-axis into the scene. If world space is bypassed, then the scene is transformed directly into view space, with the camera similarly placed at the origin and looking straight down the z-axis. Whether z values are increasing or decreasing as you move forward away from the camera into the scene is up to the programmer, but for now assume that z values are increasing as you look into the scene down the z-axis. Note that culling, back-face culling, and lighting operations can be done in view space.

The view volume is actually created by a projection, which as the name suggests, "projects the scene" in front of the camera. In this sense, it's a kind of role reversal in that the camera now becomes a projector, and the scene's view volume is defined in relation to the camera. Think of the camera as a kind of holographic projector, but instead of projecting a 3D image into air, it instead projects the 3D scene "into" your monitor. The shape of this view volume is either rectangular (called a parallel projection), or pyramidal (called a perspective projection), and this latter volume is called a view frustum (also commonly called frustrum, though frustum is the more current designation).

The view volume defines what the camera will see, but just as importantly, it defines what the camera *won't* see, and in so doing, many objects models and parts of the world can be discarded, sparing both 3D chip cycles and memory bandwidth.



click on image for full view

The frustum actually looks like an pyramid with its top cut off. The top of the inverted pyramid projection is closest to the camera's viewpoint and radiates outward. The top of the frustum is called the near (or front) clipping plane and the back is called the far (or back) clipping plane. The entire rendered 3D scene must fit between the near and far clipping planes, and also be bounded by the sides and top of the frustum. If triangles of the model (or parts of the world space) falls outside the frustum, they won't be processed. Similarly, if a triangle is partly inside and partly outside the frustrum the external portion will be clipped off at the frustum boundary, and thus the term clipping. Though the view space frustum has clipping planes, clipping is actually performed when the frustum is transformed to clip space.

Deeper Into Space

Clip Space: Similar to View Space, but the frustum is now "squished" into a unit cube, with the x and y coordinates normalized to a range between –1 and 1, and z is between 0 and 1, which simplifies clipping calculations. The "perspective divide" performs the normalization feat, by dividing all x, y, and z vertex coordinates by a special "w" value, which is a scaling factor that we'll soon discuss in more detail. The perspective divide makes nearer objects larger, and farther objects smaller as you would expect when viewing a scene in reality.

Screen Space: where the 3D image is converted into x and y 2D screen coordinates for 2D display. Note that z and w coordinates are still retained by the graphics systems for depth/Z-buffering (see Z-buffering section below) and back-face culling before the final render. Note that the conversion of the scene to pixels, called rasterization, has not yet occurred.

Because so many of the conversions involved in transforming through these different spaces essentially are changing the frame of reference, it's easy to get confused. Part of what makes the 3D

"Part of what makes the 3D

pipeline confusing is that there isn't one "definitive" way to perform all
of these operations, since researchers and programmers have discovered different tricks and optimizations that work for them, and because there are often multiple viable ways to solve a given 3D/mathematical problem. But, in general, the space conversion process follows the order we just described.

To get an idea about how these different spaces interact, consider this example:

Take several pieces of Lego, and snap them together to make some object. Think of the individual pieces of Lego as the object's edges, with vertices existing where the Legos interconnect (while Lego construction does not form triangles, the most popular primitive in 3D modeling, but rather quadrilaterals, our example will still work). Placing the object in front of you, the origin of the model space coordinates could be the lower left near corner of the object, and all other model coordinates would be measured from there. The origin can actually be any part of the model, but the lower left near corner is often used. As you move this object around a room (the 3D world space or view space, depending on the 3D system), the Lego pieces' positions relative to one another remain constant (model space), although their coordinates change in relation to the room (world or view spaces).

3D Pipeline Data Flow

In some sense, 3D chips have become physical incarnations of the pipeline, where data flows "downstream" from stage to stage. It is useful to note that most operations in the application/scene stage and the early geometry stage of the pipeline are done per vertex, whereas culling and clipping is done per triangle, and rendering operations are done per pixel. Computations in various stages of the pipeline can be overlapped, for improved performance. For example, because vertices and pixels are mutually independent of one another in both Direct3D and OpenGL, one triangle can be in the geometry stage while another is in the Rasterization stage. Furthermore, computations on two or more vertices in the Geometry stage and two or more pixels (from the same triangle) in the Rasterzation phase can be performed at the same time.

Another advantage of pipelining is that because no data is passed from one vertex to another in the geometry stage or from one pixel to another in the rendering stage, chipmakers have been able to implement multiple pixel pipes and gain considerable performance boosts using parallel processing of these independent entities. It's also useful to note that the use of pipelining for real-time rendering, though it has many advantages, is not without downsides. For instance, once a triangle is sent down the pipeline, the programmer has pretty much waved goodbye to it. To get status or color/alpha information about that vertex once it's in the pipe is very expensive in terms of performance, and can cause pipeline stalls, a definite no-no.

Pipeline Stages--The Deep Dive

1. Application/Scene

3D Pipeline - High-Level Overview

1. Application/Scene

- Scene/Geometry database traversal
- Movement of objects, and aiming and movement of view camera
- Animated movement of object models
- Description of the contents of the 3D world
- Object Visibility Check including possible Occlusion Culling
- Select Level of Detail (LOD)

2. Geometry
3. Triangle Setup
4. Rendering / Rasterization

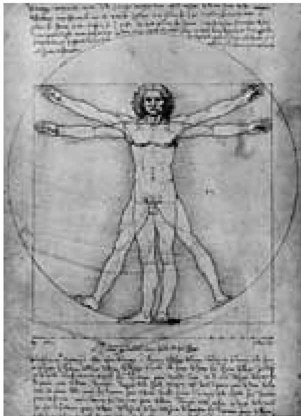
The 3D application itself could be considered the start of the 3D pipeline, though it's not truly part of the graphics subsystem, but it begins the image generation process that results in the final scene or frame of

animation. The application also positions the view camera, which is essentially your "eye" into the 3D world. Objects, both inanimate and animated are first represented in the application using geometric primitives, or basic building blocks. Triangles are the most commonly used primitives. They are simple to utilize because three vertices always describe a plane, whereas polygons with four or more vertices may not reside in the same plane. More sophisticated systems support what are called higher-order surfaces that are different types of curved primitives, which we'll cover shortly.

3D worlds and the objects in them are created in programs like 3D Studio Max, Maya, AutoDesk 3D Studio, Lightwave, and Softimage to name a few. These programs not only allow 3D artists to build models, but also to animate them. Models are first built using high triangle counts and can then be shaded and textured. Next, depending on the constraints of the rendering engine--off-line or real-time--artists can reduce the triangle counts of these high-detail models to fit within a given performance budget.

Objects are moved from frame to frame by the application, be it an offline renderer or a game engine. The application traverses the geometry database to gather necessary object information (the geometry database includes all the geometric primitives of the objects), and moves all objects that are going to change in the next frame of animation. Appreciate that in a game engine for instance, the renderer doesn't have the playground all to itself. The game engine must also tend to AI (artificial intelligence) processing, collision detection and physics, audio, and networking (if the game is being played in multiplayer mode over a network).

All models have a default "pose", and in the case of models of humans, the default pose is called its DaVinci pose, because this pose resembles DaVinci's famous Vitruvian Man. Once the application has specified the model's new "pose," this model is now ready for the next processing step.



click on image for full view

There's an operation that some applications do at this point, called "occlusion culling", a visibility test that determines whether an object is partially or completely occluded (covered) by some object in front of it. If it is, the occluded object, or the part of it that is occluded is discarded. The cost savings in terms of calculations that would otherwise need to be performed in the pipeline can be considerable, particularly in a scene with high depth complexity, meaning that objects toward the back of the scene have several "layers" of objects in front of them, occluding them from the view camera.

If these occluded objects can be discarded early, they won't have to be carried any further into the pipeline, which saves unnecessary lighting, shading and texturing calculations. For example, if you're in a game where it's you versus Godzilla, and the big guy is lurking behind a building you're walking toward, you can't see him (sneaky devil). The game engine doesn't have to worry about drawing the Godzilla model, since the building's model is in front of him, and this can spare the hardware from having to render Godzilla in that frame of animation.

A more important step is a simple visibility check on each object. This can be accomplished by determining if the object is in the view frustum (completely or partially). Some engines also try to determine whether an object in the view frustum is completely occluded by another object. This is typically done using simple concepts like portals or visibility sets, especially for indoor worlds. These are two similar techniques that get implemented in 3D game engines as a way to not have to draw parts of the 3D world that the camera won't be able to see. [Eberly, p. 413] The original Quake used what were called potentially visible sets (PVS) that divided the world into smaller pieces. Essentially, if the game player was in a particular piece of the world, other areas would not be visible, and the game engine wouldn't have to process data for those parts of the world.

Another workload-reduction trick that's a favorite among programmers is the use of bounding boxes. Say for instance you've got a 10,000-triangle model of a killer rabbit, and rather than test each of the rabbit model's triangles, a programmer can encase the model in a bounding box, consisting of 12 triangles (two for each side of the six-sided box). They can then test culling conditions (based on the bounding box vertices instead of the rabbit's vertices) to see if the killer rabbit will be visible in the scene. Even before you might further reduce the number of vertices by designating those in the killer rabbit model that are shared (vertices of adjacent triangles can be shared, a concept we'll explore in more detail later), you've already reduced your total vertex count from 30,000 (killer rabbit) to 36 (bounding box) for this test. If the test indicates the bounding box is not visible in the scene, the killer rabbit model can be trivially rejected,

you've just saved yourself a bunch of work.

You Down With LOD?

Another method for avoiding excessive work is what's called object Level of Detail, referred to as LOD. This technique is lossy, though given how it's typically used, the loss of model detail is often imperceptible. Object models are built using several discrete LOD levels. A good example is a jet fighter with a maximum LOD model using 10,000 triangles, and additional lower resolution LOD levels consisting of 5,000, 2,500, 1000 and 500 triangles. The jet's distance to the view camera will dictate which LOD level gets used. If it's very near, the highest resolution LOD gets used, but if it's just barely visible and far from the view camera, the lowest resolution LOD model would be used, and for locations between the two, the other LOD levels would be used.

LOD selection is always done by the application before it passes the object onto the pipeline for further processing. To determine which LOD to use, the application maps a simplified version of the object (often just the center point) to view space to determine the distance to the object. This operation occurs independently of the pipeline. The LOD must be known in order to determine which set of triangles (different LOD levels) to send to the pipeline..

Geometric Parlor Tricks

Generally speaking, a higher triangle count will produce a more realistic looking model. Information about these triangles--their location in 3D space, color, etc.--is stored in the descriptions of the vertices of each triangle. The aggregation of these vertices in the 3D world is referred to as a scene database, which is the very same animal as the geometry database mentioned above. Curved areas of models, like tires on a car, require many triangles to approximate a smooth curve. The adverse effect of aggressively curtailing the number of vertices/triangles in a circle, for example, via an LOD reduction would be a "bumpy" circle, where you could see the vertices of each component triangle. If many more triangles represented the circle, it would look far smoother at its edge. Optimizations can be made to reduce the actual number of vertices sent down the pipeline without compromising the quality of the model, because connected triangles share vertices. Programmers can use connected triangle patterns called triangle strips and fans to reduce vertex count. For example:

In the case of a strip of triangles, the simplest example would be a rectangle described by two right triangles, with a shared hypotenuse. Normally, two such triangles drawn separately would yield six vertices. But, with the two right triangles being connected, they form a simple triangle strip that can be described using four vertices, reducing the average number of vertices per triangle to two, rather than the original three. While this may not seem like much of reduction, the advantage grows as triangle (and resulting vertex) counts scale, and the average number of unique vertices per triangle moves toward one. [RTR, p. 234] Here's the formula for calculating the average number of vertices, given m triangles:

$$1 + 2/m$$

So, in a strip with 100 triangles, the average number of vertices per triangle would be 1.02, or about 102 vertices total, which is a considerable savings compared to processing the 300 vertices of the individual triangles. In this example, we hit the maximum cost savings obtainable from the use of strips for m number of triangles, which is m+2 vertices [RTR, p. 239]. These savings can really add up when you consider that it takes 32 bytes of data to describe the attributes (such as position, color, alpha, etc.) of a single vertex in Direct3D. Of course, the entire scene won't consist of strips and fans, but developers do look to use them where they can because of the associated cost savings.

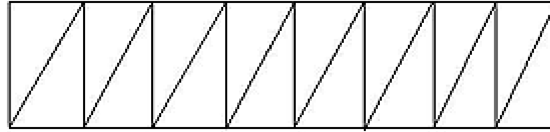
In the case of fans, a programmer might describe a semicircle using 20 triangles in a pie-slice arrangement. Normally this would consist of 60 vertices, but by describing this as a fan, the vertex count is reduced to 22. The first triangle would consist of three vertices, but each additional triangle would need only one additional vertex, and the center of the fan has a single vertex shared by all triangles. Again the maximum savings possible using strips/fans is achieved.

Another important advantage of strips and fans, is that they are a "non-lossy" type of data reduction, meaning no information or image quality is thrown away in order to get the data reduction and resulting speedup. Additionally, triangles presented to the hardware in strip or fan order improve vertex cache efficiency, which can boost geometry processing performance. Another tool available to programmers is the indexed triangle list, which can represent a large number of triangles, m, with m/2 vertices, about twice the reduction of using strips or fans. This representational method is preferred by most hardware

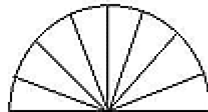
architectures.

Strips and Fans

Strip



Fan



Curved Surfaces Ahead

Rather than use numerous triangles to express a curved surface, 3D artists and programmers have another tool at their disposal: higher-order surfaces. These are curved primitives that have more complex mathematical descriptions, but in some cases, this added complexity is still cheaper than describing an object with a multitude of triangles. These primitives have some pretty odd sounding names: parametric polynomials (called SPLINEs), non-uniform rational b-splines (NURBs), Beziers, parametric bicubic surfaces and n-patches. Because 3D hardware best understands triangles, these curved surfaces defined at the application level are tessellated, or converted to triangles by the API runtime, the graphics card driver or the hardware for further handling through the 3D pipeline. Improved performance is possible if the hardware tessellates the surface after it has been sent from the CPU to the 3D card for transform and lighting (T&L) processing, placing less of a load on the AGP port, a potential bottleneck.

2. Geometry

3D Pipeline - High-Level Overview

1. Application/Scene
2. Geometry

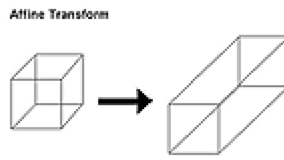
- Transforms (rotation, translation, scaling)
- Transform from Model Space to World Space (Direct3D)
- Transform from World Space to View Space
- View Projection
- Trivial Accept/Reject Culling (or can be done later in Screen Space)
- Back-Face Culling (can also be done later in Screen Space)
  Lighting
- Perspective Divide - Transform to Clip Space
- Clipping
- Transform to Screen Space

3. Triangle Setup
4. Rendering / Rasterization

Make Your Move: The Four Primary Transforms

Objects get moved from frame to frame to create the illusion of movement, and in a 3D world, objects can be moved or manipulated using four operations broadly referred to as transforms. The transforms are actually performed on object vertices using different types of "transform matrices" via matrix mathematics. All of these transform operations are affine, meaning that they occur in an affine space, which is a mathematical space that includes points and vectors. An affine transformation preserves parallelism of lines, though distance between points can change. (see illustration below, which shows a square being changed into a parallelogram with two sides shorter than the others, and parallelism is preserved, but the

angles in the object have changed). These transforms are used when moving objects within a particular coordinate system or space, or when changing between spaces.



click on image for full view

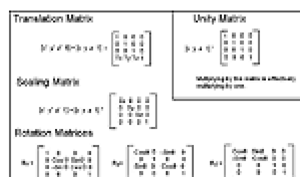
Translation: the movement of an object along any of the three axes to move that object to another location. Math operation: normally would be addition or subtraction (adding a negative number), but for efficiency purposes, transforms are done such that this operation winds up being matrix multiplication, like rotation and scaling operations.

Rotation: as the name suggests, an object can be rotated about an arbitrary axis. Math operation: in the simplest case where the object to be rotated is already positioned at the origin of the coordinate system, the multiplication of each coordinate by either the sine or cosine of θ (theta), which is the number of degrees by which the object is being rotated, produces the new post-rotation coordinates for a vertex. If an object's movement requires it to be rotated around more than one axis (x, y, or z) simultaneously, the ordering of rotation calculations is important, as different ordering can produce different visual results. Rotation around an arbitrary axis requires some extra work. It may first require a transform to move the object to the origin, then some rotations to align the arbitrary axis of rotation with the z-axis. Next the desired rotation is performed, then the alignment rotations must be undone, and the object must be translated back to its original location.

Scaling: the resizing of a model, used in creating the effect of depth of field during a perspective projection (a kind of transform that we'll soon cover in more detail). Math operation: multiply by a scaling factor, say 2 for all three axes, which would double the size of the object. Scaling can be uniform, where all three axes are scaled equally, or each vertex can be scaled by a different amount. Negative scale factors can produce a mirror image reflection of the object.

Skewing: (also called Shearing) the changing of the shape of a model, by manipulating it along one or more axes. An example might be a rendered scene of a cube of Jell-O on a plate sitting on a table, where the plate is turned up in the air 45° on its side, while the other edge remains on the table, and the cubic piece of Jell-O will now become rhomboid as a result of the pull of gravity. The top part of the model of the Jell-O cube would be skewed toward the ground of the 3D world. Math operation: adding a function of one coordinate to another, for example, add the x value to the y value and leave the x value alone.

Take a look at these diagrams:



click on image for full view

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Concatenation: The combination of matrices. Shown is a rotation about the x-axis and a translation.

As mentioned, matrix math figures large in doing 3D transform calculations. Matrices provide simple representation of transform terms, and 3D matrix math is typically comprised of straightforward multiplies or additions.

Transform processing efficiency comes from the fact that multiple matrix operations can be concatenated together into a single matrix and applied against the vertex as a single matrix operation. This spreads the matrix operation setup costs over the entire scene. One would think that given the three axes used in the various spaces utilizing the Cartesian coordinate system that this matrix should be 3x3. Instead it uses "homogenous coordinates" and is 4x4 in size. Homogenous coordinates permit the certain transforms that would otherwise require addition, such as a simple translation, to be handled as multiply operations. Complex concatenation of multiple different transforms using multiplies is also made possible with homogenous coordinates. The 4th term is actually the scaling factor "w". It is initially set to 1, and is used to compute a depth value for perspective projection in clip space as previously mentioned, and perspective-correct interpolation of pixel values across the triangle in the Rasterization stage.

Space to Space
Upon entry into the geometry section, we will likely translate objects from model to world space, and from world to view space, or directly to view space from model space in some cases, as we mentioned previously. When converting from one coordinate space to another, many of the basic transforms may be used. Some might be as simple as an inverse translation, or be more complex, such as involving the combination of two translations and a rotation. For example, transforming from world space to view space typically involves a translation and a rotation. After the transform to view space, many interesting things begin to happen.

That's it for this first segment. In the next installment we'll wrap up geometry and look at lighting, clipping, triangle setup, and many aspects of rendering.

3D Pipeline Part II

June 21, 2001
By: Dave Salvator

In our first installment of the 3D Pipeline, we covered the basics of real-time 3D processing and presented a high-level overview of 3D pipeline operations. We then reviewed coordinate systems and spaces, described the application pipeline stage in detail, and covered portions of the geometry stage.

In this much larger second segment, we'll wrap up the geometry stage, including culling, lighting, clipping, and transforming to screen space. Then we'll fully explore triangle setup, and investigate rendering functions including shading, texture-mapping/MIP-mapping, fog effects, and alpha blending. For easy reference, the sections we will cover are in bold below.

3D Pipeline - High-Level Overview

1. Application/Scene

- Scene/Geometry database traversal
- Movement of objects, and aiming and movement of view camera
- Animated movement of object models

- Description of the contents of the 3D world
- Object Visibility Check including possible Occlusion Culling
- Select Level of Detail (LOD)

2. Geometry

- Transforms (rotation, translation, scaling)
- Transform from Model Space to World Space (Direct3D)
- Transform from World Space to View Space
- View Projection
- **Trivial Accept/Reject Culling**
- **Back-Face Culling (can also be done later in Screen Space)**
  Lighting
- **Perspective Divide - Transform to Clip Space**
- **Clipping**
- **Transform to Screen Space**

3. Triangle Setup

- **Back-face Culling (or can be done in view space before lighting)**
- **Slope/Delta Calculations**
- **Scan-Line Conversion**

4. Rendering / Rasterization

- **Shading**
- **Texturing**
- **Fog**
- **Alpha Translucency Tests**
- Shadowing
- Depth Buffering
- Antialiasing (optional)
- Display

Trivial Matters

The Geometry section also includes techniques to reduce the work that must be performed. The first step in reducing the working set of triangles to be processed is to cull ("select from a group") those that are completely outside of the view volume as we noted previously. This process is called "trivial rejection," because relative to clipping, this is a fairly simple operation.

A test is performed to determine if the x, y, and z coordinates of a triangle's three vertices are completely outside of the view volume (frustum). In addition, the triangle's three vertices have to be completely outside the view volume on the same side of the view frustum, otherwise it could possible for a part of the triangle to pass through the view frustum, even though its three vertices lie completely outside the frustum.

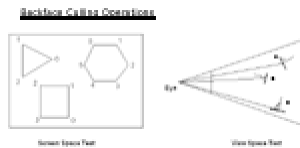
As an example, imagine a triangle that has two vertices on the left side of the frustum near the middle region of that side, and the third vertex is on the right side of the frustum located somewhere near the middle region of that side. All vertices are outside the frustum, but the triangle edges connecting the vertices actually pass through the frustum in a cross-sectional manner. If a triangle passes this test, the triangle is discarded.

If at least one (or two) of a triangle's vertices has all three of its coordinates (x, y, or z) inside the view volume, that triangle intersects the view volume boundaries somewhere and the portions falling outside the frustum will need to be clipped off later in the pipeline. The remaining portion of the triangle, now forming a non-triangular polygon will need to be subdivided into triangles (called retesselation) within the frustum. These resulting triangles will also need to be clip-tested.

The next operation is called back-face culling (BFC), which as the name suggests, is an operation that discards triangles that have surfaces that are facing away from the view camera. On average, half of a

model's triangles will be facing away from the view camera at any given time, and those triangles that are said to be back-facing, and can be discarded, since you won't be able to see them. The one exception where you would want to draw back-facing triangles would be if the triangles in front of them are translucent or transparent, but that determination would have to be made by the application, since BFC operations don't take opacity into account.

Back-face culling is one of those operations that can be done at different points in the pipeline. Some 3D texts, such as Foley-Van Dam, describe BFC being done in view space before lighting operations are done, for the obvious reason that by discarding these triangles, the hardware won't waste time lighting back-facing triangles that the viewer can't see. However, other texts, such as Moller-Haines' *Real-Time Rendering* shows that BFC can be done in either view or screen space. (Moller, T., Haines, E., *Real-Time Rendering* (RTR), (A.K. Peters, Natick, MA, 1999)



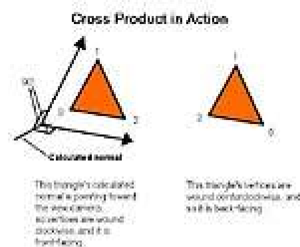
click on image for full view

Determining whether triangles are back facing depends on the space where the tests are done. In view space, the API looks at each triangle's normal. This is a vector that is pre-defined when the model is created in a program like 3D Studio Max or Maya, and is perpendicular to the surface of the triangle.

Looking from a straight-line viewing vector projected from the camera (in view space) to the center of a triangle where the normal vector originates, a measure of the angle between the normal vector and the viewing vector can be calculated. The angle is tested to see if it is greater than 90º. If so, the triangle is facing away from the camera, and can be discarded. If it's less than or equal to 90º, then it's visible to the view camera and cannot be thrown out. In other words, a check is made to see if the camera is on the same side of the plane of the triangle as the normal vector.

To determine which triangles are back facing in 2D screen space, a similar test can be done, wherein a face normal (a vector that is perpendicular to the surface of the triangle) is calculated for a projected triangle, and the test seeks to determine if this normal points at the camera or away from the camera.

This test is similar to the method used in 3D view space, but there are several differences. Rather than test the angle between the view vector and the triangle's face normal, this test instead calculates a face normal using a vector operation called a *cross product*, which is a vector multiplication whose result produces a third vector that's perpendicular to the plane of the original two, in other words, a face normal. In this case, the two vectors being fed into the cross product calculation are two sides of a triangle, and the direction the resulting normal determines whether the triangle is front- or back-facing.



click on image for full view

Once it has been determined which way the computed normal is facing, and because the API always uses the same vertex order to calculate the cross product, you can then know whether a triangles vertices are "wound" clockwise or counterclockwise (RTR, p.192). In other words, if you were to walk from vertex 0 to vertex 1 to vertex 2, which direction would you have traveled (Clockwise or Counterclockwise)? But you can't know which way a triangle's vertices are wound until you calculate the triangle's normal using the cross product operation. Take a look at this diagram to get a visual idea of what the cross product operation does.

Light of the World

The phrase "Transform and Lighting" has become standard parlance in 3D speak, and we don't often hear one without the other. But lighting doesn't happen after all transforms have occurred. It usually happens once the 3D scene has been transformed into view space. Once here, geometric lights can be applied to the objects in the scene. A geometric light is based upon very simplified lighting and reflection models, which often have little to do with how lights behave in the real world, but the net effect is deemed sufficient for the purposes of real-time 3D.

For example, objects are lit only by light sources, not by reflected/refracted light from other objects. As Watt bluntly puts it, this approach "…works because it works." Watt A., *3D Computer Graphics: Second Edition* (Addison-Wesley, New York, 1993). Detailed lighting calculations and models such as radiosity lighting using high-end 3D graphics (that is not real-time) can get very expensive very quickly on current hardware, and it's for this reason that most basic lighting models are instead a "good-enough-looking" approximation. For the purposes of 3D, there are two types of lights, global and local. Within these two broad categories, there are three types of lights:

- **Directional**: A global type of light, which is considered to be infinitely far away. If you want to see a good example of a directional light source, look up in the sky (assuming it's not a cloudy day). This light source's rays illuminate the entire scene, and its rays are cast parallel to one another. This is the most basic light source.
- **Point**: A local light source that has a defined point in space, and emits light in all directions. A spherical streetlight, or a torch mounted on a wall inside a castle would be examples.
- **Spot**: As the name implies, this local light source also has a defined point in space, and emits its light in a specific direction, in the shape of a cone.

Each of the above four factors are calculated using different math formulae, and the resulting values are i_{diff}, i_{spec}, i_{amb}, which are added together to obtain the total light value, i_{tot}. So:

$$i_{tot} = i_{amb} + i_{diff} + i_{spec}$$

When light shines on an object, the lighting effect is not only governed by the properties of the light, but also by the properties of the surface it's illuminating. So for this reason, both OpenGL and Direct3D APIs define material color properties that affect the final lighting result of a given surface. These properties are:

m_{amb}: Ambient Material Color
m_{diff}: Diffuse Material Color
m_{spec}: Specular Material Color
m_{shi}: Shininess Parameter
m_{emi}: Emissive Material Color

(RTR, p.68)

These material color properties also figure into lighting model equations, so even these "simplified" lighting models involve a good amount of hairy math to complete, particularly if many lights are being described. And although games have been somewhat slow to adapt hardware T&L, we've seen some recent titles that use it, like Microsoft's Combat Flight Simulator 2 and Papyrus' NASCAR 4. Hardware-based T&L (transform and lighting done on the 3D processor rather than the CPU) is a win for two reasons. In many cases, the T&L unit on the 3D processor can process more triangles and lights than the CPU, since the CPU also has to worry about running not only the application, but the entire system. Second, the graphics processor offloads the CPU from having to do these burdensome calculations, leaving the CPU free to tend to other chores.

Like many other areas of 3D, programmers have found clever workarounds to make lighting go faster, such as light maps, which are applied during the texture mapping stage of rendering, so we'll cover them there.

Is it possible to create light models that more closely resemble light's behavior in the real world? Yes. But, it's very expensive. Several techniques, including ray tracing and radiosity (mentioned earlier) can produce stunning results, if you're willing to wait, sometimes hours, sometimes days per frame. Ray tracing produces very impressive results, and is a "…partial solution to the global illumination problem, elegantly combining hidden surface removal, shading due to direct illumination, shading due to global illumination and shadow computation." (Watt, p. 268) Radiosity lighting is defined by Moller-Haines as "…a rendering method that computes the diffuse global illumination in a scene by letting the photons bounce around until an energy equilibrium is reached." (RTR, p.250) This method can render not only a light source's effect on an object, but reflected light from neighboring objects as well.

You may have heard terms like "per-vertex" and "per-pixel" lighting, with companies like nVidia singing the praises of the latter. That company produced demos for the GeForce 2 family of 3D processors that delivered some very impressive effects. You can download one of these demos at Nvidia's Web site

The advantage of per-pixel lighting is its granularity, especially true in low triangle count scenes with specular reflections where the realism of per-vertex lighting can diminish considerably. The obvious downside to per-pixel lighting is its considerably larger computational workload.

You've probably also heard of different types of shading that are related to lighting with terms like flat, Gouraud, Phong, and more recently Blinn shading (named after computer graphics legend Jim Blinn). Shading and lighting are closely linked, yet occur discretely at different points in the pipeline. But to better understand their relationship consider this explanation by nVidia's Chief Scientist Dave Kirk: "lighting is the luminance value, whereas shading is about reflectance and/or transmittance." These are related to lighting, but shading calculations occur later in the pipeline after rasterization, and we'll cover the topic later.

Getting Clipped and Gaining Perspective

At this point in Geometry processing, models have been animated, and put in their proper places in the scene, triangles outside of the view volume have been discarded through culling, and lighting calculations have been done. Now we move on to "clipping," which is the operation to discard only the parts of triangles that in some way partially or fully fall outside the view volume.

For clipping to occur with mathematical efficiency, the scene must be transformed from view space into clip space as we described earlier. With the transformation of the frustum into a unit cube, the clipping planes are now orthogonal (perpendicular) to the axes of the space. Recall the perspective divide occurs here, where x, y, and z are divided by w, a scale factor that represents distance of the vertex from the view point, so that objects further from the view camera are smaller.

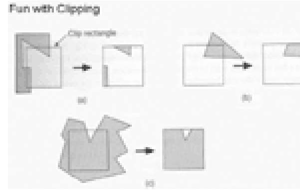
These scaled objects' relative size helps provide the illusion of depth of field in the scene. This is especially important in fooling our eyes into believing we're peering into a 3D world when we're really looking at a 2D image on a flat piece of glass. Not only do distant objects appear smaller in the real 3D world, it's also more difficult to discern object detail at a distance.

Programmers often write "pseudo-code," which describes what actions their program will perform. Note that pseudo-code is written in English, using little actual programming language syntax.

For example, a simple clipping algorithm would be:

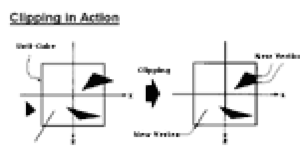
```
if (primitive isn't trivially rejected)
    and if (primitive is trivially accepted)
        draw it
    else
        clip it and draw that part of the
        triangle that's inside the view volume
```

The culling operation done back in view space has taken care of the first test, since all triangles that could be trivially rejected have already been discarded, so all remaining triangles are either completely inside the view volume, or need to be clipped. There are multiple approaches to doing clipping, and these step-by-step procedures are called algorithms, and are usually named for their inventors--Liang-Barsky, Cyrus-Beck-Liang-Barsky, Nichol-Lee-Nichol and Sutherland-Hodgeman to name several.



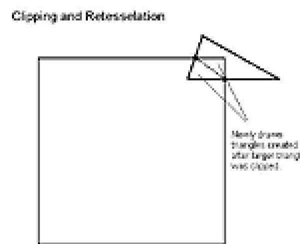
click on image for full view

Taking Sutherland-Hodgeman as an example, this algorithm can work in either 2D (four clipping boundaries) or in 3D (six clipping boundaries--left, right, top, bottom, near, far). It works by examining a triangle one boundary at a time, and in some sense is a "multi-pass" algorithm, in that it initially clips against the first clip boundary. It then takes the newly clipped polygon and compares it to the next clip boundary, re-clips if necessary, and ultimately does six "passes" in order to clip a triangle against the six sides of the 3D unit cube.



click on image for full view

Once a triangle has been clipped, it must be retesselated, or made into a new set of triangles. Take for example the diagram where you see a single triangle being clipped whose remaining visible portion now forms a quadrilateral. The clipper must now determine the intersection points of each side of the triangle with that clipping boundary, and then draws new triangles that will be part of the final scene.



click on image for full view

Another clipping method uses a "guard band" to further reduce the number of triangles that need to be clipped. To achieve this, two frusta are created--a guard band frustum and a screen frustum. A triangle that goes out of the screen frustum but stays within the larger guard band frustum doesn't need to be clipped. Triangles are trivially rejected from the screen frustum, trivially accepted within the guard band frustum, and failing both of those conditions, the triangle is clipped against the screen frustum.

Clip coordinates, are also referred to as *normalized device coordinates* (NDC), and after clipping operations have concluded, these coordinates are now mapped to the screen by transforming into screen space. Although *z* and *w* values are retained for depth buffering tests, screen space is essentially a 2D coordinate system, so only *x* and *y* coordinates need to be mapped to screen resolution. To map NDCs with a range of -1 to 1 to screen space, the following formula is generally used:

> Using 800x600 as a sample resolution, where:
>
> ScreenX(max) = 800
> ScreenY(max) = 600
>
> So…
>
> ScreenX = NDC(X) * 400 + 400

And

ScreenY = NDC(Y) * 300 + 300

To understand how this maps coordinates to the screen resolution, consider the minimum and maximum values for x and y, (-1, -1) and (1, 1) respectively.

Where NDC coordinates are (-1, -1)

ScreenX = -1 * 400 + 400 = -400 + 400 = 0

ScreenY = -1 * 300 +300 = -300 + 300 = 0

So screen coordinates are (0,0), the upper-left corner of the screen

And where NDC coordinates are (1, 1)

ScreenX = 1 * 400 + 400 = 400 + 400 = 800

ScreenY = 1 * 300 +300 = 300 + 300 = 600

So screen coordinates are (800, 600), the lower-right corner of the screen

This may seem a little counterintuitive, since positive x and y usually move to the right and up, respectively. Because of the direction of the vertical refresh (down), this mapping of y is more practical. But this is strictly a programming convention, and the origin (0,0) could be set at any of the four screen corners.

Triangle Setup: Setting the Table

Triangle Setup: Setting the Table

3D Pipeline - High-Level Overview

1. Application
2. Geometry
3. Triangle Setup
   - Back-face Culling (or can be done in view space before lighting)
   - Slope/Delta Calculations
   - Scan-Line Conversion
4. Rendering / Rasterization

Here's another place where the semantics sometimes get a bit murky. Some define the rasterization process as including triangle setup, whereas others view triangle setup as a separate step that precedes the rasterization stage of the pipeline. We're going to treat it in the latter manner. Notice that once again you can have overlapping operations here (as we'll see in a second) but for the sake of clarity, we're going to handle triangle setup separately.

We should also clarify two terms that sometimes get used interchangeably, and probably shouldn't: rasterization and rendering. Rasterization is a generic term that describes the conversion from a two-dimensional or three-dimensional vector representation to an x-y coordinate representation.

In the case of graphics cards (or printers), this process turns an image into points of color. A rasterizer can be a 3D card, or it can be a printer for that matter. But the process of rasterization is to assign color values to a given number of points to render the image, be it on a screen or a piece of paper. Here's where it gets murky. Rendering is the series of operations that determine the final pixel color displayed for a frame of animation, and while this can be thought of broadly as rasterization, the actual conversion of the 3D scene into screen-addressed pixels, or "real" rasterization, happens during triangle setup, also called scan-line conversion.

Think of triangle setup as the prelude to the rendering stage of the pipeline, because it "sets the table" for the rendering operations that will follow.

First off, the triangle setup operation computes the slope (or steepness) of a triangle edge using vertex information at each of edge's two endpoints. You may recall the equation of a straight line being y=mx+b, where y is the y-axis value, x is the x-axis value, b is the value of y when x=0 (the "y intercept"), and m is the slope (or the ratio of the rate of change between x and y values).

The slope is often called delta x/delta y, dx/dy, Dx/Dy, or literally change in x/change in y). Using the slope information, an algorithm called a digital differential analyzer (DDA) can calculate x,y values to see which pixels each triangle side (line segment) touches. The process operates horizontal scan line by horizontal scan line. The DDA figures out the x-value of the pixels touched by a given triangle side in each successive scan-line. (Watt, p. 143)

What it really does is determine how much the *x* value of the pixel touched by a given triangle side changes per scan line, and increments it by that value on each subsequent scan-line.

To actually calculate the y value of the triangle edge for a given integer value of x, as we move incrementally along the x axis one pixel at a time, we use the slope value. For every single pixel increment along the x-axis, we must increment the y-axis value of the triangle edge by Dy, which is equal to the slope m when x is incremented by one pixel.

Note that each scan line is the next incremental y coordinate in screen space. The y values of non-vertex points on the triangle edge are approximated by the DDA algorithm, and are non-integer floating-point values that typically fall between two integer y values (scan lines). The algorithm finds the nearest y value (scan line number) to assign to y.

This can be seen in the stair-step "jaggie" effect along edges that 3D systems try to reduce using higher resolution display or anti-aliasing techniques that we'll describe soon. Ultimately, the result of the DDA operation is that we now have x,y values for all scan line crossing points of each line segment in a triangle.

Taking this logic a step further, since nearly all 3D scenes are triangle-based, the rasterization operations are triangle-based as well. In carrying out its processing, the DDA generated the left- and right-hand edges of a triangle's intersection with a given scan-line. The portion of a scan line that bridges the two triangle edges is called a span (Watt, p. 143). Looking at the scan-line diagram will help this become clearer. We'll see how the span is used in the rendering section below.

A few other things happen here during the triangle setup phase worth noting. Specifically, color and depth values are interpolated for each pixel. Up until this point, only vertices have had color and depth information, but now that the triangle edge pixels are being created, interpolated color and depth values must also be calculated for those pixels.

These values are interpolated using a weighted average of the color and depth values of the edge's vertex values, where the color and depth data of edge pixels closer to a given vertex more closely approximate values for that vertex. In addition, the texture coordinates are calculated for use during texture mapping. Recall that the texture addresses define which part of the texture is needed on a specific part of a model--like a wood texture for a tree trunk--and are "pre-baked" into the model's data at each vertex. Similar to color and depth values, the texture coordinates are interpolated as well.

Now here's where we can get into some potential overlapping operations. In addition to interpolating the color and depth values, shading operations (which we cover next) can also be done during this operation as well.

Rendering / Rasterization

Rendering / Rasterization

3D Pipeline - High-Level Overview

1. Application

2. Geometry
3. Triangle Setup
4. Rendering / Rasterization
   - Shading
   - Fog
   - Alpha Translucency Tests
   - Shadowing
   - Antialiasing
   - Depth Buffering
   - Display

The rendering stage is probably one of the most interesting, because we begin to see a number of varying techniques for pixel rendering between the different graphics architectures. These differences can be relatively simple--Radeon's three pixel pipes versus GeForce 2's four--or more intricate differences in how chips do early Z buffering or hierarchical Z tests, all the way to fundamental architecture differences between brute-force triangle renderers like Radeon and the GeForce family, versus STMicroelectronics' Kyro I/II's tiling approach. We'll point these out along the way.

Made in the Shade
The rasterizer receives the two pixel endpoints per each scan line that a triangle covers from the setup stage, and calculates the shading values for each of the end pixels. Recall the span between these two pixel endpoints per scan-line described earlier. The rasterizer will shade the span based on various shading algorithms. These shading calculations can range in their demand from fairly modest (Flat and Gouraud), to much more demanding (Phong).

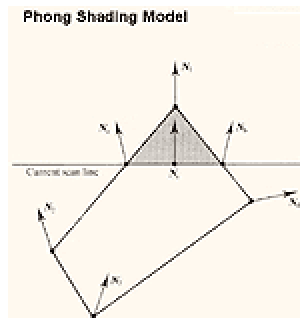
'Shading' is one of those terms that sometimes seems like a semantic football, as noted earlier, Dave Kirk, Chief Scientist at nVidia describes it this way: "Lighting is the luminance value, whereas shading is about reflectance or transmittance." The three most common shading methods, flat, Gouraud, and Phong operate per triangle, per vertex, and per pixel, respectively.

- **Flat Shading**: The simplest of the three models, here the renderer takes the color values from a triangle's three vertices (assuming triangles as primitive), and averages those values (or in the case of Direct3D, picks an arbitrary one of the three). The average value is then used to shade the entire triangle. This method is very inexpensive in terms of computations, but this method's visual cost is that individual triangles are clearly visible, and it disrupts the illusion of creating a single surface out of multiple triangles. (Lathrop, O., *The Way Computer Graphics Works*, Wiley Computer Publishing, New York, 1997)

- **Gouraud Shading**: Named after its inventor, Henri Gouraud who developed this technique in 1971 (yes, 1971). It is by far the most common type of shading used in consumer 3D graphics hardware, primarily because of its higher visual quality versus its still-modest computational demands. This technique takes the lighting values at each of a triangle's three vertices, then interpolates those values across the surface of the triangle (RTR, p. 68). Gouraud shading actually first interpolates between vertices and assigns values along triangle edges, then it interpolates across the scan line based on the interpolated edge crossing values. One of the main advantages to Gouraud is that it smoothes out triangle edges on mesh surfaces, giving objects a more realistic appearance. The disadvantage to Gouraud is that its overall effect suffers on lower triangle-count models, because with fewer vertices, shading detail (specifically peaks and valleys in the intensity) is lost. Additionally, Gouraud shading sometimes loses highlight detail, and fails to capture spotlight effects, and sometimes produces what's called Mach banding (that looks like stripes at the edges of the triangles)(RTR, p. 69).

- **Phong Shading**: Also named after its inventor, Phong Biu-Tuong, who published a paper on this technique in 1975. This technique uses shading normals, which are different from geometric normals (see the diagram). Phong shading uses these shading normals, which are stored at each vertex, to interpolate the shading normal at each pixel in the triangle (RTR, p. 68). Recall that a normal defines a vector (which has direction and magnitude (length), but not location). But unlike a surface normal that is perpendicular to a triangle's surface, a shading normal (also called a vertex normal) actually is an average of the surface normals of its surrounding triangles. Phong shading

essentially performs Gouraud lighting at each pixel (instead of at just the three vertices). And similar to the Gouraud shading method of interpolating, Phong shading first interpolates normals along triangle edges, and then interpolates normals across all pixels in a scan line based on the interpolated edge values.



click on image for full view

More recently, another per-pixel lighting model has come onto the scene using a technique called dot product texture blending, or DOT3, which debuted in the DirectX 6 version of Direct3D. A prelude to programmable shaders, this technique gains the benefit of higher resolution per-pixel lighting without introducing the overhead of interpolating across an entire triangle. This approach is somewhat similar to Phong shading, but rather than calculating interpolated shading normals for every pixel on the fly, DOT3 instead uses a normal map that contains "canned" per-pixel normal information. Think of a normal map as a kind of texture map. Using this normal map, the renderer can do a lookup of the normals to then calculate the lighting value per pixel.

Once the lighting value has been calculated, it is recombined with the original texel color value using a modulate (multiply) operation to produce the final lit, colored, textured pixel. Essentially, DOT3 combines the efficiencies of light maps, wherein you gain an advantage having expensive-to-calculate information (in the case of DOT3 per-pixel normals) "pre-baked" into a normal map rather than having to calculate them on the fly, with the more realistic lighting effect of Phong shading. the per pixel interpolators are used to interpolate the Phong normals across the triangle and DOT3 operations and texture lookups are used to compute the Phong lighting equation at each pixel.

Programmable Shaders--To Infinity and Beyond

One fact of life in creating games for the PC is that your run-time platform is a constantly moving target. A game console's hardware configuration is defined before the first box ships, and this configuration doesn't change for the lifetime of the console, so developers know exactly what hardware they'll have at run-time. Not so with the PC.

This inability to know exactly what hardware devices these applications (games, modeling tools, etc.) would be running on was one of two large driving factors that necessitated the creation of 3D APIs like OpenGL and Direct3D. By abstracting the hardware, an application could run on a variety of hardware, and conversely, one piece of hardware could run a variety of applications. The other was the need for developers to have common, known conventions for programming 3D so that they wouldn't have to re-invent the wheel each time they began work on a new project.

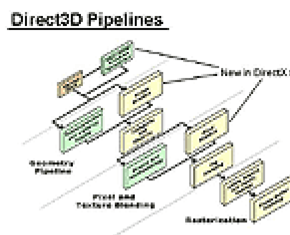
This model has served the graphics industry well over the years, but has several glaring shortcomings: first, APIs are often slow moving beasts that don't incorporate new features nearly as fast as hardware makers would like them to. As an example, Direct3D gets a refresh once every year to year-and-a-half. Second, because APIs only offer certain methods for implementing rendering effects, developers have often programmed around Direct3D in order to bring a unique aesthetic look to their games. Lighting is a good example of this.

Even with the ever-growing presence of T&L-enabled 3D accelerators, developers have been loath to use Direct3D's lighting models (and gain the hardware acceleration speedup) because they cannot achieve the desired effect using D3D's "canned" lighting model equations. In the case of game development, graphics programming is very much about pushing the envelope and showing the gaming world something it has never seen before.

This dilemma produced a need make Direct3D more malleable, so that developers could have the flexibility and creative possibilities they crave, and also take some pressure off of Microsoft to get the next version of the API out the door to incorporate the latest/greatest rendering features. Coincidentally, the generally accepted solution to making 3D hardware go fast-brute force multiple pixel pipelines applying multiple textures per clock cycle-had created a new problem: a severe scarcity of video memory bandwidth. As it turns out, a partial solution to the video memory bandwidth problem and the problem of API inflexibility dovetail in a single, intriguing solution.

DirectX 8's programmable vertex and pixel shaders give the Direct3D API a kind of "assembly language" of 127 commands, which developers can use to create custom shader programs that run on a 3D accelerator chip that supports these commands. The programs are capable of producing not only effects equal to those achieved using multiple textures applied on top of one another (which hammers video memory bandwidth), but also of creating stunning visual effects never seen before-the very thing game developers crave.

What's interesting about these shaders is that a DX8 part could implement its DX7 traditional pipeline as vertex and pixel shader programs, and then perform pre-DX8 rendering using these programs. As it turns out, programmers have to choose to use either the fixed DX7 pipe, or the programmable shader pipeline, but cannot mix these operations. To get an idea what the programmable shader pipeline looks like versus a traditional pipeline, look at this diagram:



click on image for full view

In addition to opening up creative possibilities for developers and artists, shaders also attack the problem of constrained video memory bandwidth by running these shader programs on-chip. Unlike multitexturing which requires multiple reads from texture memory AND multiple reads from and writes to frame buffer memory (the results of each texturing pass written to the frame buffer must be read back to apply the next texture), with shader programs, multiple textures can be read from texture memory and combined by the shader and applied in one texturing pass through the pipeline.,

The first step toward making shaders a reality was the release of DirectX, followed by nVidia's GeForce 3 chip. ATI currently has a DX8 part in the works that will ship later this year. There have been several very impressive demos showing the potential visual power of shaders including Massive Entertainment's Aquanox, MadOnion's 3D Mark 2001 and Zetha gameZ's Dronez. A partial list of effects that shaders will make possible includes:

Vertex Shaders:

- procedural geometry deformation
- range-based or radial fog effects
- camera lens effects including fish eye, wide angle, Fresnel and water refraction effects.

Pixel Shaders:

- per-pixel reflections
- per-pixel lighting using Phong-style shading or DOT3 effects
- procedural textures

A Tincture of Texture

Textures are hugely important to bringing realism to a 3D scene-- not only do they provide surface detail,

but they also give 3D visual cues that boost the illusion of scene depth. Here again, the semantic waters can get a bit murky, as texture mapping is much more than just slapping a digital decal on a wire-frame model.

As seen from the DOT3 example, texture mapping can also be a part of the scene lighting process, not to mention multitexturing effects. Texturing and texture cache management figure largely into the performance-or lack of it-of 3D accelerator chips. As previously mentioned, texturing is the first of the pixel operations.

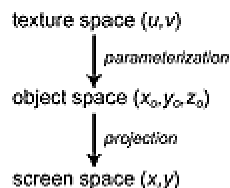
A texture can be different things: a bitmap of a surface like a brick wall, a light map, a gloss map, a shadow map, a bump map, an environment map, or even a surface for projecting an animation to simulate a television view screen to name a few.

The basic operation of texturing involves five steps:

1. Compute the object space location of the pixel to be textured.
2. Use a projector function to determine the correct (u, v) texture coordinate.
3. Use corresponder function(s) to find texel.
4. Apply value transform function.
5. Modify illumination equation value.

(RTR, p. 101)

Another way of thinking about the process is found in this three-step diagram:



click on image for full view

Textures are a kind of lookup table, and 2D textures (versus 3D textures), which are the predominate type of textures used in games today, use only two space coordinates, u and v, which are analogous to (x, y) coordinates. In the case of 3D or volumetric textures, two additional coordinates are available to describe those. Their names vary between OpenGL and Direct3D, but think of them as z and w coordinates for a 3D texture.

Each vertex in a triangle, in addition to its other data information, contains a texture coordinate (u, v), and the texture can then be interpolated across a triangle using these two texture coordinates. The actual "texture mapping" occurs when a portion of that 2D texture is mapped onto a surface in 3D space, which requires the above five-step process.

In real-time rendering systems, the projector functions are often applied at the modeling stage, and the resultant texture addresses are stored for each vertex.

Once the correct texture color has been fetched, the texel is transformed using corresponder functions from texture space (u, v) to screen space (x, y, z) and then applied to the object (RTR, p.102). One corresponder can be a matrix operation that can translate, scale, and/or rotate the texture onto the surface of the object.

Now remember that the pixel has already been shaded using one of the methods described above, and already has a color value. Once the texture color has been fetched from the texture map, the existing pixel color value is modulated (multiplied) by the texture color value to produce the resultant color value for that lit, shaded and now textured pixel.

A somewhat common problem is that the color obtained from the texture is dark, and even after

multiplying by the illumination value, it can still appear too dark. The problem is that colors and illumination values are represented as values between 0 and 1. Multiplying two such values produces darker values, closer to zero. For example, 0.5 x 0.5 is 0.25. To remedy this problem, Direct3D has a modulate 2X and 4X blending function that essentially "amplifies" the illumination to brighten pixels in a scene.

Once the correct texture has been applied to a given pixel the fun isn't over yet. There are many different ways to apply the texture to its pixel, and here we leave the realm of 3D graphics for a moment, and enter into the domain of sampling.

Texture Sampling

If you've ever studied digital audio, you've heard of a fellow named Nyquist, whose sampling theories developed way back in the 1930's are still used in digital signal processing today. In short, the Nyquist theory states that to accurately reproduce an analog waveform, you must digitally sample an analog signal at a rate at least twice its highest frequency to be able to represent the signal in the digital domain with a high level of accuracy.

You would then use digital-to-analog conversion techniques to reconstruct the original analog signal accurately. Using an audio example, to accurately sample a 22KHz tone, the upper limit of human hearing (though most adults can't hear above 17KHz), your sampling frequency would have to be 44KHz, which is the sampling rate used by CD players today (well, 44.1KHz to be precise, using 16-bit digital sample sizes). If an insufficient sampling rate is used, the result is aliasing, which is incorrectly sampled high-frequency information that appears as lower frequency noise. Using the CD example, if there was a sound at 23KHz, and your sampling rate was 44KHz (meaning your Nyquist limit was 22KHz), the 23KHz signal could not be correctly sampled and reproduced.

What, you may be asking, does sampling have to do with graphics? Assuming the case of mapping texture maps directly to screen pixel regions, where screen objects that are represented by a certain number of pixels use texture maps with a different number of texels (which is most often the case, because exact sized mappings are rare). When texture maps have more texels than the pixel space they are mapping (a single pixel can map to more than one texel) or when texture maps have fewer texels than the pixel space they are mapping (multiple pixels map to a single texel), display quality issues arise.

In effect, the texture map may either not be able to effectively map proper image detail to the designated pixel area, or conversely, a smaller pixel area may not be able use all the texture detail from a larger/richer texture map.

Specifically, in the case where a texture map is too small compared to the pixel area being mapped, then the same texel is mapped to adjacent pixels, and causes a blockiness effect when viewing the image. In the other case, if multiple texels map to the same pixel, it's often up for grabs which exact texel (from the group of texels that could map to a given pixel), will be selected from the texture map and applied to the pixel. It's algorithm-dependent and can result in artifacts known as texture swimming and pixel-popping, both of which are very noticeable when the camera moves.

Consider the latter case above, and assume we have a square billboard surface in a 3D scene made up of two equilateral (all three sides of same length) triangles. The dimensions of this square are 100x100 pixels. If you directly apply a texture map comprised of 200 horizontal red and blue stripes on top of that 100x100 pixel square area, you won't be able to discern all 200 stripes of the texture once it's applied to the billboard. The texture has more samples than the pixel area to which it will be applied. Some of the intermediate texture data will not be used in the mapping, and will essentially be ignored.

If you move your 3D viewpoint over the billboard, if you are navigating in a game for example, you'll see a phenomena called pixel popping, which is a result of the texture's stripes coming in and out of alignment with the billboard pixels. We've also seen this effect in computer-based football games on the field's yard-line markers, where the markers pop in and out of the field as the view camera moves across the field. This is a case of an insufficient texture-sampling rate to describe the scene's details.

There are several approaches to minimize the rendering artifacts caused by an insufficient texture-sampling rate, mostly in the form of filtering and re-sampling. The two filter methods used in the simple texturing methods like point-sampling (described below) are Magnification and Minification, which alter the texture sampling methods to help avoid the problems depicted above.

The minification algorithm is used in the case where multiple texels can map to a single pixel, and it selects the best fit texel from among the group texels that could map to the pixel. The magnification technique is used when multiple pixels can map to a single texel, and it maps a single texel to multiple pixel.

There are many methods varying in speed and quality for applying textures, including:

- Point sampling
- Bilinear filtering
- Trilinear MIP-mapping
- Anisotropic filtering
- Antialiasing

The most basic way to apply textures to a surface is point sampling, which uses a "nearest neighbor" sampling method. When the renderer fetches a piece of texture, it grabs the texture sample from the texture map that has u, v coordinates that map nearest to the pixel's coordinates (the pixel center), and applies it to the pixel. Though this approach requires the least amount of memory bandwidth in terms of the number of texels that need to be read from texture memory (one per pixel), the result often causes artifacts as we discussed above, owing to insufficient samples (screen pixels) to describe the texture. But even first-generation PC 3D hardware had a feature to help clean up, or at least hide these sampling problems to a certain extent: bilinear filtering.

Rather than just grab the nearest neighbor on a texture, bilinear filtering instead reads the four samples nearest to the pixel center, and uses a weighted average of those color values as the final texture color value. The weights are based on the distance from the pixel center to the four texel centers. The visual net effect is to blur out a good deal of the texture artifacts seen with point sampling, but because it's only a four-tap filter working with a single texture map, its effectiveness is limited.

Let 'Er MIP

Another approach to improving texture image quality is a technique known as MIP-mapping, where the renderer makes multiple copies of the original texture, and each successive MIP-map is exactly half the resolution of the previous one. This effectively becomes a kind of 3D texture, wherein you have the typical two coordinates, (u, v), but now a third coordinate, d, is used to measure which MIP-map (or maps) to select based on which map resolution will most closely match the pixel area to be mapped. In the case where a fighter plane is close to the view camera, a very detailed texture map would be selected, but as the plane flies off into the distance, successively smaller and less detailed maps would be used the farther away the plane files.

As the d coordinate increases, smaller and smaller MIP-maps are used. The derivation of the d coordinate is a bit complicated and implementation dependent, but relates to figuring out how much texture magnification or minification would result from a particular MIP-map selection based on the pixel area to be covered. The MIP-map with the smallest amount of texture magnification and minification would be selected.

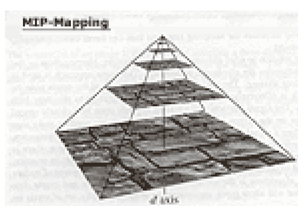
Bilinear MIP-Mapping
There are a few different ways applications can use MIP-maps, and they are often dependent on graphics hardware support. In some cases, an application may opt to select a single map that corresponds best to the pixel area to be mapped, apply bilinear filtering to texels in that map, and then apply the resulting averaged texels to their corresponding pixels.

In our Ziff Davis 3D WinBench test, we have tests that can use the NML (Nearest-MIP-map-Linear) type of MIP-mapping, which means it performs bilinear filtering within the Map (the L term), it performs MIP-mapping (the M term), and it uses only one MIP-map (the N term). Visual problems with bilinear MIP-mapping occur at map boundaries when moving through the scene. You may have played games where you drive down a roadway, and you see visible breaks in roadway texture. This is likely a due to a MIP-map switch.

Trilinear MIP-Mapping
A higher quality method of MIP-mapping often used is called trilinear MIP-mapping, and it helps alleviate the MIP-map boundary problem cited above. This filtering method takes two bilinear samples (using four

texel samples each) from the two MIP-maps nearest to the pixel (where one Map might be a bit larger in resolution and the other a bit smaller in resolution than the pixel area to be mapped).

The trilinear algorithm then uses a weighted average to combine the two bilinear-filtered texels into a resultant textured pixel, with the map whose resolution is closer to the pixel area getting more weight.



click on image for full view

But even trilinear MIP-mapping suffers some shortcomings, mostly relating to LOD biasing. Each MIP-map is said to be a level of detail (LOD), and biasing has to do with the weighting factor of the average between the two bilinear filtered samples taken from the two different maps.

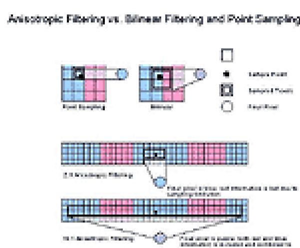
If LOD biasing is turned "up" too much, weighting the average more toward the smaller resolution map, texture blur can result. If the LOD bias is turned "down", weighting the average toward the larger MIP-map, aliasing, and more annoyingly temporal aliasing (texture shimmering/crawling) can result. LOD biasing plays with the d value, causing smaller or larger maps to be used. The real problem here is the isotropic nature of bilinear or trilinear filtering. The magnification/minification along one axis in the texture may be significantly different that in the other axis, which results in aliasing along one axis and blurring in the other. There's a filtering technique called anisotropic filtering that can address this issue, and we'll cover that in a minute.

But Wait, There's More

Trilinear MIP-mapping does a good job at filtering out texture shimmering and map boundary changes, but another more detailed, and yes, more computationally intense method is available called anisotropic filtering. According to Merriam-Webster, something that behaves anisotropically "exhibits properties with different values when measured in different directions."

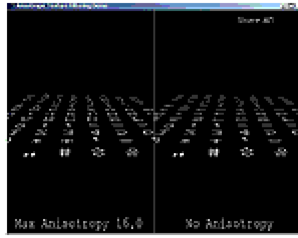
Anisotropic filtering uses MIP-map sampling, but rather than taking eight samples, it instead takes up to 16!! Think of that from a memory bandwidth perspective! In using anisotropic filtering, the renderer wants to know what the degree of anisotropy is, which is a ratio of how far a texture can be stretched before its image quality starts to go south.

In using more samples in the direction the texture would have to be stretched to fit the surface where it's being applied, anisotropic filtering achieves sharper image quality than bilinear or trilinear filtering, and also avoids the texture shimmering found with point sampling. Note too that anisotropic-filtered text that is displayed at an angle, as might be in a billboard in a racing game, renders more clearly than with other filtering methods. Anisotropic filtering will be used in future Windows GDI interfaces to allow multiple windows to display at angles on screen.



click on image for full view

Take a look at this screenshot and notice how the characters stay in focus across the surface.



click on image for full view

One More Time…

Both OpenGL and Direct3D have provisions for a single vertex to store two or more texture addresses for use in multitexturing effects, which have enjoyed widespread use dating back to Quake. Early on, multitexturing was used to create lighting effects using light maps, but it has since been applied to do additional effects like bump mapping. It was these multitexturing features that compelled all 3D chipmakers to invest chip real-estate into creating parallel pixel pipelines, and in more recent generations of chips, to process multiple texels per pixels per clock. For example, ATI's Radeon has three pixel pipes, each of which can process two texels per pixel per clock, so its theoretical texel fill rate running at 183MHz (64MB card's engine clock speed), yields nearly 1.1Gigatexels/sec, which is about 550Megapixels/sec in non-multitexturing applications.

The advantage gained by having parallel pipes capable of handling multiple texels per pixel is that multitexturing operations can often be done in a single clock cycle, rather than having to do multipass rendering, where each pixel in a scene is drawn several times. For each additional pass a chip has to make to render a scene, its effective pixel fill rate is divided by the number of passes that need to be made. So if a chip with 500Mpixels/sec of fill rate had to do two passes in order to render a scene with multitexturing effects in it, that chip's effective fill rate is halved to 250Mpixels/sec. If it makes four passes, the fill rate drops to 125Mpixels/sec. So it pays to have parallel pipes working for you.

Multitexturing doesn't operate that much differently from regular single texturing, except that the separate textures have to be blended together, and then blended with the existing pixel color produced by shading. However, doubling or tripling the number of textures being used in a scene drastically increases the amount of frame buffer bandwidth required to fetch the additional textures, and if multiple passes are being done, overall performance suffers yet again. But cards have gotten fast enough that the Quake 3 engine actually does 10 passes on faster systems to draw its scenes

Breakdown of the 10 passes of the Quake 3 Engine:

Passes 1-4: accumulate bump map
Pass 5: diffuse lighting
Pass 6: base texture (with specular component)
Pass 7: specular lighting
Pass 8: emissive lighting
Pass 9: volumetric/atmospheric effects
Pass 10: screen flashes

Source: Direct3D 7 IM Framework Programming-Multitexturing, Wolfgang Engel

At the time Quake 3 shipped, it was no great secret that many systems couldn't keep all of this rendering work moving at a reasonable clip for gaming. So the above-listed steps in parentheses could be omitted to preserve frame rate, reducing the number of passes all the way down to two (diffuse lighting and a base texture) if need be.

Another multitexturing technique that we mentioned earlier, and made famous by the Quake games, is the use of light maps. The light map option is a good substitute for systems with 3D graphics cards that don't support hardware T&L, or that have slower CPUs. This technique can be used when lights and objects are fixed in space, and is based on the fact that diffuse lighting is view-independent, and so rather than do the actual lighting calculations for all the lights in a scene, a texture map with diffuse lighting values can be

created, which functions as a lookup table. This light map is then multiplied by the texel color value to be applied to the object, and the resulting texels (texture elements) produce an object that appears to be lit.

The great part of this approach is that the light map's lighting information can actually be fairly elaborate, since they're "pre-baked" into the light map, and not computed on the fly. To apply the light map, the renderer would multiply the color value of the texel by the light value from the light map, and write the result to the back buffer, which is the next frame of animation that the renderer is preparing for display. This is called a modulation in programmer's parlance, and also is referred to as a blending operation (RTR, pp. 124-5).

Squeeze Play

Given the amount of frame buffer bandwidth and memory space textures can occupy, another area of economy that 3D chipmakers looked to was texture compression. S3 led the charge with their S3TC compression scheme, which was incorporated into DirectX 6. Now known as DXTC, this scheme allows textures to be stored and moved around in a compressed format, which is similar to JPEG compression, and decompressed only when needed for rendering, saving both frame buffer space and bandwidth. This compression scheme can generate between 4:1 and 6:1 compression ratios, and games like Unreal Tournament have incorporated it.

Other Textural Matters…

As mentioned earlier, texturing can be used to create a wide array of visual effects, with basic surface description being only the tip of the iceberg. Developers have also discovered another way to create textures-- rather than store a "canned" bitmap, a developer can instead create a small program that creates textures procedurally. This technique has been used to create anything from marble surfaces, to terrain textures in large outdoor environments, to moving water effects.

To see an example of a procedurally-generated texture, click here to see a demo created by Ken Perlin, who was one of the first programmers to develop this technique.

Foggy Notions

After texturing, we move onto the next of the pixel operations, the applying of fog.

One thing many applications, games in particular, seek to do with 3D is to set a mood. This is especially important to attempt to immerse gamers inside the world the developer has conjured up. One useful tool to achieve different moods is fog, which also helps give a scene an addition sense of depth of field. It also has some useful technical applications, as we'll see in a minute.

Fog gets implemented in several different ways, with the variations having mostly to do with how the fog becomes, well, foggier. Two very common methods are called linear and exponential fog, which as you may have discerned, scale as their names suggest. Fog can also be applied per vertex or per pixel, and as in most things in 3D, per pixel looks more convincing, but is computationally more expensive.

A more advanced method of applying fog is called range-based fog, which traditionally has not been available in consumer 3D chips, but with the arrival of vertex shaders in DirectX8, range-based fog will be an option. Another method is called fog table, where fog values are stored in a lookup table, then applied to each pixel (RTR, pp. 90-2).

But irrespective of technique, fog is a function of how far away an object is, which is usually determined by its z value, the distance from the view camera. A fog factor, whether computed linearly or exponentially, is then computed and applied to the pixel using a blending operation to combine the fog amount (color) and the lit/shaded/textured pixel color. If fog is being done per vertex, fog operations become a part of the lighting calculation, and are interpolated across each triangle using Gouraud shading (RTR, p.91).

Another practical use for fog is to allow objects at the far clipping plane (recall the frustum in view space) to gracefully "fade away" rather than just pop out of the scene. Fog can also be used to allow a renderer to only have to draw world objects relatively close to the view camera. So when the fog value gets near or reaches zero (fog values are floating-point values between zero and one, with zero being absolute fog and one being no fog) the object is essentially discarded, and no further rendering work needs to be done. One game that uses this technique is Motocross Madness 2.

Alpha Opacity Tests and Blending
Next up in the rendering phase of the pipeline is alpha testing and blending. Alpha values gets used for a variety of effects, including glass, water, see-through views in CAD modeling, and translucent or transparent materials. Alpha values are normally stored per vertex, with the data format being RGBA, each with an 8-bit value when 32-bit rendering is used. The alpha value determines a vertex's opacity, and is always between zero and one, where one is completely opaque and zero is completely transparent. Anything in between is some degree of translucency.

It turns out that like lighting, transparency/translucency effects in 3D graphics are pretty severe simplifications of real-world behaviors of light refraction and reflection through transparent or translucent objects (RTR, p.85), but as Moller and Haines put it: "a little transparency is better than none at all." (RTR, p. 86)In fact, refraction or translucency based upon a surface's thickness isn't normally factored in at all.

And we digress again… when we spoke with David Kirk from Nvidia about the near future of rendering three to five years out, he said the big improvements won't necessarily be in the number of pixels/sec (though they will increase of course), but rather in the quality or "intelligence" of those pixels. He said we'll see far more advanced programmable per-pixel shading and lighting algorithms that take into account material properties, curved surfaces, use volumetric textures, and have far more complex single-pass texture blending. He describes shifting the battle "…from more pixels to smarter pixels."

Unlike opaque objects, transparent objects must generally be depth sorted in back-to-front order to ensure that the underlying colors that are blended with the transparent objects are available when the blending operation is performed. There are several formulae that get used to calculate an object's translucency, but a common formula is:

co = a x cs + (1 - a) x cd (RTR, p. 86)

co = final color of pixel
a = alpha value (between 0 and 1)
cs = color of the transparent pixel (called the source)
cd = color of the occluded pixel (called the destination)

This formula is a sum of two pixel modulations, where the alpha value is applied (differently) to both pixels, and the resultant values are added together to give the final pixel color.

Another common usage for alpha is in what's called alpha mapping, which combines alpha effects with a texture. It's particularly useful for creating odd-shaped textures or decals, since most textures are either square or rectangular. For instance, if you had a model of Homer J. Simpson, and wanted to show him wearing a different dirty T-shirt each day, you could use an alpha-mapped texture to put different "iron-on" decals on his shirt from day to day, say, different versions of the Duff beer logo, or text saying "I'm with stupid." Where you didn't want this decal to affect the underlying texture, you'd set an alpha value to zero, and only those parts of the texture that had non-zero alpha values would appear on the shirt. This is somewhat similar to chroma-keying that's used by weathermen in TV broadcasts when they're standing in front of a weather map giving us yet another incorrect weather forecast.

That's it for Part II. Tune into our exciting final episode where we'll get deeper into rendering functions and cover shadowing, antialiasing, Z-buffering, and finally, output to the display.

Bibliography

These texts, in addition to being referred to in our story, are excellent resources for more information about 3D pipelines, programming, and deeper explanations of the math that makes 3D graphics go.

Arnaud, R., Jones, M., *Image Generator Visual Features Using Personal Computer Graphics Hardware*, presented at IMAGE 2000 Conference, July, 2000

ATI, www.ati.com/na/pages/technology/hardware/radeon/visual_details.html#17

Blinn, J., *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, Morgan Kaufman, San Francisco, 1996Blinn, J., *W Pleasure, W Fun*, IEEE Computer Graphics and Applications, May/June 1998

Bloom, Charles, *"The Genesis Terrain Renderer Technology"*
www.cbloom.com/3d/techdocs/terrain_tech.txt

Dietrich, S., *Dot Product Texture Blending and Per-Pixel Lighting*, nVidia Technology Paper

Eberly, D., *3D Game Engine Design*, Morgan Kaufmann, San Francisco, 2000.

Foley, T., van Dam., A., Feiner, S., Hughes., J., *Computer Graphics: Principles and Practice, 2nd Edition*, Addison Wesley, New York, 1997

Engel, Wolfgang, *Direct3D 7 IM Framework Programming—Multitexuring*,
www.gamedev.net/reference/programming/features/d3d7im3/page2.asp

Glassner, A., *3D Computer Graphics, 2nd Edition*, Design Books, New York, 1989

Gouraud, H., *Continuous Shading of Curved Surfaces*, IEEE Transactions on Computers, vol. C-20, June 1971

Heckbert, P., *Survey of Texture Mapping*, IEEE Computer Graphics and Applications, Nov 1986

Kugler, A., *The Setup for Triangle Rasterization*, presented at the 11th Eurographics Workshop on Computer Graphics Hardware, August, 1996, Potiers, France

Lathrop, O., *The Way Computer Graphics Works*, Wiley Computer Publishing, New York, 1997

Miller, T., *Hidden-Surfaces: Combining BSP Trees with Graph-Based Algorithms*, Brown University site, The Science and Technology Center for Computer Graphics and Scientific Visualizations

nVidia White Paper, *GeForce 3: Lightspeed Memory Architecture*, nVidia Corporation

nVidia White Paper, *Tranform and Lighting*, nVidia Corporation

Phong, B., *Illumination for Computer Generated Pictures*, Communications of the ACM, vol. 18, no. 6, June 1975

PowerVR White Paper, *PowerVR Tile-based rendering*, April 2001

Möller, T., Haines, E., *Real-Time Rendering*, A.K. Peters, Natick, MA, 1999

Watt, A., *3D Computer Graphics: Second Edition*, Addison-Wesley, New York, 1993

Wright, R., Sweet, M., *OpenGL Superbible*, Waite Group Press, Corte Madera, CA, 1996

Further Resources

Further Resources

Resource Title and Info	The Skinny
Real-Time Rendering Thomas Möller and Eric Haines, published by A.K. Peters, Ltd., ISBN# 1-56881-101-2	One of the very best resources for 3D programmers and even 3D enthusiasts who want to dig deeper on 3D. This book also does a good job of presenting most of the rudimentary math involved in 3D graphics. If you're going to buy one book about 3D, make it this one. Also check out their companion web site, www.realtimerendering.com.
3D Computer Graphics, 3rd Edition Alan Watt, published by Addison-Wesley,	A very well written book, though it's specifically intended for programmers. Includes a CD-ROM with code examples and information about algorithms.

ISBN# 0-201-63186-5	
The Way Computer Graphics Works Olin Lathrop, published by Wiley Computer Publishing, ISBN# 0-471-13040-0	A well written, easy to understand book aimed at end-users looking to start down the path to 3D righteousness.
Computer Graphics: Principles and Practices, 2nd Edition Foley, Van Dam, Feiner & Hughes, published by Addison Wesley, ISBN: 0-201-84840-6	The granddaddy of academic graphics texts, this is the Rosetta Stone of 3D. This book is heavy on theory and math, and assumes a fairly solid background in the math required to program 3D. Not light reading.
3D Game Engine Design, David Eberly, published by Morgan Kaufman Publishers, ISBN: 1-55860-593-2	A good resource for programmers looking to write their own engine. Includes many examples written in C++, and a CD-ROM containing all source code used in the book, as well as source code for a game engine.
Jim Blinn's Corner : A Trip Down the Graphics Pipeline Jim Blinn, published by Morgan Kaufman Publishers, ISBN: 1558603875	A collection of writings compiled over the years from Jim Blinn's columns in the IEEE Computer Graphics and Applications journal. Aimed primarily at progammers, Blinn's unique take on 3D is still pretty accessible for the non-programming universe.

3D Pipeline Part III

June 27, 2001
By: Dave Salvator

This is the third and final installment in our detailed tutorial on the 3D pipeline. In this, the final installment of our three-part pipeline odyssey, we'll get deeper into rendering functions such as shadowing, antialiasing, Z-buffering, and finally, output to the display.

In Part 1, we covered the basics of real-time 3D processing and presented a high-level overview of 3D pipeline operations. We then reviewed coordinate systems and spaces, described the application pipeline stage in detail, and covered portions of the geometry stage. Part II wrapped up the geometry stage (including culling, lighting, clipping, and transforming to screen space), and explored triangle setup, shading, texture-mapping/MIP-Mapping, fog effects, and alpha blending.

1. Application
2. Geometry
3. Triangle Setup
4. Rendering / Rasterization
   - Shading
   - Texturing
   - Fog
   - Alpha Translucency Tests
   - Shadowing
   - Depth Buffering
   - Antialiasing
   - Depth Buffering

- Display

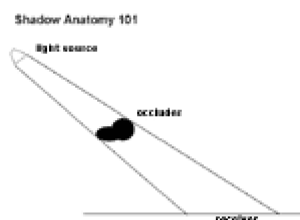
Lurking in the Shadows

As much as lighting can create a mood in a scene, it's often the rendered shadows that further define/refine that mood, and give an object further lifelike qualities. Say you're playing a game like FIFA 2001 where you're in a night match, and each player is casting four shadows (from the overhead lights). This seemingly minute detail actually helps further set the scene of a night game.

In other games like the Thief series, if you failed to make good use of the shadows to conceal yourself, you'd be history. Shadows also help prevent the appearance of objects floating above the surface to which they're supposedly attached, according to A. Watt's *3D Computer Graphics: Second Edition* (Addison-Wesley, New York, 1993). For as hard as it is to create and apply realistic and appropriate lighting to a scene, it can be equally challenging to render convincing shadows. Similar to lighting and transparency effects in 3D, even a mediocre shadow effect is often better than none at all.

Here's a brief anatomy of shadows: First, a per-vertex or per-pixel light source must exist. Next, the light source must strike an object that casts a shadow, called the *occluder*. Finally, the surface onto which the shadow is being cast which is the *receiver*. Shadows themselves have two parts, called the *umbra*, which is the inner part of the shadow, and the *penumbra*, which is the outer and border portion of the shadow (recall your basic eclipse terminology).

The penumbra creates the difference between hard and soft shadows. With hard shadows, the shadow ends abruptly, and looks unnatural, whereas with soft shadows, the penumbra transitions from the color of the shadow (usually black) to the adjacent pixel color, creating a more realistic shadow. Watt lists at least five possible methods for rendering shadows, but for the purposes of this article, we'll cover projected shadows, and shadow volumes that use the stencil buffer.



click on image for full view

Technique #1: Projected Shadows
Projected shadows, as the name suggests, are created by having a light source act as a projector, which then "projects" a shadow onto the receiver surface. A projected shadow polygon results on the receiver surface. The projected shadow polygon is not necessarily a triangle as it depends on the shape of the occluder. The projected polygon is then rendered with a dark color, or no illumination, to give the shadowing effect, as described by Möller and Haines in *Real-Time Rendering (RTR)*, (A.K. Peters, Natick, MA, 1999), pp. 169-170.

One downside to this method is that the receiver has to be a planar (flat) surface, or serious rendering errors can occur. One speedup technique is to render the projected shadow polygon into a shadow texture, which can be applied to the receiver surface, and subsequently reused, providing that neither the light source nor the occluder moves, as shadows are not view- or scene-dependent.

A variation of this technique can be used to create soft shadows. By using multiple light sources, programmers can create soft shadows by having the "central" light sources first project the hard shadow (umbra). Then, "satellite" light sources project soft shadow (penumbra) effects, attenuating the shadowing effect (color darkening) as distance from the occluder increases. But in order to use this method, objects must be rendered a second time to do the projection calculations, which adds both computational and frame buffer bandwidth overhead.

Still, this method is fairly inexpensive, and works fine so long as your shadow is being projected onto a planar (flat) surface. [RTR, p. 167]

Shadow Volumes

Technique #2: Shadow Volumes
Shadows volumes are an increasingly popular way to create shadow effects, and this technique makes use of a stencil buffer. A stencil buffer is an area of video memory that contains one to eight bits of information about each pixel of the scene, and this information can be used to mask certain areas of the scene to create shadow effects.

Here's how it works:

Shadow volumes create a separate frustum, and place the point light source at the top of the frustum and project into it. The resulting intersection of the shadow frustum and the view frustum creates a cylindrical volume inside the view frustum. Polygons that fall within this cylindrical volume will cast shadows upon receiver objects (of any shape) that are aligned with the direction the light rays that are being cast from the shadow-generating object. The actual shadow volume is the 3D volume created between the edges of the shadow caster (the occluder), and the receiver surface(s) the shadow will affect.

Although this bears some similarity to the projected shadows technique mentioned above, there's an important difference: rather than being able to only cast shadows on flat receiver surfaces, the shadow volume technique can cast shadows on any object. The actual process is pretty involved, particularly the 'sausage-making' part of creating shadows.

- [Source]
  The gory details are beyond the scope of this article, but here's a 15-step pseudo-code example from a 1998 paper by OpenGL guru Mark Kilgard of SGI:

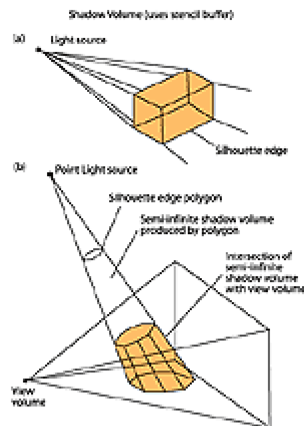
```
-- The color buffer and depth buffer are enabled for writing and depth testing is enabled.
-- Set attributes for drawing in shadow. Turn off the light source.
-- Render the entire scene.
-- Compute the polygons enclosing the shadow volume.
-- Disable the color and depth buffer for writing, but leave the depth test enabled.
-- Clear the stencil buffer to 0 if eye is outside the shadow volume, or 1 if inside.
-- Set the stencil function to always pass.
-- Set the stencil operations to increment if the depth test passes.
-- Turn on back face culling.
-- Render the shadow volume polygons.
-- Set the stencil operations to decrement if the depth test passes.
-- Turn on front face culling.
-- Render the shadow volume polygons.
-- Set the stencil function to test for equality to 0.
-- Set the stencil operations to do nothing.
```

For more on this, check out: reality.sgi.com/opengl/tips/

Here's another brief description of shadow volumes from another academic paper:

"For each shadow-generating polygon facing a light source, shadow planes are formed by each edge of the polygon, from the triangle of the edge vertices and the light source position (clipped by the view volume). Such shadow planes, termed shadow polygons by Crow, are added as invisible polygons to the scene data structure. During the scan-conversion process, the shadow polygons mark transitions into or out of shadows regions."

[Taken from "A Comparison of Three Shadow Volume Algorithms", Mel Slater, Dept. of CS, QMW, Univ. of London,]



click on image for full view

Anti-Aliasing--Quincunx, Multisampling and Grid Variants
As mentioned in the section on texture mapping, many image quality problems are sampling rate problems. That is, our 1024x768 display only has 786,432 pixels (samples) with which to display the 3D scene. And while that may seem like plenty of screen resolution to alleviate texture artifacts and edge "jaggies", it ain't.

These artifacts come under the heading of *aliasing*, which describes undesirable results that occur because of an insufficient sampling rate. And until the day comes that we're all happily running on 6000x4000 pixel displays (which would deliver 30 times as many pixels-as-samples on the screen as a 1024x768 display), the main way to alleviate these aliasing artifacts is through *antialiasing*. What this technique seeks to do is temporarily increase the scene resolution high above screen resolution, and then bring it back down to screen resolution.

The actual result of the most full-scene anti-aliasing techniques is that is that scene aliasing is masked with visual "noise," which can sometimes be seen as blurriness. New methods have improved this situation, though they still have tradeoffs, and we'll explore some of these new techniques here.

The topic of full-scene antialiasing (FSAA) generated much-spirited debate in the 3D world that past year or two. The late, great, 3dfx posited FSAA as the most notable feature of its VSA-100-based Voodoo 4 and 5 cards. And while many 3D aficionados gave 3dfx's approach to doing FSAA higher marks for visual quality, the Voodoo 5's fill rate couldn't always maintain playable frame rates when FSAA was turned on.

nVidia's approach to FSAA also ate fill-rate for lunch, and sometimes suffered compatibility problems with older 3D games that mixed 2D and 3D operations. 3dfx was correct to focus on FSAA as an important feature, but they were a bit ahead of the curve. But there's a new technique, called multisampling that nVidia's GeForce 3 supports, and other chips will likely follow suit.

Another Way to Get to Full-Scene Antialiasing

As is the way with so much in 3D, there's more than one way to get from here to there, and FSAA is no exception. Two methods that have been and continue to be used are called Ordered Grid Supersampling (OGSS) and Rotated Grid Supersampling (RGSS), with the former being used by nVidia and ATI, and the latter by 3dfx. Here's how they work:

OGSS takes the 3D scene, and upsamples it to a higher resolution, gathers a number of "sub-samples" (i.e. pixels that neighbor the original pixel (sample) at the higher resolution), performs an averaging operation of the pixel colors of those sub-samples, and that resulting average is written as the final pixel color. For the purposes of example, let's assume a screen resolution of 800x600, since those are nice round numbers. OGSS can be applied to different degrees, including:

- 2X: where the 3D scene is upsampled 2X only on one axis, say, to 800x1200, and only two sub-samples are used to calculate the final pixel color.
- 4X: where the 3D scene is upsampled 2X on both axes, say, to 1600x1200, and four sub-samples are used to calculate the final pixel color.

- 9X: where the 3D scene is upsampled 3X on both axes, say, to 2400x1800, and nine sub-samples are used to calculate the final pixel color.
- 16X: where the 3D scene is upsampled 4X on both axes, say, to 3200x2400, and sixteen sub-samples are used to calculate the final pixel color.

nVidia tweaks this approach somewhat by adding a Gaussian blur operation to one of their 4X OGSS settings, but the GeForce 2 suffered from often unacceptable frame rate hits with FSAA turned up to 4X, and also occasionally introduced a "melting screen" effect where scenes would distort severely, making games unplayable.

The nVidia driver control panel supports a 16X setting, but using it changes the frame rate unit of measurement from frames per second to seconds per frame. ATI supports two settings: 2X and 4X, and like nVidia, suffers a substantial performance hit running at 4X.

3dfx's approach, RGSS, is somewhat similar to OGSS, except that rather than upsample the scene to a higher resolution, this method instead makes two or four copies of the scene at screen resolution, then takes the scene's geometry and offsets, or "jitters" it slightly in two or four directions depending the number of copies made. The pixels from these two or four jittered copies are then averaged to produce the final pixel value. 3dfx's two settings are 2X and 4X, which dictates how many copies of the 3D scene gets created with its geometry averaged.

The Promise (and Semantics) of Multisampling

The problem with both methods is that they're heavy-handed, and consume huge tracts of graphics memory space and bandwidth. But DirectX 8 has introduced a slightly different approach called multisampling, which holds more promise as an antialiasing method that can be used without incurring as much of a performance hit.

Here we fumble upon yet another semantic football: what DirectX calls multisampling versus what hardware makers are calling multisampling are different animals. DirectX's multisampling exposes 3dfx's T-buffer in a general way. Recall that 3dfx's T-buffer is an accumulation buffer that allows motion blur, depth-of-field effects, and RGSS antialiasing.

nVidia's version of multisampling is a type of antialiasing that focuses its efforts more on edges than on textures. It operates similarly to supersampling in that it upsamples the original image, generating two samples per pixel, but instead of obtaining a separate color value for each sub-sample, it instead "reuses" sub-samples from neighboring pixels to attain higher quality but with less of a frame-buffer bandwidth hit.

Unlike super-sampling, multisampling uses the color from the original sample for all sub-samples, and relies on the sub-samples' positions to achieve its effect, which spares texture memory bandwidth. This method relies on the fact that visible object edges are typically marked by fairly abrupt color changes, and multisampling's visual effect is most palpable at these edges. One assumption that multisampling relies upon, is that high-quality texture filtering and the higher screen resolutions allowed by the newer chips' higher fill rates generally solves the problems of texture shimmering and blurriness. To date, only nVidia's GeForce 3 supports this type of multisampling, though ATI's next-generation R200 will likely have a similar feature as well.

The GeForce 3 chip also supports a mode of multisampling called Quincunx, which uses a five-sample per target pixel method, while only requiring the same video memory footprint as 2X OGSS. The five samples are actually "sub-samples" of existing pixels. The first two sub-samples are derived from the target pixel, using one sub-sample at pixel center, and the other in the upper-left corner of the pixel. The other three sub-samples are obtained from the upper-left corner of three neighboring pixels that exist beneath, to the right, and down a row and to the right of the target pixel.

An x-shaped filter pattern results, and this five-tap filter only hits memory twice instead of four times (like 4X OGSS). Beyond the sub-sample at pixel center, the offset and slightly different z-values of the other sub-samples is what achieves the edge antialiasing effect, while using what is essentially a blur function to achieve some texture antialiasing.

The Quincunx sampling pattern is actually a variation of the rotated grid super sampling (RGSS) technique that 3dfx made famous. nVidia claims that Quincunx AA delivers nearly equivalent quality to 4X OGSS,

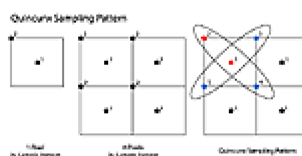
without the performance hit. Here are some benchmark numbers in frames/sec from recent testing we performed that shows Quincunx performance relative to 4X OGSS.

We can see in our results that this was NOT the case for Jane's USAF, and it actually performed nearly identical to 4X OGSS, but it was much better in Quake III.

	No FSAA (baseline)	Quincunx	4X OGSS	%slow-down from no FSAA to Quinxunx
Jane's USAF	40.6	32.6	33.5	20%
Quake 3	95	61.6	45.7	35%

Both games run at 1024x768x32 on 12.41 nVidia reference drivers on a Pentium 4 1.4GHz box with 128MB of RDRAM, a 64MB GeForce 3 reference board, and EIDE storage components.

To see the sub-sample pattern Quincunx uses, see this diagram:



click on image for full view

For more gory detail on antialiasing, check out: http://fullon3d.com/reports/fsaa/

Z-Buffering, or, My Z's Bigger Than Yours

Among the competing consumer 3D architectures--GeForce 2/3, ATI's Radeon and STMicro's Kyro II--there's a considerable variation in alternate z-buffering methods. These adhere to the 3D graphics mantra of "don't do what you don't have to", and all endeavor to avoid unneeded rendering work by doing z-buffering tests earlier before other tasks get performed, most notably texture mapping. Appreciate that z-buffering isn't the only method to determine visibility.

A fairly rudimentary z-buffering technique system is called the Painter's Algorithm (a back-to-front method), which begins from the back of the scene and draws everything in the scene, including full rendering of objects that might be occluded by objects nearer to the camera. In most cases, this algorithm achieves correct depth sorting, but it's inefficient, and can cause some drawing errors where triangles overlap one another. Further, it is costly in terms how many times a given pixel might be rendered.

Back in the days where 3D engines were run entirely on CPUs, the Painter's algorithm was used because a z-buffer wasn't readily available. But this algorithm is hardly used anymore as all consumer 3D graphics hardware now has z-buffering as a standard feature.

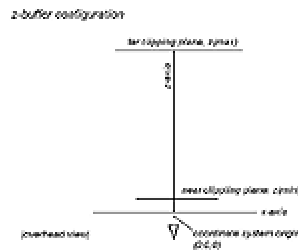
The multiple renders of a particular pixel in a scene is called overdraw. Averaged across the scene, this overdraw is often termed the scene's "depth complexity" and describes how many "layers" of objects you're looking at on average in a scene. Most games currently have a depth complexity of around 2.5 to 3.

Remember that although screen space is pretty much a "2D" mapping to the screen coordinates, say 1600x1200, the z-values have been carried through all of the previous operations in the pipeline.

But with all modern hardware having a z-buffer, this is now the preferred method of depth testing, as it is more efficient, and produces more correct results. To set the z-buffer setup for depth tests for the next frame of animation, (generally at the end of the previous scene being drawn) the z-buffer is cleared, meaning that the value of z_{max} gets written to all pixel positions in preparation for the next frame of animation's depth testing.

Here's another example where multiple methods exist to perform the same operation. An application can set up its z-buffer such that positive z-axis values go away from the view camera, or that negative z values

go away from the view camera. See the Z-Buffer diagram.



click on image for full view

For the sake of our example, let's set z_{min} at the near clipping plane, with positive z going away from the view camera. Irrespective of which way the z-buffer gets configured, depth testing is a pixel-by-pixel logical test that asks: "is there anything in front of this pixel?" If the answer is returned yes, that pixel gets discarded, if the answer is no, that pixel color gets written into the color buffer (back buffer) and the z-buffer's z-value at that pixel location is updated.

Another choice in how depth buffering gets done is "sort ordering", with the choices being front-to-back or back-to-front. But, z-buffering can do its depth tests either way, though back-to-front seems to be the preferred method. Sounds simple enough, right?

Well, another maxim of 3D graphics is that things are rarely as simple as they seem. Here are some the potential perils of z-buffering:

1. First, a 16-bit z-buffer has 65,535 different values (0 to 65,534), which one would think of as being plenty of accuracy. But there are two problems, one having to do with the scale of the 3D scene, and the other having to do with the non-linear behavior of values in the z-buffer. If an application is drawing a large, outdoor environment, say in a flight simulator, where the viewer may be looking at tens or hundreds of miles of terrain, a 16-bit z-buffer may not provide sufficient resolution for all objects in the scene.
2. Secondly, the z-values in screen space don't behave in a linear fashion, because they, like the x and y values back in clip space, were divided by w during the perspective divide. This non-linearity is a result of using a perspective projection (which creates the view frustum). The result is much greater accuracy or sensitivity for z-values in the near field, and then values become less accurate at points further away from z_{min}.

One way programmers can gain more consistent accuracy using the z-buffer is to set the near clipping plane further out, and bring the far clipping plane closer in, which brings the ratio of z-near to z-far closer to one, and evens out the variance in accuracy. [RTR, pp. 368-9] The trick of "compressing" the z-near and z-far planes is often used in conjunction with CAD programs, and is one way to even out the z-buffer's accuracy.

Another approach to solving the issue of uneven accuracy in depth values is to use w in lieu of z for depth buffering. Recall that the w is a scaled view-space depth value, and rather than write the z/w value for each vertex, the w value itself is instead used for depth tests. In some sense, w-buffering occurs whether it's being used for depth buffering or not, since a value of 1/w is used to interpolate pixels across triangle spans (the "space" between the two points where a triangle crosses a given scan-line) to produce perspective-correct texture mapping. [RTR, p. 369] But additionally, w can be used for depth buffering, and w values, which are floating-point values between 0 and 1, produce a linear scale from z-near to z-far, providing consistent accuracy.

According to Jim Blinn, in *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, (Morgan Kaufman, San Francisco, 1996), the decision to use w or z for depth buffering is best determined based on the ratio of z-near to z-far (z_{min} to z_{max}). His conclusions are:

- If z-near/z-far = 0.3, use w-buffering

- If z-near/z-far > 0.3, probably use z-buffering
- And if z-near/z-far is > 0.5, definitely use z-buffering [BLINN]

After the depth buffering tests have completed, and the front and back buffers get page-flipped, the z-buffer is "cleared", where the z_{max} value gets written into each pixel position in preparation for the next round of depth buffering tests.

One curious thing about the way traditional depth buffering has been done is the fact that is done so late in the pipeline, after texturing, fog and alpha operations have been processed. In other words, a good number of pixels will be thrown out at the last minute after having been textured, alpha-tested/blended and had fog applied. For this reason, ATI, nVidia and STMicro all have alternative approaches that seek to either move depth buffering further up the pipeline, or make depth buffering operations themselves faster.

ATI Radeon's Z-Buffering

With Radeon, ATI introduced the first "brute force" 3D chip that included alternative z-buffering methods, which broadly came under the heading of HyperZ. Its three principal features are Z Compression, Hierarchical Z, and Fast Z Clears. According to ATI, using HyperZ can yield about a 20% increase in frame rate, and frees up anywhere from 50-60% of frame buffer bandwidth by reducing the load introduced by depth-buffering operations.

Here's how hierarchical Z works: During scan-line conversion, the screen is divided into 8x8 pixel blocks (64 pixels), and each block is assigned a bit that determines whether that block is "not cleared" (visible) or "cleared" (occluded). A quick Z-check is performed on each of these 8x8 blocks to determine whether any pixels in them are visible. If a pixel in a given block is visible, the status bit is flipped to "not cleared", and the new z-value for that pixel is written to the Z-buffer.

If the block is not visible, its status bit remains set to "cleared", and the Z-buffer isn't touched, which saves bandwidth. In this second case, nothing is written into that block, and the values for the 64 values in that block don't matter, since they are assumed to be zero.

One by one, all blocks will come to be in the "not cleared" state as visible pixels are drawn. And finally, at the end of a scene, to prepare the Z-buffer for the next frame of animation, a "fast" Z-clear resets back to zero, or "cleared" only those status bits that had been flipped to one, or "not cleared, and the process repeats.

A traditional Z-clear actually has to write z-far values for all pixels in the z-buffer to prepare for the next set of depth tests, which impacts video memory bandwidth. But by only having to flip the status bit of a block of pixels to 0, much less data (1/64th as much to be precise) is written into the z-buffer, and it is ready for the next round of depth tests.

It turns out that not all games benefit equally from these HyperZ technologies. In particular, those game engines that have fast, efficient back-to-front sorting algorithms like Quake 3 receive less benefit from HyperZ, whereas games that don't z-sort in-engine, or don't do this especially well, can benefit considerably. As for Z compression, data in these 8x8 blocks can also be compressed, achieving anywhere from 2:1 to 4:1 compression ratios using a lossless compression algorithm, though ATI would not disclose specifics about how this compression algorithm works.

nVidia and Kyro II

nVidia
With its latest GeForce 3 chip, nVidia also introduced early z-checks and occlusion culling testing to try to discard unneeded vertices earlier to avoid unnecessary processing later in the pipe. Although nVidia declined to say exactly where they perform their early z-checks citing competitive reasons, the goal of this operation is to avoid unneeded work, and so the early z-check is likely occurring just before shading operations are carried out. nVidia also implemented occlusion culling in its driver to discard non-visible vertices early in the rendering process.

NVidia allows a programmer to add code to their 3D engine that lets the programmer do early occlusion culling tests, which can spare the hardware additional work further down in the pipeline. For more on occlusion culling, see the Application section. And finally, nVidia also implemented a 4:1 z-compression

scheme that reduces z-buffer memory traffic considerably.

Kyro II
Some of ATI's approaches to speeding up depth-buffering operations are similar to how depth buffering operations are handled on tiling renderers, such as STMicro's Kyro II, which made its debut on Hercules' Prophet 3D 4500. Tiling renderers, also known as deferred renderers, display list renderers, or chunking architectures, take a 3D scene and break it into component blocks, or tiles of pixels, usually (but not always) in a square pattern of say 32x32 pixels. Kyro II works on rectangular tiles of 32x16 pixels. By reducing amount of the screen information that the chip is working on at any given time, many rendering operations can be executed completely on-chip, sparing graphics memory bandwidth, and producing impressive performance, usually with lower chip transistor counts than traditional brute-force architectures.

On-chip operations can include full-scene antialiasing as well as depth "buffering" operations, though a tiling chip doesn't keep an actual z-buffer in graphics memory. One potential downside with tile-based renderers is that the display lists, or batches of polygons on which they operate, must be sorted in order to group polygons in a given region together. However, this sorting allows tiling architectures to do hidden surface removal and on-chip depth tests.

But the real downside is that the APIs are designed assuming immediate triangle rendering, and must be constrained in some way to work efficiently on these architectures. For example, reading back some portion of the frame buffer, z buffer, or stencil buffer during a frame is very inefficient. Mixing 2D and 3D operations in a frame is similarly inefficient. Most modern games don't do such things though, as they also introduce bottlenecks in highly pipelined traditional architectures.

Currently no tiling architecture on the market has accelerated T&L, though Kyro's upcoming STG5500 will offer accelerated T&L in addition to its four pixel pipes (see our news story here.) In addition, subsequent generations of tiling architectures will likely increase the number of transistors whose sole purpose in life will be to quickly perform polygon sorting, so that it doesn't bog down the rest of the chip.

Its Show Time

The final operations before the frame of animation is displayed have traditionally been dithering and blending. But dithering has become something of a vestige of the past as displays have moved to 16- and 32-bit color.

Dithering
In the days of 8- and even 4-bit color systems, programmers had a very limited set of colors to worth with, and often had to fudge colors using dithering. This technique tries to fool the eye into seeing more colors than are actually present by placing different colored pixels next to one another to create a composite color that the eye will see. An example would be checker-boarding blue and yellow pixels next to one another to try and give a surface a green appearance. Dither is another one of those "better than nothing" techniques whose visual effect isn't stellar, but is better than no effect at all.

Blending
Blending on the other hand, is alive and well, and is used extensively for texturing and translucency (called alpha blending). When a texture map is applied to a pixel to create a texel, the texture color is blended with the pixel's existing illumination color, and similarly, when an alpha (translucency) value is added to a pixel to make it translucent or transparent, that alpha value is blended with the existing pixel color to produce the alpha-blended pixel. These operations are typically adds or multiplies. Once blending operations have completed, this set of processed pixels that comprise the 3D scene is now ready to be page-flipped to being the front-buffer, and the previous frame of animation that was displayed now becomes the back-buffer that will be filled in with the next frame of animation.

Display
And finally, after this long and strange trip down the 3D graphics pipeline, what was a mass of vertices has been converted into the 3D scene we see, which, if we're moving along at 60fps, will be on screen for all of about 17 milliseconds. During its short time in the limelight, the next frame of animation is being transformed, lit, culled, clipped, projected, shaded, textured, fogged, alpha and depth tested, and ultimately, page-flipped onto the screen where it too will enjoy a fleeting 17ms of glory before yet another frame of animation will take its place.

When one stops to consider that not only can today's 3D hardware perform these tasks, but can run

through the process in tens of milliseconds, a tip of the propeller-hat is definitely in order to the hardware and software architects who create these wondrous 3D machines that live in our PCs. Euclid would be so proud…

Editor's Note:
The author wishes to thank the following for lending their assistance and considerable 3D kung fu on this article:

Jay Patel, Blizzard Entertainment
David Kirk, nVidia
Erik Lindholm, nVidia
Doug Rogers, nVidia
Derek Perez, nVidia
David Nalasco, ATI
Stuart Adler, CWRU
Dave Morey, eTesting Labs

Further Reading

These texts are excellent resources for more information about 3D pipelines, programming, and deeper explanations of the math that makes 3D graphics go.

Further Resources

Resource Title and Info	The Skinny
Real-Time Rendering Thomas Möller and Eric Haines, published by A.K. Peters, Ltd., ISBN# 1-56881-101-2	One of the very best resources for 3D programmers and even 3D enthusiasts who want to dig deeper on 3D. This book also does a good job of presenting most of the rudimentary math involved in 3D graphics. If you're going to buy one book about 3D, make it this one. Also check out their companion Web site, www.realtimerendering.com.
3D Computer Graphics, 3rd Edition Alan Watt, published by Addison-Wesley, ISBN# 0-201-63186-5	A very well written book, though it's specifically intended for programmers. Includes a CD-ROM with code examples and information about algorithms.
The Way Computer Graphics Works Olin Lathrop, published by Wiley Computer Publishing, ISBN# 0-471-13040-0	A well written, easy to understand book aimed at end-users looking to start down the path to 3D righteousness.
Computer Graphics: Principles and Practices, 2nd Edition Foley, Van Dam, Feiner & Hughes, published by Addison Wesley, ISBN: 0-201-84840-6	The granddaddy of academic graphics texts, this is the Rosetta Stone of 3D. This book is heavy on theory and math, and assumes a fairly solid background in the math required to program 3D. Not light reading.
3D Game Engine Design, David Eberly, published by Morgan Kaufman Publishers, ISBN: 1-55860-593-2	A good resource for programmers looking to write their own engine. Includes many examples written in C++, and a CD-ROM containing all source code used in the book, as well as source code for a game engine.
Jim Blinn's Corner : A Trip Down the Graphics Pipeline Jim Blinn, published by	A collection of writings compiled over the years from Jim Blinn's columns in the *IEEE Computer Graphics and Applications* journal. Aimed primarily at progammers, Blinn's unique take on 3D is still pretty

Morgan Kaufman Publishers, ISBN: 1558603875	accessible for the non-programming universe.

Online Resources

These links provide more detailed information about various facets of 3D graphics technology and programming. So check 'em out.

Gamasutra - Features

PC 3D Graphics Glossary of Terms

OpenGL-based Rendering Techniques

3D Graphics Glossary

DirectX Technical Articles

GameDev.net - Direct3D Immediate Mode

GameDev.net - Your Source for Game Development

Computer Graphics Links

Real-Time Rendering Resources

The Mesa 3D Graphics Library

The Animation Process

File-Format-Library 3d Graphic

First Steps To Animation

Direct3D Tutorials

Jim Blinn's Corner

Bookpool.com: *Jim Blinn's Corner: A Trip down the Graphics Pipeline*

Coordinate Representations

AntiAliasing Techniques

Sampling, aliasing, and antialiasing (part III)

Wildcat's SuperScene Anti-Aliasing

Matrices can be your Friends

Basic 3D Math

cs123 lectures

Gamasutra - Features - Occlusion Culling Algorithms [11.09.99]

Gamasutra - Features - Four Tricks for Fast Blurring in Software and Hardware [2.09.01]

SVG Requirements

IEEE Computer Graphics and Applications

Chris Hecker's Home Page

Sébastien Loisel

http://www.3dgamedev.com-resources-openglfaq.txt

Lighting Design Glossary

Matrix and Quaternion FAQ

Howstuffworks

3D Texture Making

The OpenGL Graphics Interface

Overviews of OpenGL

API Specs

GRAFICA Obscura

Computer Graphics

Introduction to DirectX 8.0

Paul Heckbert's Web Page

Procedural Texture Page

GameDev.net -- Direct3D 7 Immediate Mode Framework Programming 3 Multitexturing

Charles Poynton - Color technology

VRML Gamma

PowerVR Technologies

Gamasutra - Features - Implementing a 3D SIMD Geometry and Lighting Pipeline [04.16.99]

The Direct3D Transformation Pipeline

The good-looking textured light-sourced bouncy fun smart and stretchy page

Introduction to OpenGL

Evil Smokin' Dude's House O' 3D

Introduction to DirectX 8.0 again...

3D Graphics On-line Dictionary

Bibliography

Bracketed names are cited reference. Others are references used for general background information.

Arnaud, R., Jones, M., *Image Generator Visual Features Using Personal Computer Graphics Hardware*, presented at IMAGE 2000 Conference, July, 2000

[Blinn] Blinn, J., *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, Morgan Kaufman, San Francisco, 1996Blinn, J., W Pleasure, W Fun, IEEE Computer Graphics and Applications, May/June 1998

Dietrich, S., *Dot Product Texture Blending and Per-Pixel Lighting*, nVidia Technology Paper

[Eberly] Eberly, D., *3D Game Engine Design*, Morgan Kaufmann, San Francisco, 2000.

Foley, T., van Dam., A., Feiner, S., Hughes., J., *Computer Graphics: Principles and Practice, 2nd Edition*, Addison Wesley, New York, 1997

[Glassner] Glassner, A., *3D Computer Graphics, 2nd Edition*, Design Books, New York, 1989

[Gouraud] Gouraud, H., *Continuous Shading of Curved Surfaces*, IEEE Transactions on Computers, vol. C-20, June 1971

[Heckbert] Heckbert, P., *Survey of Texture Mapping*, IEEE Computer Graphics and Applications, Nov 1986

Kugler, A., *The Setup for Triangle Rasterization*, presented at the 11th Eurographics Workshop on Computer Graphics Hardware, August, 1996, Potiers, France

[Lathrop] Lathrop, O., *The Way Computer Graphics Works*, Wiley Computer Publishing, New York, 1997

Miller, T., *Hidden-Surfaces: Combining BSP Trees with Graph-Based Algorithms*, Brown University site, The Science and Technology Center for Computer Graphics and Scientific Visualizations

nVidia White Paper, *GeForce 3: Lightspeed Memory Architecture*, nVidia Corporation

nVidia White Paper, *Tranform and Lighting*, nVidia Corporation

[Phong] Phong, B., *Illumination for Computer Generated Pictures*, Communications of the ACM, vol. 18, no. 6, June 1975

PowerVR White Paper, *PowerVR Tile-based rendering*, April 2001

[RTR] Möller, T., Haines, E., *Real-Time Rendering*, A.K. Peters, Natick, MA, 1999

[Watt] Watt, A., *3D Computer Graphics: Second Edition*, Addison-Wesley, New York, 1993

Wright, R., Sweet, M., *OpenGL Superbible*, Waite Group Press, Corte Madera, CA, 1996

Copyright (c) 2004 Ziff Davis Media Inc. All Rights Reserved.