Game Engine Anatomy 101

April 12, 2002
By: Jake Simpson

We've come a very long way since the days of *Doom*. But that groundbreaking title wasn't just a great game, it also brought forth and popularized a new game-programming model: the game "engine." This modular, extensible and oh-so-tweakable design concept allowed gamers and programmers alike to hack into the game's core to create new games with new models, scenery, and sounds, or put a different twist on the existing game material. *CounterStrike, Team Fortress, TacOps, Strike Force*, and the wonderfully macabre *Quake Soccer* are among numerous new games created from existing game engines, with most using one of iD's *Quake* engines as their basis.

click on image for full view

TacOps and *Strike Force* both use the *Unreal Tournament* engine. In fact, the term "game engine" has come to be standard verbiage in gamers' conversations, but where does the engine end, and the game begin? And what exactly is going on behind the scenes to push all those pixels, play sounds, make monsters think and trigger game events? If you've ever pondered any of these questions, and want to know more about what makes games go, then you've come to the right place. What's in store is a deep, multi-part guided tour of the guts of game engines, with a particular focus on the Quake engines, since Raven Software (the company where I worked recently) has built several titles, *Soldier of Fortune* most notably, based on the *Quake* engine.

Start from the Start

So let's get started by first talking about the key differences between a game engine, and the game itself. Many people confuse the engine with the entire game. That would be like confusing an automobile engine with an entire car. You can take the engine out of the car, and build another shell around it, and use it again. Games are like that too. The engine can be defined as all the non-game specific technology. The game part would be all the content (models, animations, sounds, AI, and physics) which are called 'assets', and the code required specifically to make that game work, like the AI, or how the controls work.

For those that have ever glanced at *Quake's* game structure, the engine would be the Quake.exe, and the game side would be the QAGame.dll and CGame.dll. If you don't know what that means, don't worry; I didn't either till someone explained it to me. But you'll know exactly what it means and a whole lot more before were through. This game engine tutorial will run for eleven parts. Yes, count'em, eleven parts! We're doling them out in bite-sized chunks, probably around 3000 words each. So it's time to commence with Part I of our exploration into the innards of the games we play, where we'll review some of the basics, setting the stage for subsequent chapters …

The Renderer

Let's begin our discussion of game engine design with the renderer, and we'll speak from the perspective of a game developer (my background). In fact, throughout all segments of this tutorial, we'll often be speaking from the game developer perspective to get you thinking like we're thinking!

click on image for full view

So what exactly is the renderer and why is it important? Well, without it you don't get to see anything. It visualizes the scene for the player / viewer so he or she can make appropriate decisions based upon what's displayed. While some of our upcoming discussion may seem a bit daunting to the novice, hang in there. We'll explain key aspects of what a renderer does, and why it's so necessary.

The renderer is generally the first thing you tend to build when constructing an engine. But without seeing anything -- how do you know your code is working? The renderer is where over 50% of the CPU's processing time is spent, and where game developers will often be judged the most harshly. If we get it wrong, we can screw things up so badly our programming skills, our game, and our company can become the industry joke in 10 days flat. It's also where we are most dependent on outside vendors and forces, and the area where they have to deal with the widest scope of potential operating targets. When put like that, it really doesn't sound all that attractive to be building one of these renderers (but it is), but without a good renderer, the game would probably never make the Top 10.

The business of getting pixels on screen these days involves 3D accelerator cards, API's, three-dimensional math, an understanding of how 3D hardware works, and a dash of magic dust. For consoles, the same kind of knowledge is required, but at least with consoles you aren't trying to hit a moving target. A console's hardware configuration is a frozen "snapshot in time", and unlike the PC, it doesn't change at all over the lifetime of the console.

In a general sense, the renderer's job is to create the visual flare that will make a game stand apart from the herd, and actually pulling this off requires a tremendous amount of ingenuity. 3D graphics is essentially the art of the creating the most while doing the least, since additional 3D processing is often expensive both in terms of processor cycles and memory bandwidth. It's also a matter of budgeting, figuring out where you want to spend cycles, and where you're willing to cut corners in order to achieve the best overall effect. What we'll cover next are the tools of the trade, and how they get put to good use to make game engines work.

Creating the 3D World

Recently I had a conversation with someone who has been in the computer graphics biz for years, and she confided with me that the first time she saw a 3D computer image being manipulated in real time she had no idea how it was done, and how the computer was able to store a 3D image. This is likely true for the average person on the street today, even if they play PC, console, or arcade games frequently. We'll discuss some of the details of creating a 3D world from a game designers perspective below, but you should also read Dave Salvator's three-part 3D Pipeline Tutorial for a structured overview of all the main processes involved in generating a 3D image.

http://www.tnt.uni-hannover.de/js/project/3dmod/multview/wireframe.html

3D objects are stored as points in the 3D world (called vertices), with a relation to each other, so that the computer knows to draw lines or filled surfaces between these points in the world. So a box would have 8 points, one for each of the corners. There are 6 surfaces for the box, one for each of the sides it would have. This is pretty much the basis of how 3D objects are stored. When you start getting down to some of the more complicated 3D stuff, like a *Quake* level for example, you are talking about thousands of vertices (sometimes hundreds of thousands), and thousands of polygonal surfaces. See the above graphic for a wireframe representation. Essentially though, it relates to the box example above, only with lots and lots of small polygons to make up complicated scenes.

How models and worlds are stored is a part of the function of the renderer, more than it is part of the application / game. The game logic doesn't need to know how objects are represented in memory, or how the renderer is going to go about displaying them. The game simply needs to know that the renderer is going to represent objects using the correct view, and displaying the correct models in their correct frames of animation.

In a good engine, it should be possible to completely replace the renderer with a new one, and not touch a line of game code. Many cross-platform engines, such as the *Unreal* engine, and many homegrown console engines do just that—for example, the renderer model for the GameCube version of the game can be replaced, and off you go.

Back to internal representation-- there's more than one way to represent points in space in computer memory beyond using a coordinate system. You can do it mathematically, using an equation to describe straight or curved lines, and derive polygons, which pretty much all 3D cards use as their final rendering primitive. A primitive is the lowest rendering unit you can use on any card, which for almost all hardware now is a three-point polygon (triangle). The newer nVidia and ATI cards do allow you render mathematically (called higher-order surfaces), but since this isn't standard across all graphics cards, you can't depend on it as a rendering strategy just yet. This is usually somewhat expensive from a processing perspective, but it's often the basis for new and experimental technologies, such as terrain rendering or making hard-edged objects have softer edges. We'll define these higher-order surfaces a little more in the patches section below.

Culling Overview

Here's the problem. I have a world described in several hundred thousand vertices / polygons. I have a first person view that's located on one side of our 3D world. In this view are some of the world's polygons, though others are not visible, because some object or objects, like a visible wall, is obscuring them. Even the best game coders can't handle 300,000 triangles in the view on a current 3D card and still maintain 60fps (a key goal). The cards simply can't handle it, so we have to do some coding to remove those polygons that aren't visible before handing them to the card. The process is called culling.

click on image for full view

If you don't see it, it isn't there. By culling the non-visible parts of a 3D world, a game engine can reduce its workload considerably. Look at this scene and imagine that there's a room behind the one under construction, but if it's not visible from this vantage point, the other room's geometry and other 3D data can be discarded.

There are many different approaches to culling. Before we get into that however, let's discuss *why* the card can't handle super-high polygon counts. I mean, doesn't the latest card handle X million polygons per second? Shouldn't it be able to handle anything? First, you have to understand that there are such things as *marketing* polygon rates, and then *real world* polygon rates. Marketing polygon rates are the rates the card can achieve theoretically.

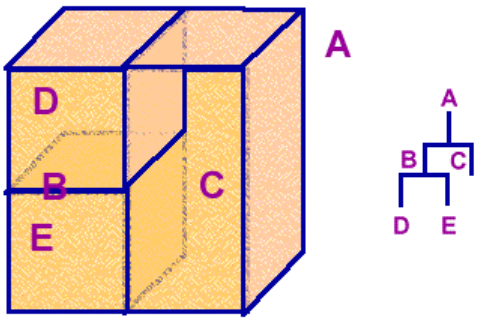
How many polygons can it handle if they are all on screen, the same texture, and the same size, without the application that's throwing polygons at the card doing anything *except* throwing polygons at the card. Those are numbers the graphics chip vendors throw at you. However, in real gaming situations the application is often doing lots of other things in the background -- doing the 3D transforms for the polygons, lighting them, moving more textures to the card memory, and so on. Not only do textures get sent to the card, but the details for each polygon too. Some of the newer cards allow you to actually store the model / world geometry details within the card memory itself, but this can be costly in terms of eating up space that textures would normally use, plus you'd better be sure you are using those model vertexes every frame, or you are just wasting space on the card. But we're digressing here. The key point is that what you read on the side of the box isn't necessarily what you would get when actually using the card, and this is especially true if you have a slow CPU, or insufficient memory.

Basic Culling Methods

The simplest approach to culling is to divide the world up into sections, with each section having a list of other sections that can be seen. That way you only display what's possible to be seen from any given point. How you create the list of possible view sections is the tricky bit. Again, there are many ways to do this, using BSP trees, Portals and so on.

I'm sure you've heard the term BSP used when talking about *Doom* or *Quake*. It stands for Binary Space Partitioning. This is a way of dividing up the world into small sections, and organizing the world polygons such that it's easy to determine what's visible and what's not -- handy for software based renderers that don't want to be doing too much overdrawing. It also has the effect of telling you where you are in the world in a very efficient fashion.

BSP Tree Hierarchy



Source:

http://www.martinb.com/threed/solidmodel/spatialdecomposition/bsp.htm

A Portal based engine (first really brought to the gaming world by the defunct project Prey from 3D

Realms) is one where each area (or room) is built as its own model, with doors (or portals) in each section that can view another section. The renderer renders each section individually as separate scenes. At least that's the theory. Suffice to say this is a required part of any renderer and is more often than not of great importance. Some of these techniques fall under the heading of "occlusion culling", but all of them have the same intent: eliminate unnecessary work early.

For an FPS (first-person shooter game) where there are often a lot of triangles in view, and the player assumes control of the view, it's imperative that the triangles that can't be seen be discarded, or culled. The same holds true for space simulations, where you can see for a long, long way -- culling out stuff beyond the visual range is very important. For games where the view is controlled -- like an RTS (real-time strategy game)-- this is usually a lot easier to implement. Often this part of the renderer is still in software, and not handed off to the card, but it's pretty much only a matter of time before the card will do it for you.

Basic Graphics Pipeline Flow

A simple example of a graphics pipeline from game to rendered polygons might flow something like this:

- Game determines what objects are in the game, what models they have, what textures they use, what animation frame they might be on, and where they are located in the game world. The game also determines where the camera is located and the direction it's pointed.
- Game passes this information to the renderer. In the case of models, the renderer might first look at the size of the model, and where the camera is located, and then determine if the model is onscreen at all, or to the left of the observer (camera view), behind the observer, or so far in the distance it wouldn't be visible. It might even use some form of world determination to work out if the model is visible (see next item).
- The world visualization system determines where in the world the camera is located, and what sections / polygons of the world are visible from the camera viewpoint. This can be done many ways, from a brute force method of splitting the world up into sections and having a straight "I can see sections AB&C from section D" for each part, to the more elegant BSP (binary space partitioned) worlds. All the polygons that pass these culling tests get passed to the polygon renderer.
- For each polygon that is passed into the renderer, the renderer transforms the polygon according to both local math (i.e. is the model animating) and world math (where is it in relation to the camera?), and then examines the polygon to determine if it is back-faced (i.e. facing away from the camera) or not. Those that are back-faced are discarded. Those that are not are lit, according to whatever lights the renderer finds in the vicinity. The renderer then looks at what texture(s) this polygon uses and ensures the API/graphics card is using that texture as its rendering base. At this point the polygons are fed off to the rendering API and then onto the card.

Obviously this is very simplistic, but you get the idea. The following chart is excerpted from Dave Salvator's 3D pipeline story, and gives you some more specifics:

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

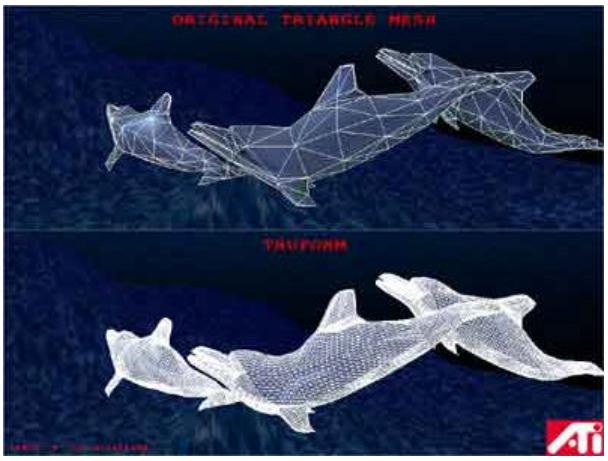
Usually you would feed all the polygons into some sort of list, and then sort this list according to texture (so you only feed the texture to the card once, rather than per polygon), and so on. It used to be that polygons would be sorted using distance from the camera, and those farthest away rendered first, but these days, with the advent of Z buffering that is less important. Except of course, for those polygons that have transparency in them. These have to be rendered after all the non translucent polygons are done, so that what's behind them can show up correctly in the scene. Of course, given that, you'd have to render these polygons back-to-front as a matter of course. But often in any given FPS scene there generally aren't too many of transparent polys. It might look like there are, but actually in comparison to those polygons that don't have alpha in them, it's a pretty low percentage.

Once the application hands the scene off to the API, the API in turn can take advantage of hardware-accelerated transform and lighting (T&L), which is now commonplace in 3D cards. Without going into an explanation of the matrix math involved (see Dave's 3D Pipeline Tutorial), transforms allow the 3D card to render the polygons of whatever you are trying to draw at the correct angle and at the correct place in the world relative to where your camera happens to be pointing at any given moment.

There are a lot of calculations done for each point, or vertex, including clipping operations to determine if any given polygon is actually viewable, due to it being off screen or partially on screen. Lighting operations work out how bright textures' colors need to be, depending on how light in the world falls on this vertex, and from what angle. In the past, the CPU handled these calculations, but now current-generation graphics hardware can do it for you, which means your CPU can go off and do other stuff. Obviously this is a Good Thing(tm), but since you can't depend on all 3D cards out there having T&L on board, you will have to write all these routines yourself anyway (again speaking from a game developer perspective). You'll see the "Good Thing(tm)" phrase throughout various segments of this story. These are features I think make very useful contributions to making games look better. Not surprisingly, you'll also see its opposite; you guessed it, a Bad Thing(tm) as well. I'm getting these phrases copyrighted, but for a small fee you can still use them too.

Patches (Higher-Order Surfaces)

In addition to triangles, patches are now becoming more commonly used. Patches (another name for higher-order surfaces) are very cool because they can describe geometry (usually geometry that involves some kind of curve) with a mathematical expression rather than just listing out gobs of polygons and their positions in the gaming world. This way you can actually build (and deform) a mesh of polygons from the equation on the fly, and then decide how many polygons you actually want to see from the patch. So you could describe a pipe for instance, then have many examples of this pipe in the world. In some rooms, where you are already displaying 10,000 polygons you can say, "OK, this pipe should only have 100 polygons in it, because we are already displaying lots and lots of polygons, and any more would slow down the frame rate". But in another room, where there are only 5,000 polygons in view, you can say, "Now, this pipe can have 500 polygons in it, because we aren't approaching our polygon display budget for this frame". Very cool stuff -- but then you do have to decode all this in the first place and build the meshes, and that's not trivial. But there's a real cost savings of sending a patch's equation across AGP versus sending boatloads of vertices to describe the same object. SOF2 uses a variation in this approach to build its terrain system.



In fact ATI now has TruForm, which can take a triangle-based model, and convert that model to one based on higher-order surfaces to smooth them out -- and then convert it back to a higher triangle-count model (called retesselation) with as many as ten times as many triangles before. The model then gets sent down the pipeline for further processing. ATI actually added a stage just before their T&L engine to handle this processing. The drawback here is controlling what gets smoothed and what doesn't. Often some edges you want to leave hard, like noses for instance may get smoothed inappropriately. Still, it's a clever technology, and I can see it getting used more in the future.

That's it for Part I -- we'll be continuing our introductory material in Part II by lighting up and texturing the world, and jumping into more depth in subsequent segments.

Game Engine Anatomy 101

April 15, 2002
By: Jake Simpson

Light of the World

During the transform process, usually in a coordinate space that's known as view space, we get to one of the most crucial operations: lighting. It's one of those things that when it works, you don't notice it, but when it doesn't, you notice it all too much. There are various approaches to lighting, ranging from simply figuring out how a polygon is oriented toward a light, and adding a percentage of the light's color based on orientation and distance to the polygon, all the way to generating smooth-edged lighting maps to overlay on basic textures. And some APIs will actually offer pre-built lighting approaches. For example, OpenGL offers per polygon, per vertex, and per pixel lighting.

click on image for full view

In vertex lighting, you determine how many polygons are touching one vertex and then take the mean of all the resultant polygons orientations (called a normal) and assign that normal to the vertex. Each vertex for a given polygon will point in slightly different directions, so you wind up gradating or interpolating light colors across a polygon, in order to get smoother lighting. You don't necessarily see each individual polygon with this lighting approach. The advantage of this approach is that hardware can often help do this in a faster manner using hardware transform and lighting (T&L). The drawback is that it doesn't produce shadowing. For instance, both arms on a model will be lit the same way, even if the light is on the right side of the model, and the left arm should be left in shadow cast by the body.

These simple approaches use shading to achieve their aims. For flat polygon lighting when rendering a polygon, you ask the rendering engine to tint the polygon to a given color all over. This is called flat shading lighting (each polygon reflects a specific light value across the entire polygon, giving a very flat effect in the rendering, not to mention showing exactly where the edges of each polygon exist).

For vertex shading (called Gouraud shading) you ask the rendering engine to tint each vertex provided with a specific color. Each of these vertex colors is then taken into account when rendering each pixel depending on its distance from each vertex based on interpolating. (This is actually what Quake III uses on its models, to surprisingly good effect).

Then there's Phong shading. Like Gouraud shading, this works across the texture, but rather than just using interpolation from each vertex to determine each pixels color, it does the same work for each pixel that would be done for each vertex. For Gouraud shading, you need to know what lights fall on each vertex. For Phong, you do this for each pixel.

Not surprisingly, Phong Shading gives much smoother effects, but is far more costly in rendering time, since each pixel requires lighting calculations. The flat shading method is fast, but crude. Phong shading is more computationally expensive than Gouraud shading, but gives the best results of all, allowing effects like specularity ("shiny-ness").These are just some of the tradeoffs you must deal with in game development.

In a Different Light

Next up is light map generation, where you use a second texture map (the light map) and blend it with the existing texture to create the lighting effect. This works quite well, but is essentially a canned effect that's pre-generated before rendering. But if you have dynamic lights (i.e. lights that move, or get turned on and off with no program intervention) then you will have to regenerate the light maps every frame, modifying them according to how your dynamic lights may have moved. Light maps can render quickly, but they are very expensive in terms of memory required to store all these textures. You can use some compression tricks to make them take less memory space, or reduce their size, even make them monochromatic (though if you do that, you don't get colored lights), and so on. But if you do have multiple dynamic lights in the scene, regenerating light maps could end up being expensive in terms of CPU cycles.

Usually there's some kind of hybrid lighting approach used in many games. *Quake III* for instance, uses light maps for the world, and vertex lighting for the animating models. Pre-processed lights don't affect the animated models correctly--they take their overall light value for the whole model from the polygon they are standing on--and dynamic lights will be applied to give the right effect. Using a hybrid lighting approach is a tradeoff that most people don't notice, but it usually gives an effect that looks "right". That's what games are all about--going as far as necessary to make the effect look "right", but not necessarily correct.

Of course all that goes out the window for the new *Doom* engine, but then that's going to require a 1GHz CPU and a GeForce 2 at the very least to get all the effects. Progress it is, but it does all come at a price.

Once the scene has been transformed and lit, we move on to clipping operations. Without getting into gory detail, clipping operations determine which triangles are completely inside the scene (called the view frustum) or are partially inside the scene. Those triangles completely inside the scene are said to be trivially accepted, and they can be processed. For a given triangle that is partially inside the scene, the portion outside the frustum will need to be clipped off, and the remaining polygon inside the frustum will need to be retesselated so that it fits completely inside the visible scene. (see our 3D Pipeline Tutorial for more details).

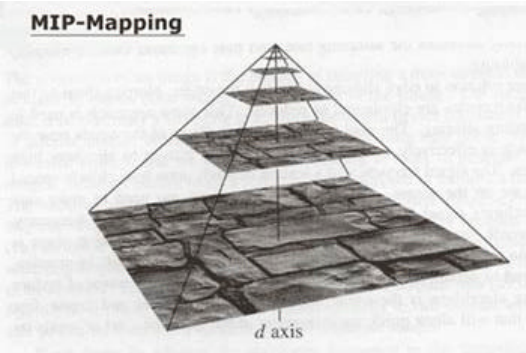
Once the scene has been clipped, the next stage in the pipeline is the triangle setup phase (also called scan-line conversion) where the scene is mapped to 2D screen coordinates. At this point we get into rendering operations.

Textures and MIP Mapping

Textures are hugely important to making 3D scenes look real, and are basically little pictures that you break up into polygons and apply to an object or area in a scene. Multiple textures can take up a lot of memory, and it helps to manage their size with various techniques. Texture compression is one way of making texture data smaller, while retaining the picture information. Compressed textures take up less space on the game CD, and more importantly, in memory and on your 3D card. Another upside is that when you ask the card to display the texture for the first time, the compressed (smaller) version is sent from the PC main memory across the AGP interconnect to the 3D card, making everything that little bit faster. Texture compression is a Good Thing. We'll discuss more about texture compression below.

MIP Mapping

Another technique used by game engines to reduce the memory footprint and bandwidth demands of textures is to use MIP maps. The technique of MIP mapping involves preprocessing a texture to create multiple copies, where each successive copy is one-half the size of the prior copy. Why would you do this? To answer that, you need to understand how 3D cards display a texture. In the worst case you take a texture, stick it on a polygon, and just whack it out to the screen. Let's say there's a 1:1 relationship, so one texel (texture element) in the original texture map corresponds to one pixel on a polygon associated with the object being textured. If the polygon you are displaying is scaled down to half size, then effectively the texture is displaying every other texel. Now this is usually OK -- but can lead to some visual weirdness in some cases. Let's take the idea of a brick wall. Say the original texture is a brick wall, with lots of bricks, but the mortar between them is only one pixel wide. If you scale the polygon down to half-size, and if only every other texel is applied, all of sudden all your mortar vanishes, since it's being scaled out. It just gives you weird images.



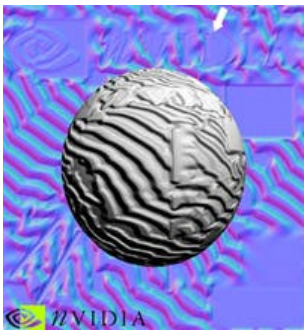
With MIP mapping, you scale the image yourself, before the card gets at it, and since you can pre-process it, you do a better job of it, so the mortar isn't just scaled out. When the 3D card draws the polygon with the texture on it, it detects the scale factor and says, "you know, instead of just scaling the largest texture, I'll use the smaller one, and it will look better." There. MIP mapping for all, and all for MIP mapping.

Multiple Textures and Bump Mapping

Single texture maps make a large difference in overall 3D graphics realism, but using multiple textures can achieve even more impressive effects. This used to require multiple rendering passes that ate fill rate for lunch. But with multi-piped 3D accelerators like ATI's Radeon and nVidia's GeForce 2 and above, multiple textures can often be applied in a single rendering pass. When generating multitexture effects, you draw one polygon with one texture on it, then render another one right over the top with another texture, but with some transparency to it. This allows you to have textures appearing to move, or pulse, or even to have shadows (as we described in the lighting section). Just draw the first texture, then draw a texture that is all black but has a transparency layer over the top of it, and voila -- instant shadowing. This technique is called light mapping (or sometimes-dark mapping), and up until the new *Doom* has been the traditional way that levels are lit in Id engines.

Bump mapping is an old technology that has recently come to the fore. Matrox was the first to really promote various forms of bump mapping in popular 3D gaming a few years ago. It's all about creating a texture that shows the way light falls on a surface, to show bumps or crevices in that surface. Bump mapping doesn't move with lights-- it's designed to be used for creating small imperfections on a surface, not for large bumps. For instance you could use bump mapping to create seeming randomness to a terrain's detail in a flight simulator, rather than use the same texture repeatedly, which doesn't look very

interesting.



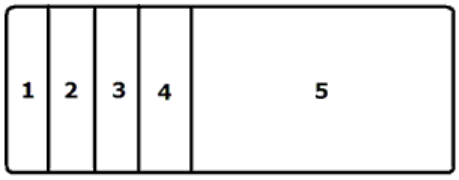
click on image for full view

Bump mapping creates a good deal more apparent surface detail, although there's a certain amount of sleight of hand going on here, since by strict definition it doesn't change relative to your viewing angle. Given the per-pixel operations that the newer ATI and nVidia cards can perform, this default viewing angle drawback isn't really a hard and fast rule anymore. Either way, it hasn't been used much by game developers since until recently; more games can and should use bump mapping.

Cache Thrash = Bad Thing

Texture cache management is vital to making game engines go fast. Like any cache, hits are good, and misses are bad. If you get into a situation where you've got textures being swapped in and out of your graphics card's memory, you've got yourself a case of texture cache thrashing. Often APIs will dump every texture when this happens, resulting in every one of them having to be reloaded next frame, and that's time consuming and wasteful. To the gamer, this will cause frame rate stutters as the API reloads the texture cache.

Typical Frame-Buffer Memory Allocation



1. **Front-Buffer (3MB)**
2. **First Back-Buffer (3MB)**
3. **Second Back-Buffer (3MB)**
4. **Z-Buffer (3MB)**
5. **Texture Cache (20MB)**

Based on running triple-buffered at a display resolution of 1024x768x32 on a 32MB frame buffer. Some memory may be allocated for driver code. But after allocating for color and Z-buffers, there's 20MB left over for texture caching.

There are various techniques for keeping texture cache thrashing to a minimum, and fall under the rubric of texture cache management-- a crucial element of making any 3D game engine go fast. Texture management is a Good Thing--what that means is only asking the card to use a texture once, rather than asking it to use it repeatedly. It sounds contradictory, but in effect it means saying to the card, "look, I have all these polygons and they all use this one texture, can we just upload this once instead of many times?" This stops the API (or software behind the graphics drivers) from uploading the one texture to the card more than once. An API like OpenGL actually usually handles texture caching and means that the API handles what textures are stored on the card, and what's left in main memory, based on rules like how often the texture is accessed. The real issue comes here in that a) you don't often know the exact rules the API is using and b) often you ask to draw more textures in a frame than there is space in the card to hold them.

Another texture cache management technique is texture compression, as we discussed a bit earlier. Textures can be compressed much like wave files are compressed to MP3 files, although with nowhere near the compression ratio. Wave to MP3 compression yields about an 11:1 compression ratio, whereas most texture compression algorithms supported in hardware are more like 4:1, but even that can make a huge difference. In addition, the hardware decompressed textures only as it needs them on the fly *as it is rendering*. This is pretty cool, but we've only just scratched the surface of what's possible there in the future.

As mentioned, another technique is ensuring that the renderer only asks the card to render one texture once. Ensure that all the polygons that you want the card to render using the same texture get sent across at once, rather than doing one model here, another model there, and then coming back to the original texture again. Just do it once, and you only transfer it across the AGP interconnect once. *Quake III* does this with its shader system. As it processes polygons it adds them to an internal shader list, and once all the polygons have been processed, the renderer goes through the texture list sending across the textures and all the polygons that use them in one shot.

The above process does tend to work against using hardware T&L on the card (if it is present) efficiently. What you end up with are large numbers of small groups of polygons that use the same texture all over the screen, all using different transformation matrices. This means more time spent setting up the hardware T&L engine on the card, and more time wasted. It works OK for actual onscreen models, because they tend to use a uniform texture over the whole model anyway. But it does often play hell with the world rendering, because many polygons tend to use the same wall texture. It's usually not that big of a deal because by and large, textures for the world don't tend to be that big, so your texture caching system in the API will handle this for you, and keep the texture around on the card ready for use again.

On a console there usually isn't a texture caching system (unless you write one). In the case of the PS2 you'd be better off going with the "'texture once" approach. On the Xbox it's immaterial since there is no graphics memory per se (it's a UMA architecture), and all the textures stay in main memory all the time anyway.

Trying to whack too many textures across the AGP interconnect is, in actual fact, the second most common bottleneck in modern PC FPS games today. The biggest bottleneck is the actual geometry processing that is required to make stuff appear where it's supposed to appear. The math involved in generating the correct world positions for each vertex in models is by far the most time consuming thing that 3D FPSes do these days. Closely followed by shoving large numbers of textures across the AGP interconnect if you don't keep your scene texture budget under control. You do have the capability to affect this, however. By dropping your top MIP level (remember that--where the system constantly subdivides your textures for you?), you can halve the size of the textures the system is trying to push off to the card. Your visual quality goes down--especially noticeable in cinematic sequences--but your frame rate goes up. This approach is especially helpful for online games. Both *Soldier of Fortune II* and *Jedi Knight II: Outcast* are actually designed with cards in mind that aren't really prevalent in the marketplace yet. In order to view the textures at their maximum size, you would need a minimum of 128MB on your 3D card. Both products are being designed with the future in mind.

And that wraps Part II. In the next segment, we'll be introducing many topics, including memory management, fog effects, depth testing, anti-aliasing, vertex shaders, APIs, and more.

Game Engine Anatomy 101

April 19, 2002
By: Jake Simpson

Musings on Memory Usage

Let's consider how 3D card memory is actually used today and how it will be used in the future. Most 3D cards these days handle 32-bit color, which is 8 bits for red, 8 for blue, 8 for green, and 8 for transparency of any given pixel. That's 256 shades of red, blue, and green in combination, which allows for 16.7 million colors-- that's pretty much all the colors you and I are going to be able to see on a monitor.

So why is game design guru John Carmack calling for 64-bit color resolution? If we can't see the difference, what's the point? The point is this: let's say we have a point on a model where several lights are falling, all of different colors. We take the original color of the model and then apply one light to it, which changes the color value. Then we apply another light, which changes it further. The problem here is that with only 8 bits to play with, after applying 4 lights, the 8 bits just aren't enough to give us a good resolution and representation of the final color. The lack of resolution is caused by quantization errors, which are essentially rounding errors resulting from an insufficient number of bits. You can very quickly run out of bits, and as such, all the colors tend to get washed out. With 16 or 32 bits per color, you have a much higher resolution, so you can apply tint after tint to properly represent the final color. Such color-depths can quickly consume much storage space.

We should also mention the whole card memory vs. texture memory thing. What's going on here is that each 3D card really only has a finite amount of memory on board to stuff the front and back buffers, the z-buffer, plus all the wonderful textures. With the Original Voodoo1 card, it was 2MB, then came the Riva TNT, which upped it to16MB. Then the GeForce and ATI Rage gave you 32MB, now some versions of the GeForce 2 through 4 and Radeons come with 64MB to 128MB. Why is this important? Well, let's crunch some numbers...

Let's say you want to run your game using a 32-bit screen at 1280x1024 with a 32-bit Z-buffer because you want it to look the best it can. OK, that's 4 bytes per pixel for the screen, plus 4 bytes per pixel for the z-buffer, since both are 32 bits wide per pixel. So we have 1280x1024 pixels -- that's 1,310,720 pixels. Multiply that by 8 based on the number of bytes for the front buffer and the Z-buffer, and you get 10,485,760 bytes. Include a back buffer, and you have 1280x1024x12, which is 15,728,640 bytes, or 15MB. On a 16MB card, that would leave us with just 1MB to store all the textures. Now if the original textures are true 32 bits or 4 bytes wide, as most stuff is these days, then we can store 1MB / 4 bytes per pixel = 262,144 pixels of textures per frame on the card itself. That's about 4 256x256 texture pages.

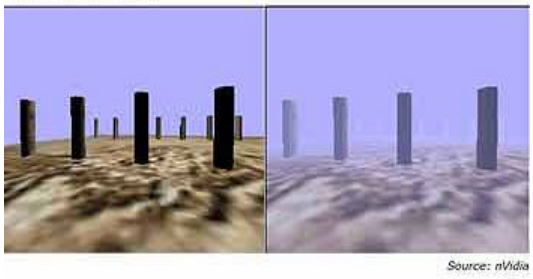
Clearly, the above example shows that the older 16MB frame buffers have nowhere near enough memory for what a modern game requires to draw its prettiness these days. We'd obviously have to reload textures per frame to the card while it's drawing. That's actually what the AGP bus was designed to do, but still, AGP is slower than a 3D card's frame buffer, so you'dincur a sizeable performance hit. Obviously if you drop the textures down to 16-bit instead of 32-bit, you could push twice as many lower resolutions across AGP. Also, if you ran at a lower color resolution per pixel, then more memory is available on the card for keeping often used textures around (called caching textures). But you can never actually predict how users will set up their system. If they have a card that runs at high resolutions and color depths, then chances are they'll set their cards that way.

Let's Get Down and Get Foggy

Now we come to fog, since it is a visual effect of sorts. Most engines these days can handle this, as it

comes in mighty handy for fading out the world in the distance, so you don't see models and scene geography popping on in the distance as they come into visual range crossing the far clipping plane. There's also a technique called volumetric fogging. For the uninitiated, this is where fog isn't a product of distance from the camera, but is an actual physical object that you can see, travel through, and pass out the other side-- with the visual fog levels changing as you move through the object. Think of traveling through a cloud -- that's a perfect example of volumetric fogging. A couple of good examples of implementation of volumetric fogging are *Quake III's* red mist on some of their levels, or the new Lucas Arts GameCube version of *Rogue Squadron II*. That has some of the best-looking clouds I've ever seen -- about as real as you can get.

Fog Rendering



Source: nVidia

click on image for full view

While we are talking about fogging, it might be a good time to briefly mention alpha testing and alpha blending for textures. When the renderer goes to put a specific pixel on the screen, assuming it's passed the Z-buffer test (defined below), we might end up doing some alpha testing. We may discover that the pixel needs to be rendered transparently to show some of what's behind it. Meaning that we have to retrieve the pixel that's already there, mix in our new pixel and put the resulting blended pixel back in the same location. This is called a read-modify-write operation, and it's far more time-consuming than an ordinary pixel write.

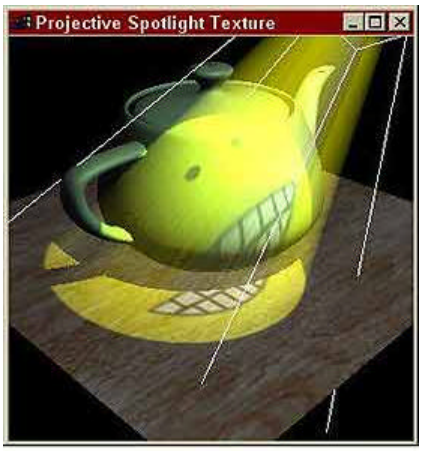
There are different types of mixing (or blending) that you can do, and these different effects are called blend modes. Straight Alpha blending simply adds a percentage of the background pixel to an inverse percentage of the new pixel. Then there's additive blending, which takes a percentage of the old pixel, and just adds a specific amount of the new pixel (rather than a percentage). This gives much brighter effects (Kyle's Lightsaber effect in Jedi Knight II does this, to give the bright core).

With each new card we see from vendors, we get newer and more complex blending modes made available to us in hardware to do more and crazier effects. And of course with the per pixel operations available in the GF3+4 and latest Radeon boards, the sky is the limit.

Stencil Shadowing and Depth Testing

With stencil shadowing, things gets complicated and expensive. Without going into too many gory details right now (this could be an article all by itself), the idea is to render a view of a model from the light source's perspective and then use this to create or cast a polygon with this texture's shape onto the affected receptor surfaces. You are actually casting a light volume which will 'fall' on other polygons in the view. You end up with a real-looking lighting, that even has perspective built into it. But it's expensive, because you're creating textures on the fly, and doing multiple renders of the same scene.

You can create shadows a multitude of different ways, and as is often the case, the rendering quality is proportional to the rendering work needed to pull off the effect. There's delineation between what are called hard or soft shadows, with the latter being preferable, since they more accurately model how shadows usually behave in the real world. There are several "good enough" methods that are generally favored by game developers. For more on shadows, check out Dave Salvator 3D Pipeline story..



http://www.r3.nu/~cass/shadowsandstuff/

Depth Testing

Now we come to depth testing, where occluded pixels are discarded and the concept of overdraw comes into play. Overdraw is pretty straightforward-- it's just the number of times you've drawn one pixel location in a frame. It's based on the number of elements existing in the 3D scene in the Z (depth) dimension, and is also called depth complexity. If you do this overdrawing often enough -- for instance to have dazzling visual special effects for spells, like Heretic II had, then you can reduce your frame rate to a crawl. Some of the initial effects designed in Heretic II, when several people where on screen throwing spells at each other, resulted in situations where they were drawing the equivalent of every pixel in the screen some 40 times in one frame! Needless to say, this was something that had to be adjusted, especially for the software renderer, which simply couldn't handle this load without reducing the game to a slide show. Depth testing is a technique used to determine which objects are in front of other objects at the same pixel location, so we can avoid drawing objects that are occluded.

click on image for full view

Look at this scene and think about what you can't see. In other words, what's in front of, or occluding other scene objects? Depth testing makes that determination.

I'll explain exactly how depth testing helps improve frame rates. Imagine a scene that's pretty detailed, with lots of polygons (or pixels) behind each other, without a fast way to discard them before the renderer gets them. By sort-ordering (in the Z-dimension) your non-alpha blended polygons so those closest to you are rendered first, you fill the screen with pixels that are closest first. So when you come to render pixels that are behind these (as determined by Z or depth testing), they get discarded quickly, avoiding blending steps and saving time. If you rendered these back to front, all the occluded objects would be rendered completely, then completely overwritten with others. The more complex the scene, the worse this situation could get, so depth testing is a Good Thing!

Antialiasing

Let's quickly review anti-aliasing. When rendering an individual polygon, the 3D card takes a good look at what's been rendered already, and will blur the edges of the new polygon so you don't get jagged pixel edges that would otherwise be plainly visible. This technique is usually be handled one of two ways. The first approach is at the individual polygon level, which requires you to render polygons from back to front of the view, so each polygon can blend appropriately with what's behind it. If you render out of order, you can end up with all sorts of strange effects. In the second approach, you render the whole frame at a much larger resolution than you intend to display it, and then when you scale the image down your sharp jagged edges tend to get blended away in the scaling. This second approach gives nice results, but requires a large memory footprint, and a ton of memory bandwidth, as the card needs to render more pixels than are actually in the resulting frame.. Most new cards handle this fairly well, but still have multiple antialiasing

modes you can choose, so you can trade off performance vs. quality. For a more detailed discussion of various popular antialiasing techniques used today, see Dave Salvator 3D Pipeline story.

Vertex and Pixel Shaders

Before we leave rendering technology, lets chat quickly about vertex and pixel shaders, since they are getting a fair amount of attention recently. Vertex Shaders are a way of getting directly at the features of the hardware on the card without using the API very much. For example, if a card has hardware T&L, you can either write DirectX or OpenGL code and hope your vertices go through the T&L unit (there is no way to be sure because it's all handled inside the driver), or you can go right to the metal and use vertex shaders directly. They allow you to specifically code to the features of the card itself, using your own specialized code that uses the T&L engines and whatever else the card has to offer to the best of your advantage. In fact both nVidia and ATI offer this feature in their current crop of cards.

Unfortunately, the way to address vertex shaders isn't consistent across cards. You can't just write code once for vertex shaders and have it run on any card as you can with OpenGL or DirectX, which is bad news. However, since you are talking directly to the metal of the card, it does offer the most promise for fast rendering of the effects that vertex shaders make possible. (As well as creating clever special effects too--you can affect things using vertex shaders in ways an API just doesn't offer you). In fact vertex shaders are really bringing 3D graphics cards back to the way that consoles are coded, with direct access to the hardware, and knowledge necessary of the best way to get the most out of the system, rather than relying on APIs to do it all for you. For some programmers this will be a bit of a coding shock, but it's the price of progress.

To clarify further, vertex shaders are programs or routines to calculate and perform effects on vertexes before submitting them to the card to render. You could do such things in software on the main CPU, or use vertex shaders on the card. Transforming a mesh for an animated model is a prime candidate for a vertex program.

Pixel shaders are routines that you write that are performed per pixel when the texture is rendered. Effectively you are subverting the blend mode operations that the card would normally do in hardware with your new routine. This allows you to do some very clever pixel effects, like making textures in the distance out of focus, adding heat haze, and creating internal reflection for water effects to mention just a few possibilities.

Once ATI and nVidia can actually agree on pixel shader versioning (and DX9's new higher-level shading language will help further this cause), I wouldn't be at all surprised to see DirectX and OpenGL go the way of Glide--helpful to get started with, but ultimately not the best way to get the best out of any card. I know I will be watching the coming years with interest.

In Closing...

Ultimately the renderer is where the game programmer gets judged most heavily. Visual prettiness counts for a lot in this business, so it pays to know what you're doing. One of the worst aspects for renderer programmers is the speed at which the 3D card industry changes. One day you are trying to get images with transparency working correctly; the next day nVidia is doing presentations on vertex shader programming. It moves very quickly, and for the most part, code written four years ago for 3D cards of that era is now obsolete, and needs to be completely reworked. Even John Carmack has made mention of how he knows that the cool stuff he coded four years ago to get the most out of 3D cards at that time is now commonplace these days -- hence his desire to completely rework the renderer for each new project id produces. Tim Sweeney of Epic agrees--here's a comment he made to me late last year:

We've spent a good 9 months replacing all the rendering code. The original Unreal was designed for software rendering and later extended to hardware rendering. The next-gen engine is designed for GeForce and better graphics cards, and has 100X higher polygon throughput than Unreal Tournament.

This requires a wholesale replacement of the renderer. Fortunately, the engine is modular enough that we've been able to keep the rest of the engine -- editor, physics, AI, networking -- intact, though we've been improving them in many ways.

Sidebar: APIs--A Blessing and a Curse

So what is an API? It's an Application Programming Interface, which presents a consistent front end to an inconsistent backend. For example, pretty much every 3D card out there has differences in how it implements its 3D-ness. However, they all present a consistent front end to the end user or programmer, so they know that the code they write for 3D card X will give the same results on 3D card Y. Well, that's the theory anyway. About three years ago this might have been a fairly true statement, but since then things have changed in 3D card land, with nVidia leading the charge.

click on image for full view

Right now in PC land, unless you are planning on building your own software rasterizer, where you use the CPU to draw all your sprites, polygons, and particles -- and people still do this. *Age of Empires II: Age of Kings* has an excellent software renderer, as does *Unreal* -- then you are going to be using one of two possible graphical APIs, OpenGL or DirectX. OpenGL is a truly cross-platform API (software written for this API will work on Linux, Windows and the MacOS), and has been around for more than a few years, is well understood, but is also beginning to show its age around the edges. Until about four years ago the definition of an OpenGL driver feature set was what all the card manufacturers were working towards. However, once that was achieved, there was no predefined roadmap of features to work towards, which is when all the card developers started to diverge in their feature set, using OpenGL extensions.

3dfx created the T-Buffer. nVidia went for hardware Transform and Lighting. Matrox went for bump mapping. And so on. My earlier statement, "things have changed in 3D card land over the past few years" is putting it mildly.

Anyway, the other possible API of choice is DirectX. This is Microsoft-controlled, and is supported purely on the PC and Xbox. No Apple or Linux versions exist for this for obvious reasons. Because Microsoft has control of DirectX, it tends to be better integrated into Windows in general.

The basic difference between OpenGL and DirectX is that the former is owned by the 'community' and the latter by Microsoft. If you want DirectX to support a new feature for your 3D card, then you need to lobby Microsoft, hopefully get your wish, and wait for a new release version of DirectX. With OpenGL, since the card manufacturer supplies the driver for the 3D card, you can get access to the new features of the card immediately via OpenGL extensions. This is OK, but as a game developer, you can't rely on them being widespread when you code your game. And while they may speed up your game 50%, but you can't require someone to have a GeForce 3 to run your game. Well, you can, but it's a pretty silly idea if you want to be in the business next year.

This is a **vast** simplification of the issue, and there are all sorts of exceptions to what I've described, but the general idea here is pretty solid. With DirectX you tend to know exactly what you can get out of a card at any given time, since if a feature isn't available to you, DirectX will simulate it in software (not always a good thing either, because it can sometimes be dog slow, but that's another discussion). With OpenGL you get to go more to the guts of the card, but the trade off is uncertainty as to which exact guts will be there.

Game Engine Anatomy 101

April 29, 2002
By: Jake Simpson

Character Modeling and Animation

How your character models look on screen, and how easy they are to build, texture, and animate can be critical to the 'suspension of disbelief' factor that the current crop of games try to accomplish. Character modeling systems have become increasingly sophisticated, with higher polygon count models, and cooler and cleverer ways to make the model move on screen.

These days you need a skeletal modeling system with bone and mesh level of detail, individual vertex bone weighting, bone animation overrides, and angle overrides just to stay in the race. And that doesn't even begin to cover some of the cooler things you can do, like animation blending, bone Inverse Kinematics, and individual bone constraints, along with photo realistic texturing. The list can go on and on. But really, after dropping all this very 'in' jargon, what are we really talking about here? Let's find out.

click on image for full view

To begin, let's define a mesh based system and its opposite, a skeletal animation system. With a mesh based system, for every frame of an animation, you define the position in the world of every point within the model mesh. Let's say, for instance, that you have a hand model containing 200 polygons, with 300 vertices (note that there usually isn't a 3-to-1 relationship between vertices and polygons, because lots of polygons often share vertices--and using strips and fans, you can drastically reduce your vertex count). If you have 10 frames of animation, then for each frame you have the data for the location of 300 vertices in memory. 300 x 10 = 3000 vertices made up of x, y, z, and color/alpha info for each vertex. You can see how this adds up real fast. Quake I, II, and III all shipped with this system, which does allow for the ability to deform a mesh on the fly, like making skirts flap, or hair wave.

In contrast, with a skeletal animation system, the mesh is a skeleton made up of bones (the things you animate). The mesh vertices relate to the bones themselves, so instead of the mesh representing each and every vertex position in the world, they are all positioned relative to the bones in the model. Thus, if you move the bone, the position of the vertices that make up the polygons changes too. This means you only have to animate the skeleton, which is typically about 50 bones or so--obviously a huge saving in memory.

Added Benefits of Skeletal Animation

Another advantage of skeletal animation is being able to 'weight' each vertex individually based on a number of other bones that influence the vertex. For example, movement of bones in the arms, the shoulders, the neck and even the torso can affect the mesh in the shoulders. When you move the torso, the mesh moves like a living character might move. The overall effect is a far more fluid and believable set of animations the 3D character can pull off, for less memory. Everybody wins.

Of course the drawback here is that if you want to animate something organic and very fine, like hair for instance, or a cape, you have to end up sticking a ton of bones in it for it to look natural, which drives up processing times a bit.

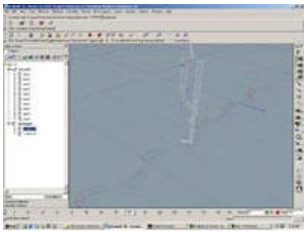
A couple of other things that skeletal based systems can give you is the ability to 'override' bones at a specific level--to say, "I don't care what the animation wants to do for this bone, I want to point it at a specific point in the world". This is great. You can get models looking around at events in the world, or keep their feet actually level on the ground they are standing. It's all very subtle, but it helps bring additional realism to the scene.

With skeletal systems you can even specify "I want this particular animation to be applied to the legs of the model, while a different gun-carrying or shooting animation runs on the torso of the model, and a different animation effect of the guy yelling runs on the head of the model". Very nice. Ghoul2 (Raven's animation system used in Soldier of Fortune II: Double Helix and Jedi Knight I: Outcast) has all this good stuff, and is specifically designed to allow the programmer access to all this override-ability. This saves on animations like you wouldn't believe. Instead of requiring one animation for the guy walking and firing, Raven has animations for the guy walking, and one for him firing standing still, and can combine the two for the situation of him walking and firing at the same time.

More Skeletons in the Closet

The previously described effects can be accomplished with a skeletal system that's hierarchical. What does that mean? It means that instead of each bone being located at a direct place in space, each bone is actually positioned relative to its parent. This means that if you move the parent bone, all the bones that are its children move too, with no extra effort on the code. This is what gives you the ability to change animations at any bone level, and have that filter down through the rest of the skeleton.

It is possible to create a skeletal system that isn't hierarchical--but at that point you can't override one bone and expect it to work. What you'll see is simply one bone in the body starting a new animation, but the ones under it adhering to the original animation, unless you implement some sort of 'info carry down' system. By starting with a hierarchical system in the first place, you get this effect automatically.

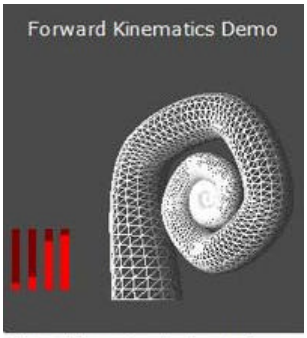


click on image for full view

Some of the newer features that are starting to show up in lots of the animation systems of today are things like animation blending, which is where you transition from one currently running animation to another over a small period of time, rather than snapping instantly from one to another. For example, you have a guy walking, but then he comes to a stop. Rather than just switching animations abruptly and having his feet and legs glitch to the idle stance, you blend it over ½ a second, so the feet appear to move to the new animation naturally. The effect of this cannot be overestimated--blending is a subtle thing, but used correctly it really makes a difference.

Inverse Kinematics

Inverse Kinematics (IK) is a buzzword thrown around by many people without much of an idea of what it really means. IK is a relatively new system in games these days. Imagine a hand, attached to an arm, attached to a body. Now imagine you hit the body hard. Usually the arm flails about, and the hand bobs about on the end of the arm. IK is the ability to move the body, and have the rest of the limbs move in a realistic way by themselves. Basically it requires a ton of information about each job to be set up by the animator--stuff like range of motion that the joint can go through, how much of a percentage this bone will move if the one in front of it does, and so on. It's quite an intensive processing problem as it stands right now, and while it's cool, you can get away without using it by having a diverse enough animation set. It is worth noting that a real IK solution requires a hierarchical skeletal system rather than a model space system--otherwise it all just gets far too time-consuming to work out each bone appropriately.



http://personal.nbnet.nb.ca/daveg/opengl/bones/index.html

IK has a reverse too, called Forward Kinematics. This is where you reverse the bone decision-making, and allow the programmer to move a hand, or a leg, and have the rest of the model's joints reposition themselves so the model is correctly oriented. For example, you'd say, "OK, hand, go pick up that cup on the table" and point the hand at the world position where the cup is located. The hand would move there, and the body behind it would sort itself out so the arms moved, the body bent appropriately, and so on.

LOD Geometry Systems

Lastly, we should quickly discuss Level of Detail (LOD) systems related to scaling the geometric complexity of models (in contrast to the use of LOD used when discussing MIP-Mapping). Given the vast scope of processor speeds that most PC games support these days, and the dynamic nature of any given visual scene you might render (is there one guy on screen, or 12?), you generally need some system to cope with situations such as when the system is nearly maxed out trying to draw 12 characters, all of 3,000 polygons each, on screen at once, while maintaining a realistic frame rate. LOD is designed to assist in such scenarios. At its most basic, it's the ability to dynamically alter the number of polygons you are using to draw a character on screen at any given time. Let's face it, when a character is far off in the distance, and has a height of maybe 10 pixels screen, you really don't need 3000 polygons to render the character--300 would probably do, and you'd be hard pressed to tell the difference.

click on image for full view

Some LOD systems will require you to build multiple versions of your models, and they will change LOD levels on screen depending on how close the model is to the viewer, and also how many polygons are being displayed at once. More sophisticated systems will actually dynamically reduce the number of polygons on screen at any given time, in any given character, on the fly--Messiah and Sacrifice include this style of technology, although it's not cheap CPU wise. You have to be sure that your LOD system isn't taking more time to work out which polygons to render (or not) compared to simply rendering the whole thing in the first place. Either approach will work, and with the amount we try to shove on screen these days, it's a very necessary thing. Note that DX9 will support adaptive geometry scaling (tessellation) performed by the hardware.

What it boils down to is getting a realistic looking model on screen that moves fluidly, and is visually believable in its representation and movement. Fluid animations come about most often through a combination of hand-built animations, and motion-captured animations. Sometimes you just build a given animation by hand--you tend to do this a lot when you are animating a model that is doing something you can't do in real life--for example, you can't really bend over backwards, or do an ongoing bicycle kick like Lui Kang in Mortal Kombat 4, so motion capture is generally out! Usually motion captured animations--actually video capturing a living actor going though the motions of what you want to see on screen--is the way to get realistic stuff. And realistic stuff can make an average game look great, and cover lots of things. NFL Blitz for instance, shipped with models on screen of about 200 polygons. They were horribly blocky to look at standing still, but once they had fast fluid animations running on those models, lots of the ugliness of the models themselves went away. The eye tended to see the 'realistic' animations rather than the construction of the models themselves. A decent animator can cover over a multitude of modeling sins.

I hope that gave you some insight into modeling and animation issues. In Part V, we'll get deeper into building 3D worlds, and discuss a bit about physics, motion, and effects systems.