Game Engine Anatomy 101, Part I

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.

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 …

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!

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.

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.

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.

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.

> *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.

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.

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.

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.

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, Part II

April 15, 2002

By Jake Simpson

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.

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.

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 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.

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.

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.

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.

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, Part III

April 19, 2002

By Jake Simpson

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.

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.

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.

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..

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.

> *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!

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.

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.

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.

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.

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, Part IV
April 29, 2002

By Jake Simpson

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.

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.

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.

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.

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 (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. Using IK, the programmer can 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.

IK has a reverse too, called Forward Kinematics, which works essentially in opposite order of IK. 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.

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.

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.

Game Engine Anatomy 101, Part V
May 8, 2002

By Jake Simpson

Quite often when building a game that has any 3D component to it, you end up trying to build a 3D environment in which the game action will take place. Somehow the game developer has to come up with a way of building this environment so it's easily modifiable, efficient, has a low polygon count, and is easy for the game to both render and to perform physics against. Simple, right? What do I do with my left hand while doing this? Yeah. Right.

While there are plenty of 3D architectural programs out there, from CAD/CAM programs to 3D Studio Max, building a game world is a different kettle of fish from just building models of world interiors or exteriors. You have polygon count issues--any given renderer can only render so many polys in one go, and this is never enough for the talented level designer. Not only that, you can only store a predefined number of poly's per level too, so even if your renderer can handle 250,000 poly's in view, if you can only store 500,000 poly's in a reasonable amount of space, then depending on how you go about it, you can end up with as little as two rooms worth of level. Not good.

Either way, the developer needs to come up with a creation tool--preferably one that is flexible enough to allow the kinds of things game engines require -- like placement of objects in the world, a decent preview of the level before it goes into the game, and an accurate lighting preview. These capabilities allow game developers to see how the level is going to look before they spend three hours pre-processing the level to produce an 'engine-digestible' format. Developers need relevant information regarding the level, poly counts, mesh counts, and so on. They need a nice and friendly way of being able to texture the world, easy access to polygon count reduction tools, and so on and so forth. The list goes on.

It is possible to find this functionality in pre-existing tools. Many developers use Max or Maya to build their levels, but even 3DMax requires some task-specific extensions to its functionality to do the job of level building efficiently. It's even possible to use a purpose-built level building tool, like QERadient (see below), and reprocess its output into a format your own engine can interpret.

Recall in Part I we discussed BSP (Binary Space Partitioning) trees, and you may have also heard of the term Potentially Visible Sets (PVS) being bandied around. Both have the same goal, and without delving into the hairy math involved, it's a way of breaking down a world into the smallest subset of walls that you can see from any given position in the world. When implemented, they will return only that which you can see, not walls hidden behind those that would have been obscured. You can imagine the benefits this gives to a software renderer where every pixel rendered (or not as the case may be) counts enormously. They also returns those walls in back to front order, which is handy when rendering, since you can be sure where an object actually is positioned in render order.

BSP trees by and large have been falling out of favor of late because of their several quirks, and because with the pixel throughput we get on 3D cards these days, coupled with the Z-buffer pixel test, a BSP is often a redundant process. They are handy for working out exactly where you are in a world, and what geometry is immediately around you, but there are often better and more intuitive ways for storing this information than BSP trees.

A Potentially Visible Set is pretty much as it sounds. It's a method for determining what surfaces of a world, and what objects, are actually in view at any given time, given your location in the world. Often this is used for culling objects before rendering, and also for culling them to reduce AI and animation processing. After all, if you can't actually see it, why bother processing. More often than not it really is immaterial if a non-player character (NPC) is animating, or even performing his AI thinking.

Now that we've got your world structure in memory, we have to stop our characters from falling out of it, and deal with floors, slopes, walls, doors, and moving platforms. Plus we must handle gravity, velocity changes, and inertia correctly, as well as colliding with other objects placed in the world. This is considered game physics. And before we go any further, there is one myth I'd like to dispel right here and now. Anytime you see physics in a world, or anyone claiming "real physics" in a complex gaming environment, well, it's BS. More than 80% of the energy of building an efficient gaming physics systems is in finding ways to shortcut the real equations that one should be using to handle objects in a world. And even then, often you'll ignore what's 'real' and create something that's 'fun' instead, which is, after all, the aim here.

Quite often gamers will ignore the classic Newtonian physics of the real world, and play their own, more fun, versions of reality. For instance in Quake II you can accelerate from 0 to 35MPH instantly, and stop just as fast. There is no friction, and slopes don't offer the same kind of gravitational issues that real slopes do. Bodies don't have gravity acting on all the joints as they should do--you don't see bodies slumping over tables or edges as they would in real life - and gravity itself can even be variable. And let's face it, in the real world, ships in space do not act like World War II fighters with air operating on their surfaces. In space, it's all force and counter forces, and force acting around a weight point, and so on. Not screaming around like Luke Skywalker in an X-Wing. But it is more fun to do that though!

Speaking as game developer, whatever we do, we need to be able to detect walls, detect floors, and handle collisions with other objects in the world. These are musts for the modern gaming engine - what we've decided to do further to them is really up to us and what our game demands.

Most game engines these days have some kind of effects generator built in that allows us to put up all that lovely eye candy that the discerning gamer has come to expect. However, what goes on behind the scenes with effect systems can radically affect frame rates, so this is an area we need to be quite concerned about. These days we have great 3D cards that we can hurl gobs and gobs of triangles at, and they still ask for more. It wasn't always that way. During Heretic II, with its lovely software-rendering mode, Raven ran into some pretty severe overdraw problems due to their pretty spell effects. Recall that overdraw occurs when you draw the same pixel on screen more than once. When you have lots of effects going on, by their very nature you have lots of triangles, and chances are multiple triangles are stacked on top of each other. So you wind up with lots of re-drawing of the same pixel. Add in alpha, and that means you have to read the old pixel and mix it in with the new one before re-drawing it, and it gets even more CPU intensive.

Heretic II got to the point with some effects where we were re-drawing the entire screen some 40 times in one frame. Pretty scary, eh? So they implemented a system within the effects system that sampled the frame rates over the last 30 frames, and if things started getting slow it automatically scaled down the number of triangles rendered for any given effect. That got the primary job done, keeping the frame rates up, but man did some of the effects look ugly.

Anyway, because most effects these days tend to use lots and lots of very small particles to simulate fire and smoke and so on, you end up processing numerous triangles per frame in your effects code. You have to move them all from one frame to the next, decide if they are done, and even do world physics on them so they bounce on the floor appropriately. This is all pretty expensive on the PC, so even now you have to have practical limits on what you can do. For example, it would be nice to generate fire using one-pixel particles, but don't expect to be doing much else on screen while you are doing it.

Particles are defined as very small renderable articles that all have their own world position and velocity. They are different from oriented sprites, which are used for big particles--stuff like smoke puffs. They automatically orient towards the camera and typically do things like rotate, scale, and change their transparency levels so they can fade out over time. We tend to see lots of particles, but we limit the number of sprites--although the real difference between them is blurring these days. In the future, particularly after DX9 and more advanced graphics hardware surfaces, we may see more people using procedural shaders to perform effects similar to or better than particle systems, creating very cool animation effects.

When talking about effects systems you may hear the word 'primitives' bantered about. A primitive is pretty much the lowest level of physical representation of an effect that your system will handle. To explain a bit more, a triangle is a primitive. That's what most engines end up rendering at the bottom level--lots and lots of triangles. As you go up through the systems, your definition of a primitive changes. For instance the game programmer at the top level doesn't want to think about dealing with individual triangles. He just wants to say, "This effect happens here" and let the system handle it in a sort of black-box fashion. So to him, an effect primitive is "Let's spawn a bunch of particles at this point in the world for this long with this gravity on it". Inside of the effects system, it might consider an effect primitive to be each individual effect it is generating right then, like a group of triangles all following the same physics rules - it then hands off all of the individual triangles to the renderer for rendering, so at the renderer level the primitive is an individual triangle. A bit confusing but you get the general idea.

That wraps Part V, and the next segment it's all about sound systems, and the various audio APIs, 3D audio effects, dealing with occlusions and obstructions, how various materials affect sounds, audio mixing, and so on.

Game Engine Anatomy 101, Part VI

May 17, 2002

By Jake Simpson

Sound and music in games are becoming increasingly important in recent years due to both advances in the game genres people are playing (where sound is an actual game play feature, such as the aural cues in Thief and other games of that ilk), and in technology. Four speaker surround systems are now both affordable and commonplace in the gamer's arsenal. Given spatialization of sound, obstruction and occlusion of noise, and the dynamic music many games employ these days to heighten the emotional responses of the player, it's no wonder that more care is being given to this area.

Right now in the PC arena, there is really only one card of choice for gamers – the Sound Blaster Live!. From old time PC sound card manufacturer Creative Labs. Creative has provided their EAX sound extensions for DirectX for a number of years now, and they are a founder of the new OpenAL (Open Audio Library) initiative. OpenAL, as it sounds, is an API for sound systems in the same way that OpenGL is a graphical API. OpenAL is designed to support a number of features that are mostly common across sound cards, and provide a software alternative if a specific hardware feature is not available.

For a better definition of OpenAL, I asked Garin Hiebert of Creative Labs for a definition:

> *"Borrowing from our "OpenAL Specification and Reference" here's a definition:*
>
> *OpenAL is a software interface to audio hardware, providing a programmer with the ability to produce high-quality multi-channel output. OpenAL is foremost a means to generate audio in a simulated three-dimensional environment. It is intended to be cross-platform and easy to use, resembling the OpenGL API in style and conventions. Any programmer who is already familiar with OpenGL will find OpenAL to be very familiar.*

> *The OpenAL API can easily be extended to accommodate add-on technologies. Creative Labs has already added EAX support to the API, which programmers can use to add sophisticated reverberation, occlusion, and obstruction effects to their audio environment. "*

Soldier of Fortune II features this new system, as does *Jedi Knight II: Outcast*, along with the Eagle world/sound properties editor. What is Eagle? Before we get into that, let's discuss a couple of other systems, and define some sound terms.

Another system out there is the Miles Sound System. Miles is a company that produces a plug-in to your own code that will handle all the necessary talking to specific sound cards (like the Sound Blaster Live! series, or an older A3D card for example) while getting the most of each card. It's very much like an API front end, with extra features bundled in. Miles gives you access to things like MP3 decompression amongst other things. It's a nice one-stop solution, but like everything, it costs money as well as being one extra layer between your code and the hardware. For fast sound system production though, it's very useful, and they've been around for a while, so they do know their stuff.

Let's start with obstruction and occlusion. They sound the same, but are not. Occlusion means basically that the listener has some encompassing obstacle between them and a sound being played.

Let's say you hear bad guys inside the house in this screenshot from NOLF 2. You can hear them, but their sounds are pretty muffled and muted. Obstruction is similar, but the obstacle between you and the sound is not encompassing. A good example of this is having a pillar between you and the sound source. You are still hearing the sound due to echoes in the room, but it isn't the same as the sound coming directly to your ears. Of course this does rely on knowing what is in a direct line between your ears and the sound source. And the required processing can get pretty time consuming depending on the size of the room, the distance of the sound source from you, and so on. We will talk about traces later-- suffice to say it can often be the reason for slower frame rates. The A3D code inside of *Quake III* does this stuff, and frame rates can often be improved by turning these options off. Tribes 2 is another sufferer from this malady. Turn off the 3D Sound options and your frame rates instantly get better, which makes sense when you consider how big the *Tribes* worlds are, and how far you can see.

Next are sound material properties. Most sound cards give you the capability to modify sounds being played by using definable filters to act on that sound. For instance, there is a big difference between hearing a sound underwater, or in a cloth-covered room, or in a long corridor, or an opera house. It's pretty cool to be able to change the way you hear a sound depending on the environment you are in.

Back to Eagle… This is an editor that allows most first-person shooter map designers to import their maps into the tool, and then construct simplified geometry that creates a sound map for EAX code in the actual game engine. The idea is you don't need the complex geometry of a real graphical map to simulate sound environments. You can also assign sound materials to the resulting simplified map so sound environments can change dynamically. I was witness to a demonstration of this on *Soldier of Fortune* and *Unreal Tournament* and it really is quite striking. Hearing all the sounds change when you plunge into water is a very immersive experience.

OK, let's move on.

For the consoles, you are more limited in your possibilities because of static hardware – although both on the PlayStation 2 and the Xbox this hardware is pretty damn cool. When I say limited, I only mean in expansion, not in what it's capable of doing. I wouldn't be the least bit surprised to see games with Dolby Digital 5.1 output very shortly from these consoles. Xbox, with its MCP audio processor can encode any game's audio into 5.1, and the game doesn't need to be specially coded to take advantage of this feature. Dolby has brought ProLogic II to the PS2, and has teamed up with Factor 5 to enable ProLogic II for GameCube games. On Xbox, game titles *Halo, Madden 2002* and *Project Gotham Racing* all have 5.1 Dolby Digital audio content. DTS also recently unveiled

an SDK for PS2 game developers to bring a reduced bit-rate version of DTS audio to games on that platform.

Now there are some issues with sound spatialization that not many have dealt with. I speak of putting sound in a real 3D world. Having four speakers around you is a great start, but it is still only in two dimensions. Without speakers above you and below you, you really aren't getting 3D sound. There are some sound modulation filters that attempt to address this, but really there is no substitute for the real deal. Of course most games are really only played in two dimensions for the most part anyway, so it's not such a big deal. Yet.

One of the most important features of any sound system is actually the mixing of the sounds together. Once you've decided what sounds you can actually hear based on where you are located, where the sounds are in space, and what the volume is for each sound, then you have to mix the sounds. Usually the sound card itself handles this, which is the primary reason for the existence of the card in the first place. However, some engines out there decide to do a 'premix' in software first. That doesn't make much sense really, until you look at a bit of history.

When sound cards first came out there where many different approaches to mixing. Some cards could mix 8 sounds together, some 16, some 32, and so on. If you always want to hear 16 possible sounds, but you don't know if the card can handle it, then you fall back to the tried and tested route-- that of doing the mixing yourself in software. This is actually how the *Quake III* sound system works, but begs the question: "*Quake III* was released into a world of A3D and Sound Blaster Live! Cards, which are more standardized than ever, so why do this?" That's a good question. The sound system for *Quake III* is actually almost line-for-line the same sound system that was in Quake II. And Quake I, and even *Doom*. When you think about it, up until A3D cards and the SB Live! Card, sound systems hadn't really changed in requirements for years. Two speakers, two dimensions, and a simple volume drop off for distance. From *Doom* through *Quake III* not much has changed. And in the games industry, if it ain't broke, don't fix it.

Usually you would just use DirectSound to do your mixing for you, since it would detect the sound hardware available, or fall back to software, much as DirectX does for 3D cards. In 90% of sound cases, a software fall back doesn't really make that much difference to your frame rate. But the *Doom* engine was created when Direct Sound wasn't even a glimmer in some mad coder's eye. It never got updated because it never really needed to.

Of course, you could use some of the clever features of the SoundBlaster Live! Card, such as a room's echoic characteristics: a rock room, or an auditorium, a cavern, a football stadium, etc. And you really should use the mixer provided by the hardware, after all, that's what it is there for. One drawback of this approach is that often the result of the mix isn't available to the program itself, as it's being done inside the card rather than in main memory. If you need to look at the resulting volumes for any reason, you are out of luck.

We haven't talked much about music generation in games. Traditionally there are two approaches, one being a straight music .wav file (or equivalent). It's pre-made, ready to run, and a minimum of fuss. However, these are expensive in terms of memory and playback time. The second approach is to code up a MIDI track using preset samples. This is often cheaper in memory, but has the drawback of having to mix several sounds together at once, thereby using up sound channels.

Dynamic music is the ability to change your music depending on the action being witnessed in the game, such as slow music for exploration, and fast for combat. One thing that's hard to do with pre-made music is to beat-match it so you can fade from one piece of music to another, which is easy with a MIDI track. Often though, if you do the fade fast enough, or fade one down before bringing up the other, you can get away with it.

By the way, before we leave this subject it's worth mentioning that companies exist that specialize in creating purpose-composed music for your games. The FatMan (www.fatman.com) is one such company. It's probably easier to contract out music than anything else, which is how they exist.

Lastly of course, the in-thing for games right now is the MP3 format, which allows massive 11-to-1 compression

of sound samples, yet only takes a fraction of the CPU time to decompress them before being thrown at a sound card. When I was at Raven Software, with *Star Trek Voyager: Elite Force* we managed to get three complete languages on one CD by use of MP3, and still have space left over for more graphics. Mainly, we MP3 only for the non-player characters (NPC) voices, since streaming an MP3 and decompressing it on the fly for all of the game's audio effects was more than the hardware could handle, although it's a definite possibility in the future. Newer formats like AAC from Dolby and WMA from Microsoft offer equal or superior audio quality at nearly twice the compression rate (well, actually half the bit rate) of MP3, and may find their way into future game titles.

That's it for this segment. Next up will be developing for networked and online gaming environments.

Game Engine Anatomy 101, Part VII

May 23, 2002

By Jake Simpson

I remember sitting in on a lecture at the GDC a couple of years back titled "*The Internet Sucks*" by the guys responsible for *X-Wing Vs TIE Fighter*, which was all about getting network games to work in real time over the Internet. How right they were in choosing that title. It does suck indeed when it comes to dealing with dropped or lost packets, out of order packets, latency (the time it takes for one packet to get from you to where it is going), and so on. Yet it is possible. It requires some cleverness and experience with the Internet, but it is definitely possible. Witness the host of online games we see today, from *Quake III*, *Unreal Tournament*, *Counter Strike* all the way to *EverQuest* and *Ultima Online*.

Most games that are serious about longevity these days have at least some online component to them. Most purely single-player experiences tend to get played once, maybe twice, or even three times if it's a damn good one, but once that's done, up on the shelf it goes. If you want any kind of longevity, then online multiplayer is where it's at, and that means dealing with the Internet, and opening that Pandora 's Box for coders.

So what's involved in dealing with the Internet? First up is an understanding of how the Internet operates, and a quick discussion of peer-to-peer and client/server architectures. Peer to peer is where you set up a game on two machines, and simply share inputs between them. Each individual game assumes it's correct, and just incorporates the input from the other machine in its frame-by-frame updates. Client/server is where effectively one machine is running the game, and everyone else is just a terminal that accepts input from the player, and renders whatever the server tells it to render.

The advantage of client/server is that every machine will be showing the same game, since all processing is done in one place, not across multiple machines where you can get out of synch with each other. The drawback is that the server itself needs to have some serious CPU time available for the processing of each of the connected clients, as well as a decent network connection to ensure each client receives its updates in a timely fashion.

We've all heard of TCP/IP (Transmission Control Protocol / Internet Protocol) and UDP (User Datagram Protocol), and there's a tremendous amount of deep technical information on the Web about the protocols. In fact, here's a page from Cisco with some excellent TCP/IP tutorials. We'll cover a few of the TCP/IP basics at a high level, with the goal of giving you a better understanding of challenges faced by networked game designers using these standard protocols.

TCP/IP and UDP/IP are two levels of communication protocol systems. The IP figures out the transmission of packets of data to and from the Internet. UDP or TCP hands it a big fat packet of data, and IP splits it up into sub packets, puts an envelope around each packet, and figures out the IP address of its destination, and how it should get there, and then sends the packet out to your ISP, or however you are connected to the Net. It's effectively like writing down what you want to send on a postcard, stamping it, addressing it, and stuffing it in a mailbox, and off it goes.

UDP and TCP are higher layers that accept the packet of data from you the coder, or the game, and decides what to do with it. The difference between UDP and TCP is that TCP guarantees delivery of the packets, in order, and UDP doesn't. UDP is effectively a pathway to talk directly to IP, whereas TCP is an interface between you and IP. It's like having an admin assistant between you and your mail. With UDP you would type up your letters yourself, put them in an envelope, etc. With TCP you would just dictate the letter to your admin, and the admin would do all the work and follow up to be sure the letter arrived.

However, all this wonderful work-done-for-you comes at a cost. In order to be sure that packets that are sent via the Internet get there intact, TCP expects an Acknowledgement (an ACK in net parlance) to be sent back from the destination for every packet it sends. If it doesn't get an ACK within a certain time, then it holds up sending any new packets, resends the one that was lost, and will continue to do so until the destination responds. We've all seen this in action when you've gone to a web page, and half way through the download it stops for bit and then restarts. Chances are (assuming it's not an ISP problem) a packet has been lost somewhere, and TCP is demanding it gets resent before any more come down the pipe.

The problem with all this is the delay that occurs between the sender realizing something is amiss, and the packet actually getting through. This can take multiple seconds sometimes, which is not that much of a worry if you are just downloading a file or a web page, but if it's a game packet, of which there are at least 10 per second, then you're in real trouble, especially since it's holding up everything else. This is actually such a problem that almost no games use TCP as their main Internet protocol of choice, unless it's not a real-time action game. Most games use UDP--they can't guarantee order or delivery, but it sure is fast - or at least faster than TCP/IP usually ends up. So now we understand that, what's next?

Since UDP is obviously the way to go for fast-response games, we will have to handle lost packets and out of order packets ourselves. And that's where it gets tricky. Without giving away too many coding secrets, I can say there are ways. For a start, there is Client Prediction, a word that's been bandied around a fair amount. Client prediction kicks in when you as a client are connected to the big server, but aren't seeing updates from the server consistently. The part of the game that is running on your computer looks at the input you are giving it, and makes a 'best guess' at what it thinks should be going on, in the absence of any overriding information from the server. It will display the guessed information, and then correct itself if need be when it does get the latest state of the world from the server. You'd be surprised at the effectiveness of this method. By and large, packets don't tend to be missing for any large period of time--most of the time it's fractions of a second, in which case there isn't too much time for the game to deviate from what the server actually believes is going on. Since the deviation does get larger over time, most games have a time-out function in them, stopping game play when too much time without contact from the server occurs.

The type of game you are creating can matter here--first person shooter games don't require such effective client prediction, since it mostly just deals with "where am I, and did I shoot or not?" With a third-person game you have to be much more accurate, so you can correctly predict the animation that your character is playing, so the action is smooth. Fluid animation in this situation is absolutely essential. *Heretic II* has enormous problems in this area, and was one of the hardest things Raven has had to deal with when it came to network coding.

Of course if you have a decent network connection, such as a broadband connection, then these issues are far less important. Having a broader pipe for larger packet sizes, and faster access works wonders for your ping time. In fact, the main advantage of broadband for gaming isn't so much the fatter pipe, but rather the greatly reduced latency, particularly on your first hop to your ISP. With 56K modems, this first hop can typically be 100ms, which has already severely increased your potential ping time to any game server on the Net. With broadband connections like DSL for instance, that first hop latency is more like 20ms. You can figure out your first-hop ping time by using a command-line program in Windows called TraceRoute (TRACERT.EXE) where you specify a target IP address or domain name. Watch the first hop very closely, as this is almost always your ping time to your ISP. And also watch how many hops you're making inside your ISP's network until you see a different domain name listed on a given hop.

Mind you, broadband doesn't always solve lag issues. You are still at the mercy of the slowest router/server and number of hops your packets travel through on their journey from the server to you and vice versa. Having a broadband connection does tend to mitigate these, but it's not like they just end up going away. Of course, if you are going to run a server of any sort, you will require broadband with a reasonably fast upstream data rate, since a modem simply cannot handle the load that a server spits out.

It's probably worth mentioning that if you want to play network games on either the PS2 or the Xbox you will need a broadband connection, since neither supports a modem.

Something else that has to be handled is packet size. If you have a game with 64 people all running around blasting at each other, the data in the packets sent from one machine to another can get pretty large, to the point where some modems just don't have the bandwidth to deal with it. This is becoming particularly relevant to those games having large terrain systems. The added issue here is since you have this nice terrain system, you can see pretty far, and therefore can see a lot of other players, making the amount of data you need from the server for accurate rendering grow at very fast rate. What can we do about this?

Well, first up it's essential to only send what's absolutely necessary to any given client, so he only gets what he needs to view the game from his vantage point. No point in sending data on people outside of his view--he's not going to see it. Also, it's good to ensure you only send data that's actually changed from frame to frame. No point in re-sending data if the guy is still running the same animation. Of course this does give problems if packets are dropped, but that's why good network coders are paid the big money, to handle stuff like this.

There are other things to deal with too. Recently there has been a distressing amount of online cheating going on. This is where someone modifies the game to give them an unfair advantage. Although this isn't strictly part of networking, it does come into it. Sometimes people will create mods that allow them to instantly target anyone who comes into view, or one that simply allows them to see through walls, or make themselves invisible to other players. Most of the time these things can be dealt with inside the networking layer, or on the server. Anyone with a 100% hit rate simply gets kicked off, since it's not humanly possible.

Game developers must do as much as possible to stop the cheating, but unfortunately, what's man-made can be man broken. All you can do is make it difficult, and try and detect it when it does happen.

Well, that's it for now. Coming up in Part VIII, we'll look at the interesting world of game scripting systems that assist in story telling by rendering or enabling predefined scenes and actions based on events that occur during game play.

Game Engine Anatomy 101, Part VIII

June 21, 2002

By Jake Simpson

We move from game networking issues in Part VII to scripting systems, which have emerged as a big game element recently, because of the story telling opportunities they present. Given a situation that needs explanation in a controlled way, pre-scripted cinematics is the way to go. In movies, this is taken care of usually by either the protagonist explaining the situation to a sidekick, or by the enemy explaining to the hero. There are, of course, other ways to do it - narrators, flashbacks and so on--but generally it's done using real-time situational people and events. Games are different of course, and game developers shouldn't do too many flashbacks in their average FPS, because that would usually require a new environment or level being loaded in, and new textures and/or models too. All of this extra processing and rendering can impact performance of the main game sequences. You could reuse scene elements already stored in memory for flashbacks, but that could look obviously cheesy.

Star Trek Voyager: Elite Force by RavenSoft made extensive use of scripted sequences to generate both in-game events and cut scenes using the game engine itself.

An interesting trend for designing scripted scenarios in games is to use the current greatly improved 3D game engines themselves to create cut-scenes. This may seem fairly obvious now, but years ago, when 3D graphics were more primitive, cut-scenes would often be done using a high-end 3D workstation, and a resultant 3D animation would then be recorded as a digital video file, and streamed off the CD-ROM. Then you went from the beautiful graphics of the cut-scene back to the relatively uninspiring 3D of the actual game, and it was quite the jarring letdown. But games like *Half-Life* and *Star Trek Voyager : Elite Force* made very good use of their own engines to produce all cut-scenes, and the result is a much smoother transition between cut-scenes and game-play.

It's probably a good idea to differentiate scripting from AI. Scripting is where you take complete control over a given scene, setting up events that the player almost always has no control over, where the gamer is "on a rail" in order to move though to a given plot point, or set up a situation the player needs to resolve. A good example might be a case where boulders are dropped down a corridor requiring the player to find a new way to escape.

There are a couple different types of scripting systems available these days for programmers or artists, and it takes a very methodical and logical mind to do this stuff properly. The first is the simple text-based, single-threaded style, just like we programmers are used to coding. In many cases it's actually based on C, albeit in a very simple form. Lots of "if this, then do that" kind of stuff. Most scripting tends to be fairly linear in scope--that means that it's usually comprised of many commands that follow each other. Move guy A to point B in the world. When you've done that, make him speak, then when that's done, move him to point C. Pretty simple stuff.

Then there's the complicated stuff--allowing multiple threads, and actually allowing variable situations. Variable situations are those where you don't actually know for sure who's around when the script starts, but you have to write the script in such a way that it will work with whoever is around. For instance--a normal straightforward script would have three guys, all predefined, all with a set situation they will discuss. A variable script will have three people, not one of which you can guarantee is one person in particular, and has to work the same way. Or in an extreme case, maybe only two, or even one guy will be there, which makes having a three-way conversation kinda hard!

One big issue Raven faced in *Star Trek Voyager: Elite Force* was the situation where the user would want to take a character from one part of the ship to another, but the route from point A to point B might change radically game to game. For example, they need to get Munro (the game's main character whom you play as) from the engine room to the transporter room. Unfortunately due to the non-linearity of the game, you might have destroyed the turbo lift during the events leading up to this, or perhaps a Jeffries tube was damaged beyond traversal. Given that we don't know the state of the world when the script commences, we are forced to create a script for almost every eventuality to cover these 'what if' situations. And it only gets worse from there. Some of the situations we can set up provide so many possible combinations of situations that it's almost impossible to accurately test each eventuality for a satisfactory conclusion. Just talk to anyone who worked on *SiN*, *Star Trek Voyager : Elite Force* or *Deus Ex*. QA departments traditionally hate these kinds of titles because it made their job 50 times more difficult than it was already.

You can imagine how hard it is to script for those situations. But that's what the non-linear game path of today demand, and why it takes a superior development house to be able to pull it off.

Late last year I interviewed Jim Dose, - ex-Ritual developer and now a developer at id Software and designer of the scripting system (among other things) going into Doom3. While the interview is a bit older, it's still very insightful.

Jim had this to say of scripting systems and creating an easy to use and robust system (as opposed to including all the features that designers traditionally want to use):

> *The hardest part of designing a script system is knowing when to stop. Once you have it up and running, you discover that there are lots of systems that can take advantage of it. For Sin, the original idea was just to have an easier method for level designers to describe how dynamic objects moved around the environment. By the end of the project we were also using it for syncing sounds and game events to animation, keeping track of mission objectives over multiple levels, controlling the layout of the HUD and user interfaces for in-game computer consoles, describing how AI reacted to different situations, and particle systems.*
>
> *Controlling complexity can also be quite difficult. When you put the power of scripting into the hands of creative people, they begin to explore the bounds of what they can do. Often, they're inspired to do something that's just slightly out of reach of the system. It's very easy to get caught up in adding "just one more feature" to allow them to do what they want. As the language grows, a language structure that may have made sense for the initial spec may be grossly over-extended. At some point, it makes sense to rethink the system, but at that point you have probably amassed a huge amount of script that would have to be rewritten. Sin suffered from this, as did FAKK 2. I didn't get the opportunity to give the scripting system a massive overhaul until I rewrote it for Rogue's 'Alice'.*

Amen Jim -Raven has seen exactly this occur with their ICARUS system. ICARUS is effectively the same kind of scripting system Jim describes above, and was responsible for all the scripted events in *Star Trek: Voyager: Elite Force*. It's been reused in *Soldier of Fortune II* and *Jedi Knight II : Outcast*. Much has been re-written to address new issues the system was required to handle not foreseen / required in the original implementation.

The second type of scripting is the visual scripting systems. Doing it this way, instead of using a text file coding approach, you can actually build your scripting using real characters in a real gaming environment. You can trace paths for characters to follow in the world, define animations to be used, and generally get a much better idea of exactly how your script is going to look. It doesn't honestly help that much with the non-linear problems we've already discussed, but it does make creating the initial script a hell of lot faster.

Again, Jim Dose on visual scripting systems.

> *Visual scripting systems definitely have their uses, but tend to be more difficult to implement and, if poorly designed, are prone to confusing the developer as complexity rises. For example, AI can be designed visually using a flowchart-like structure. You can visually represent how a creature behaves very easily using boxes to represent states and arrows, to other states indicating ways the character can transition from state to state.*
>
> *One common use of scripting is to control objects in the game world, dictating how they move through the world. The ability to visually move the objects to key positions and to play the entire movement in an editor may be more intuitive to a designer. It does have its limits, however, as another interface would be necessary to design any decisions that the object must make during its movement. That ability is what separates a scripted sequence from an animation created in a program like 3DS Max or Maya.*
>
> *At some point, the user may need some way to determine why a script isn't doing what they expect it to. Some form of debugging tools can make this task a lot easier. At a minimum, some way to determine which scripts are running and the current location in the script is necessary. Also helpful is the ability to inspect variables, and start, stop, and single step through a script. Usually, a programmer can get by debugging in their debugger, but the process tends to take much longer than if some built-in script debugger is available.*

That wraps up Part VIII, and in the next segment we'll discuss the merits of using off-the-shelf versus custom

game engine design tools, and then dive into game control mechanisms, and developing game objects, plus things that go 'bang' (weapons systems).

Game Engine Anatomy 101, Part IX
June 27, 2002

EDITOR RATING:

By Jake Simpson
We move from Scripting Engines in Part VIII, to a number of topics in this segment that we think hard-core gamers and those aspiring to be game developers will find quite interesting. We'll start with a discussion on off-the-shelf versus custom design tools.

Selection of your tools is a VERY important part of your engine design because this is what you will use to produce content for your game, which is the most time-consuming part. Anything that aids in this process by saving time and resources is **good**. Anything that doesn't is **bad**. There, that was easy.

Of course it's not that easy. There's a lot more to this than might immediately meet the eye. Choice of your tool set, and asset pathing from tool to game is a lot trickier than it sounds, and is influenced by factors such as suitability for the task at hand, cost, content producer familiarity, market penetration, tool support, and so on. When considering off-the-shelf tool selection, or even when developing your own tools, remember that the developers actually doing the work had better be able to do what needs to be done with the tool. Some of the off-the-shelf tools can get up there in price, and when you get into multiple copy licenses, the costs skyrocket.

Then there is the alluring possibility of producing your own tool set from scratch, purpose-designed for what your game and the engine requires. This of course requires time, and no small amount of kung fu on the part of your programmers to produce what's required in a developer-friendly format. Whipping up Windows-based file converters is one thing, building a complete level design tool from scratch is quite another. On the other hand, if you do take this route you end up with tools that no one else out there in Game Developer Land has, so your stuff will look unique. And standing out from the crowd these days is a very desirable thing, given all the competition.

Of course with in house tool development, you need someone that will make all those little tweaks and changes that are inevitable. But the real point here is that this is possible. With off-the-shelf tools, the tool developer will rarely change their output file format to add a couple of features that just you need. So your stuff ends up looking a little more generic, or you have to take an extra step with another utility to get the desired result, which of course takes more developer time.

It's worth bearing in mind that a lot of the big name 3D tools out there these days have been around a while, and are producing fairly bug free products, and more importantly, have some degree of experience in what they do.

If you choose to build your own tool then more often than not you are a) re-inventing the wheel to some degree and b) going to run into the same issues that those people building the off the shelf tools have, only they've already solved them. Very often people building a single, specific tool have invested considerable time and effort, and have come up with a tool that far outstrips your own personal requirements. Plus they've typically incorporated features you either wouldn't have thought to be useful, or didn't have time to implement yourself. This is quite a compelling argument for third-party software. Usually most game development processes end up with a mixture of home grown file converter tools, off the shelf content creation tools, and usually some extra plug-ins for those tools to add some necessary specific functionality. Off the shelf tools go a long way in offering functionality you will inevitably require, but just as inevitably, there's always something you don't get that would be very useful, helpful, or plain necessary. A small plug-in might be a good substitute, and often that is the road traveled. Purpose-built pre-processing programs are available too, such as converting TGA files into a Playstation2 friendly format, or something along those lines.

If you or your company tends to build one type of genre, then these tools tend to evolve from project to project rather than get rebuilt from scratch. If you switch genres, well, obviously those tools that produce high resolution models with per-polygon hit capability aren't necessarily required for an RTS style game.

Gil Gribb, Technology Lead at Raven Software has this to say on the issue of 'off-the-shelf' versus 'build-it-yourself':

"Homegrown tools have the advantage that they can be customized for the needs of your shop, you have the code to fix any bugs or add any enhancements.

The disadvantage of in-house tools is that they are far more expensive to build and maintain, often costing more than off-the-shelf tools. In many cases it is utterly impossible to build your own tools because of the scope of the application, for example 3D modeling and animation packages or bitmap editors."

Of course if you want gamers to be able to modify your game, and you build all your tools yourself, then you pretty much have to release them to the world. This can cause a bit of head scratching; bearing in mind part of the reason to build your own tools is so you have a leap on your competitors. Sometimes it might even benefit you to release source code to these tools, which really gives away the approach used to create content. Again, Gil Gribb on the subject:

"I am in favor of releasing almost all source code. I don't think we have anything to fear from our competitors; legitimate business wouldn't dream of stealing intellectual property. The fans, amateur game makers, and the popularity of the game can all benefit from released source code."

OK, up to this point in our Game Engine Anatomy series, we've covered a lot of topics specifically related to the engines of course, but let's get on with some of the game-specific parts.

Controlling mechanisms can make a huge difference to the game under development, sometimes even so far as to dictate the style or genre of game you are building.

Try playing an RTS with a gamepad sometime--it's just not fun. Sometimes when you are limited to a specific input device, such as a mouse and keyboard, inventing new control methods for your game can be an exhausting process. When Raven started out developing *Heretic II* one of the first things they decided to do was to try and figure out an intuitive method for using the third-person camera with a mouse. Previous to this, most games had adopted the *Tomb Raider*-style of camera (third-person predictive chase), and they found this often just didn't work correctly, and could in many cases lead to frustration on the player's part. The camera would often find arbitrary view angles, move if possible, and sometimes change the orientation of the player.

Given that their target audience was the FPS crowd, Raven needed to find a way to control Corvus in such a way as to be instinctive to FPS gamers. They did it, but it did take a bit of time, and some different approaches--should they keep the camera fixed with one orientation, or have it floating? Most game development efforts--unless a no-frills implementation of an established genre--tend to take a bit of R&D to find the most intuitive melding of physical control device and in-game control mechanism the game requires. Here's a hint--once you find one that works, Stick With It. Messing around with the way to control something inherent to the game can totally alter view, perception, and even focus of the game into something you never intended. Find something that works, prove that it works, then leave it alone. Over-engineering controls can lead to feature creep and perceived problems with the game concept.

A good example of such feature creep is seen in *Independence War*. This game has so many modes, keys, and so on, that it's impossible to just get in and play.

The key here is definitely simplicity. A good rule of thumb is that if, in normal game play, your game requires

more keys to play than you have on the average gamepad, or fingers on your hands, then something needs to be reworked. Note, I am not saying a game shouldn't be flexible-- certainly *Soldier of Fortune* has many possible key settings. But generally, the Quake engine has a good approach when you consider that most of them aren't actually required. Yes, you can select what weapon you want to use, but *you don't have to*. The game will do that automatically for you. This is the difference between flexibility and over-engineering. If the game required you to pick a weapon with a key press, then there would be a problem. You get the point?

Control mechanisms can't be overvalued--often a game will stand or fall based on how much control the player feels they have over events or the principle character. If control is changed, re-oriented, or just plain removed from them, it can lead to a lack of involvement with the game itself, and that, needless to say, is a **Bad Thing**. Spend time on this stuff, but keep it simple and it will help tremendously.

Now we get to the sticky part of the engine, the part that is least-defined. Here's where a game can get extremely buggy, time-consuming when being run, or just plain limiting.

What we are talking about here is the "game" part of the game engine. This is the part that uses all the other technology to get something up on the screen, move it around, have it react to you and also give you something to react to. There are many approaches to this system, but for now I'll stick with the Quake approach because that's what I'm most familiar with.

Let's start with Entities. These can be defined as 'game objects'. Now that doesn't just mean the models that you see on screen, although entities certainly control these-- entities can be other things as well. Basically it's anything the game needs to be aware of at any given time, such as timers for things going on, collision boxes for models, special effects, models, the player, and so on.

Even cameras can be (and are in the case of almost all Raven products) entities. Cameras are assigned an origin in the world with angles, and they're updated every frame and tell the renderer this is where it should be getting it's view data from, and off we go. Entities are typically what get saved and loaded in order to return a game to a previous state. Generally the method used during the load process is to load the game map up, as though you were starting a level afresh, then load up all the saved entities so they return to the state they were in when the game saved. Voila, instant return to a saved game. It isn't quite as easy as that *cough* as I learned the hard way with *Heretic II* *cough*– load/saves gave me more problems than almost anything else, particularly in co-op mode.

Cameras have many forms:

- *Free-form*: where the camera can go anywhere.
- *Scripted*: where the camera has a set path to follow.
- *Game time*: where the camera has a defined behavior that must be followed.

It isn't enough to just say, "Well, I'll just follow the main character". This means that you might also need to stick the camera through walls, or have it move in ways such as to cause even the hardest of stomachs some queasiness. Making it follow some defined up and down motion has its benefits too, as anyone that played *Descent* for more than an hour can tell you. The body and brain are used to having up and down as a static variable, and when it's not, they don't like it. Mike Abrash who worked on *Quake 1*, once told me that even when it was defined, he still got motion sickness from the game they were working on. He mentioned that *Quake 1*, for him, was a year of leaving the building and waiting for his stomach to settle down. Ahh, the sacrifices we make.

Another part of the game module is the weapons system. Most games have weapons systems or something analogous. This is the stuff that affects other objects in the world, and makes them react to given situations, - like getting shot for instance. Usually weapons systems are comprised of many different types; hit scan, missile

based, and radius form.

Hit scan are instant hit weapons. The effect generated for them on screen is just that, an effect. It has no bearing on the actual operation of the weapon when it is used. When you fire the pistol, the bullet is deemed to have traveled across the world instantly and hits anyone/thing in its path instantly.

Missile based weapons have an actual projectile that takes a finite amount of time to travel in the world, giving the opposition some time to get out of the way.

Radius based weapons are stuff like grenades and explosions, which don't have to hit you to affect you; you just have to be in the blast radius. Players caught in that blast radius take what's called *splash damage*. Lava is another form of radius-based weaponry.

So how do you determine what's been hit and what hasn't? Well that brings us on to traces, which we'll touch on more in the upcoming physics and AI segment. This is a set of routines which tells the game what it's hit when given a straight line in the world from point A to point B, for instance from the end of a gun to a predefined distance away. Traces are great, but expensive, since they have to "collision check" all the polys along that line to see if any where hit, not to mention models and other objects. This is also how some physics work, by doing a trace straight down from a given character to tell where the floor is located. Rampant trace abuse--i.e. the use of them multiple times in one game frame—has been responsible for many a slowdown in games out there today. With *Jedi Knight II: Outcast* they've had problems with this for the light saber battles, since they do multiple traces for the light sabers because they need to know not only whether the light saber hit anything where it was, and where it is now, but for all the points in between.

Well, we reach the end of another segment, and only two more to go. Next up will be more details on AI and navigation.

Game Engine Anatomy 101, Part X

July 11, 2002

By Jake Simpson

With nine other segments of our Game Engine story behind us, let's dive in to very interesting and important subject of artificial intelligence. AI is becoming one of the most talked about areas of game development next to the rendering capabilities of engines these days, and rightly so. Up until about two and a half years ago, games seemed to be mainly about how many polygons you could push out, how pretty the eye candy was, and… OK… how bouncy Lara's chest was... Now that we've gotten to the point where we can render out very realistic mammaries, the focus is shifting to what we actually do with those polygons (i.e. game play). AI is critical in this area, because it provides you with the stimulation to actually play and be involved with what's going on in the gaming world.

AI covers a large gamut ranging from deciding what new brick to drop down in *Tetris* (which is pretty much just a random number generator), all the way up to creating squad-based tactical games, which interact with you the player, and actually learn from you as you play. AI encompasses many disciplines and is the area that will come back to bite you in the arse if you (as a game developer) don't spend enough time getting it right. So let's talk about a couple of those disciplines shall we? That way you can get a better idea of exactly how complex AI systems can get. We'll use a mythical game as our example rather than a live one, to avoid legal entanglements

Imagine we've got this game with bad guys trying to live in a 3D world, go about their business, and react to you (the player) if you disturb their normal routines. The first thing you have to decide is what exactly is the business they are going about anyway? Are they guarding something? Patrolling? Planning a party? Doing the grocery shopping? Making the bed? Establishing a baseline of behavior is job one for the game developer. Once you have that, you always have something that NPC (Non Player Characters), or computer-controlled 'people', can

revert to doing should the player's interaction with them be completed.

Once we know what an NPC character needs to be doing – let's say it's guarding a door, and doing a small patrol of the area, the NPC must also have 'world awareness'. The game designer needs to determine how the NPC's AI is going to see the world, and the extent of its knowledge. Are you just going to say, "Well the computer knows everything that's going on"? This is generally regarded as a Bad Thing, because it becomes obvious pretty soon that the computer can see and hear things that you can't, and it's seen as cheating. Not a fun experience. Or are you going to simulate his Field of View, so he can only react to that which he can see? Issues arise here when walls get in the way, because you start getting into those 'trace' routines I mentioned in Part IX, to see if the NPC is trying to react to someone that's obstructed by a wall. This is an obvious AI problem, but it gets even more complicated when doors and windows get involved.

When you start adding aural awareness into the AI stimulus routines, it gets more complex still. But, this awareness is one of those key "little things" that make an imaginary game world seem that much more realistic, or can dowse the suspension of disbelief. Raise your hand if this has happened to you: you engage an NPC in a gunfight, and having dispensed with one NPC, you walk just around a corner and encounter another who's still in his default behavior pattern, unaware of what has just transpired. Now, guns are noisy things, and the gunplay would have obviously alerted a "hearing" NPC that something was afoot. The trick in pulling something like this off is to find an efficient way to determine whether a sound emitter (i.e. your weapon discharging) is close enough for an NPC to hear.

Next are the decision-making routines. What behaviors do you try to implement when our patrolling NPC character can hear something but not see it? Does he go looking for it? Ignore it? How do you determine what are important sounds he should be investigating or not? It can get very complicated very quickly, as you can see. There are many methods to building code that deal with this stuff, but generally it's a good idea to come up with a system that is not NPC-specific, but will work for all NPCs based on attributes you can build in text files outside of the game engine. That way it doesn't require a programmer to change the AI for any given character, and if you do make changes in the game code, it automatically will get applied to all characters instantly, which in most cases is a Good Thing.

Other world awareness issues can crop up, such as the situation where two guards are standing next to one another, and you pick one off using a sniping weapon, and the other stands there completely oblivious to what happened. Again, nailing down the little facets of real-world behavior is the difference between a good game and a great game.

But let's say you have that stimulus-response section all squared away – you've scanned the world, decided there is something going on that the NPC should react to - he's heard the player character make a sound - and you (the developer) decided what he should do about it – he's going to investigate. Now comes more complex stuff. How does he get from where he is now, to where he thinks the sound is coming from, without running into walls, bumping into furniture, and generally looking like a digital dolt? Read on…

Fast, accurate world navigation (also called path-finding) has become the Holy Grail for game developers lately. Making it look convincingly human is very much a hard thing to do. You need knowledge of the local world geography - where the walls are located, the steps, the edges of cliffs and buildings, etc. You also require knowledge of objects in the world – like furniture, cars, and most especially where other people are located. And it's really the last element that's the problem, but we'll come back to this in a moment.

World navigation is generally broken up into two areas, world navigation and local navigation. The distinction between the two is really just one of scope, but most programmers address it separately, since it's easier to handle that way. World navigation routines handle understanding rooms, doors, and general geography, and will work out a way to get the player or character from point A to point B in the world. Now that's a real nice little statement to make isn't it? "It will get you from point A to point B". Easy to say, but hard to do. Understanding the world is a very complex problem, and I've seen many solutions attempted. *Quake III*'s bots follow a pre-

processed map that's built, in very general terms, using the original map's floors. The pre-processor detects the floor pieces, which are flagged as such by the map builder, and builds itself a simplified map of the world with just the floors. The bots don't really care about walls because they never approach them, as they follow the map of the floor, which has wall avoidance built into it by design.

Other approaches are to build small nodes into the map itself, which the AI can follow. These nodes are usually built to be within the line-of-sight of each other, and with a link from one node to all the other nodes, the character AI can 'see' directly, so you are guaranteed that the AI won't try to go through walls when traveling from one node to another. If doors or drops are in the way, you can use these nodes to encode information about the route ahead, so the NPC can adopt the appropriate behavior – wait for a lift, open a door, or jump from one point to another. This is in fact the system used by Heretic II, and one that Raven adapted for use in most of their other games.

Here's what Jess Crable of 3D Realms, currently working on Duke Nukem Forever, has to say on the subject:

> *"Navigation is a big challenge in many ways, mainly when there's a lot going on in the game and something unplanned gets in the way, like an obstruction. The AI needs to be well aware of what's going on around it in order to avoid and or realistically navigate around unplanned obstructions (like another AI for example). Another comparatively large challenge is realism. If the AI is representing something that players have seen in real life- like a human, or a dog for example, then it's all the more difficult to make it appear truly believable."*

Then there is local navigation. We may have a route to get our NPC from where he is in the world, to where he thinks he heard the noise, but you can't just follow this blindly and expect great looking results. Paths of this nature tend to be very specific for a given purpose. When you are running through corridors from one room to another it's OK, but if you are trying to guide him around a large room, the path node approach tends to end up with some weird looking path-finding. These routes are also not dynamic. Because they are pre-built, they don't tend to take into account any dynamic changes in the world. Tables may have been moved, chairs destroyed, walls get blasted in, and of course, people move. This is where local navigation differs from world navigation. It has to take into account the local world and navigate the NPC through it. It has to know what's around, what paths exist that it can take, and decide which one to take.

The biggest problem in local navigation is other NPCs. Given a specific routine for finding a path, if you have more than one NPC in the same general area all trying to get to the same point in the world, they all tend to end up with pretty much the same path. Then they try to take it, end up bumping into each other, then spend all their time trying to disentangle themselves, and once they've done that, they try getting to the objective again, and we see the same thing happen again. It all looks very silly and very "Keystone Cops-like", which is hardly the effect most people would desire. So obviously some variance in the pathing is required to avoid this scenario, and some code to cope with avoidance is required. There are plenty of flocking algorithms that can help here.

Of course somewhere in all this you have to decide exactly what animations you want the character to be playing while he navigates himself around the world. Sounds trivial? It isn't. Raven's Chris Reed – current custodian of the AI system named LICH that *Soldier of Fortune II* is using – has this to say on the subject.

> *"I can tell you that at the moment, we are having the most difficulty with smooth movement. Trying to make five characters walk around each other in a hilly grassy jungle is a very hard problem. Making the low-level systems perfect is important, because unless the characters look realistic on a lower level (avoiding walls, proper animation), they are not able to effectively convey the intelligence of any higher-level decisions. For that reason alone, the animation and low-level movement is the most important and the hardest to do. It does need to be perfect."*

So we've got our guy from point A to point B, he's navigated himself through the world, avoided obstacles, animated correctly, on the way and is now here. He sees you. What next? More decision making obviously. He's

going to shoot at you. Great. You shoot back. Now what? Now you go through the same stuff all over again when he tries to run away.

You see the wealth of issues you have to deal with here to make this situation look convincing. It can be compounded if you set up your AI to use behaviors that you simply don't have animations for the NPC to perform. An example of this was some of the AI in *Soldier of Fortune*. They got railed on because the bad guys didn't react to stimulus in appropriate fashion. Enemy NPCs didn't strafe, or simply didn't run away when evidently they should have. Part of the problem was that they simply didn't have the animations to strafe the enemy NPCs, or have them run backwards, due to space issues. So all the greatest AI code in the world wouldn't have fixed that. This is all important stuff to consider.

Wanna know the real kicker though? Take all that I have described above, then try applying it to a squad of NPCs that all have to talk to each other, set goals, and communicate with each other, while not getting in each other's way. And once you've done that, try taking *that* code and making teammates for the player that do all that is described above, yet don't get in his way in a firefight. Now that's complicated. And then make it fun. And that's the hardest part of all. Raven's Chris Reed had some comments on AI 'feel':

> *"I think feedback is a huge issue with AI. The realism of the game is completely broken if a character simply does not react to a change around him. There are plenty of obvious examples of this (hearing gunfire, seeing a buddy get shot...), as well as some that are more subtle (two people look at each other and nod as they pass in a hall way). Players are willing to accept some stiffness and predictability, but these kinds of things tend to bring the game to life."*

And Jess Crable agrees:

> *"Balance is hugely important and… crucial to how much fun the player's going to have, but there are other issues to balance as well. Players often say that they want to see more realistic AI in games. Too much realism, however, begins to take away from the fun. There has to be good balance between the two. Variety and randomness are important as well- variety in behaviors, and some level of unpredictability within the realm of staying believable."*

In all of our descriptions of AI, we've taken a very FPS approach. There is more than one kind of AI. What we've been describing is a set of rules for coping with a 3D world. AI is much more than that. Often the best AI is in fact very simple. It's a set of rules for responding (or initiating) actions that the player must respond to and handle.

There is a buzzword called "emergent game play" that should be dealt with here. Emergent game play is essentially the creating of rules the game will adhere to, that will result in situations that the game programmer couldn't foresee.

For example, Chess can be considered emergent game play. There are set rules, but the game can get into all sorts of situations that the programmer can't deal with on an individual basis. You can't have a coded rule for every possible board play scenario. Clearly, the player will not always be faced with the same game every time. To a certain extent, the kind of game being played will change depending on his actions. *Black and White* is a perfect example of this situation, as is *The Sims* – the game has its rules, but how you use them and blend them together is up to you. In effect, you are creating the game as you go along, rather than following set routes the game designer / programmer has defined for you.

It is possible to blend the rules-based, emergent game play approach with an FPS environment. *Half Life* did this with some of its Marine behavior – suppressing fire and flanking was done on the fly from set rules. It looked dynamic, and to a certain extent it was. However, having just a set of rules in an FPS world often isn't enough. Geometry and other AI can often defeat simple rules, which makes it all a lot more difficult to get right and still be fun. So have some sympathy for those poor AI programmers out there. Their job isn't easy.

Well, that's another segment down, and just one more to go! In the last section, coming soon, we'll wrap things up by taking about heads-up displays, menuing systems, game customization and configuration, game engine licensing versus building, and finally game "modding". Stay tuned.

Game Engine Anatomy 101, Part XI

July 26, 2002

By Jake Simpson

You've seen menuing systems, and you probably understand that in-game heads-up displays (HUDs) are often a much ignored and maligned part of the gaming experience. Recently, attention has been drawn to this area by the very impressive *Black and White*, which doesn't really have a HUD. After Peter Molyneux's experiences with *Dungeon Keeper*, and its myriad of icons on screen, he decided that too much of the game was being played through these icons, and not enough was being played on the main screen. So he decided to do away with all of them. Bold move Peter, and we applaud you for it. Unfortunately while this approach could be made to work for the style of gaming that B&W uses, it's not always practical for other genres.

HUDs by and large should be unobtrusively, providing ONLY the critical info you need; which just by itself can lead to arguments among the design team. The original design of *Soldier of Fortune* had an icon on screen that showed you exactly where you had been hit on the body when shot. Eventually this was discarded when they decided that they weren't going to penalize the player for different body part damage. On some of the earlier screen shots of *Soldier of Fortune* this icon can still be seen on the top right of the screen.

In a perfect world the HUD would be configurable, so you can decide what is displayed, where, and how often. If you don't feel the need for a local radar, then it should be removable. Any HUD info displayed should have a degree of alpha (transparency), so you can see through them if need be.

Speaking of configuring stuff, I'm a great fan of personal customization for gaming. It's not as widely available on console games because of the lack of instant storage devices to store config files, which is fair enough. But with the coming along of the hard drives for the PlayStation 2 and the Xbox, I expect to see this used more in the future. Everything that can be customizable should be, as I see it. Obviously, sensible defaults should be provided for everything too, so the player isn't forced to wade through screen after screen of tedious selection processes -- and we'll touch more on this in a minute -- but pretty much the player should be able to customize the gaming experience to individual taste and available computing power.

Getting back to that defaults thing, it's definitely important to keep the required modifications to a minimum. Getting into a game fast and quickly with a minimum of decisions is always a Good Thing. *Mortal Kombat*, even *Quake III* has a very fast game entry system. A couple of choices and you're in. That doesn't mean you can't have menu after menu allowing you to change everything, but they shouldn't be required and should have sensible defaults already in place. If you have to build up a character using 14 screens before you even get into the game, chances are that first time through you may not have a clue about what you're selecting and will just do anything to get through the screens, but may well have done something that will significantly affect the initial game play experience. And more likely than not it will be an adverse affect, and as a games programmer/designer, I am here to tell you that no matter what you do, you will have enough opportunities to make a bad first impression without making it worse by allowing the player to select something stupid the first time around.

With that brief discourse on configuration and HUDs (along with much info in many of the other ten story segments), we've finally reached the end of our discussion about the main building blocks of a current game engine. Of course each particular game has its own additions (or subtractions) to the list, depending on what the game is, and who's making it. However, there are a couple of other elements to game engines that aren't really part of the engine design, but they nonetheless require some attention.

These days if you want to make a game, often the fastest way to get started is by licensing an existing game engine and going from there -- that's what Raven does, most recently with *Star Trek Elite Force* written using the Quake 3 engine. *Half Life* was based on the Quake 1 engine, *Deus Ex* was based on *Unreal*, and the list continues. There are two ways to go with licensing as it stands today -- a complete game engine (or Game Operating System as Jason Hall dubs LithTech), or a partial solution for a given set of issues. A good example of this would be RenderWare, which is a partial solution for rendering a scene and giving you some physics to play around with too. You couldn't just slap in some models and call it a done game though -- there are sound systems required, game mechanics, and so on. But it does give you a firm foundation to build on. Remember all the math I mentioned in the rendering and physics sections? Well, this way you avoid all that.

With LithTech, *Unreal* and *Quake* you do get a complete solution -- or at least as complete a solution as the originator requires for their games. Remember *Quake III* is multiplayer, and is NOT built on the foundation of a single player game as *Unreal Tournament*. With *Quake III* you lose certain systems that a single player game requires, like Load/Save, scripting and so on. You are only getting what id required to make a game, not necessarily what you'd need. Sometimes this can be a real drawback if the one limitation of the system is exactly what you need. Putting scripting and load/save into *Star Trek Voyager : Elite Force* was no picnic, but necessary. However, using the Quake 3 engine was still one heck of a head start.

Tim Sweeney of *Unreal* fame has some observations on some of the current pre-packaged game engine solutions around today.

> *"I think I can fairly compare the game engines (Quake, Unreal, etc) to the game components like RenderWare and Karma. Game engines are top-down frameworks that encompass all technical aspects of game development: rendering, editing tools, physics, AI, networking, and so on.*
>
> *They are aimed at developers who want a complete, off-the-shelf solution so that they can focus on their gameplay and content. Game components like RenderWare are aimed at developers who are developing their own technology, but don't want to reinvent the wheel in some area of technology that's already well-defined.*
>
> *Game engines have the advantage of solving the complete set of technical problems in game development, with the drawback that they tend to include assumptions about game genres built into them. For example, Unreal has been used for first-person shooters, third person action games, role-playing games, and even a pinball game. But nobody has used it for a flight sim-- it's not an appropriate technology for that kind of game. Game engines come with full source, which is a blessing (you can see exactly what's happening internally, and you're free to extend it however you want), and a curse (if you change it, you'll have to merge changes into new versions).*
>
> *Game components have the advantage of solving problems in a focused area of technology, such as rendering or physics, and doing so better than the typical developer could do without spending a huge amount of time on it. Their drawback is that it's up to you to get their component talking to the rest of your engine, which can sometimes be quite complex. Game components tend to come without complete source, so it's not always clear what they do internally."*

Thanks Tim, a masterful analysis.

Rather than licensing, you could just say "The heck with it" and build your own engine. This avoids all the legal entanglements of who owns what, royalties, etc., and if you produce something of sufficient quality, you can even go out there and license it to other people. However, as has been pointed out, this takes time and money to do, not to mention damn good programmers. LithTech has been evolving over the years, similar to *Unreal*. Interestingly enough, the initial iteration of *Unreal* actually took four years to complete, mainly because of moving hardware and API versions. When they started, software rendering was the only game in town. While development was going on, 3dfx came along with Glide and then Nvidia with their TNT card (and certainly

there's been much more advancement in hardware and APIs since then). That's why it supports so many different rendering paths. Of course supporting them all from within the same engine was a bit of a coding nightmare, but that's another story. Each engine has a modular approach, and when one module -- say, scripting -- becomes outdated or requirements change, you just yank it out and start a new one.

The *Quake* engines tend to have more complete evolutions over time. Every iteration of the engine goes through complete rewrites while John Carmack creates something that runs fastest on the current hardware corresponding to the set of requirements for id's next game. *Quake II* was completely rebuilt no less than four times, and I personally saw three different iterations of the bot code for *Quake III*. Other Developers aren't immune to this situation either. John Scott, who built the terrain system for *Soldier of Fortune II*, mentioned to me that he tried many approaches to get physics to work correctly on the dynamically generated terrain.

Building either the technology or a complete engine is no trivial matter. There are many, many systems required for present-day game engines, from the simple display of text on the screen to the high-level AI, as many people have found when attempting to create the 'next big engine'. And as I mentioned above, constantly evolving new technology makes building a fast, efficient engine a moving target. As it is, I've seen it take people four days of messing around with the PlayStation 2's blend modes just to get a texture with alpha to show up correctly.

Other engines worth considering are the *Tribes 2* engine made available by the Garage Games people -- it's called The Torque Game Engine. My understanding is that this can be licensed for very nominal fee, with some royalty deals in the future. This is definitely worth thinking about. You can see details of the engine features here. And then there's the *Serious Sam* engine. That's out there for licensing too, and is definitely worth a look. If you are interested in that, talk to God Games-- they should be able to point you in the right direction.

There are some free engines on the web you can down load -- the first that comes to mind is the Crystal Space engine. You can download this from here and use it in your games as you wish. It's not a professional engine, but it is often a good learning experience to see how all the bits fit together.

Then there is the original Quake Engine, which has now been open-sourced by id. This is a terrific start for any aspiring games programmer -- just download it, compile, and start tinkering. It's worth remembering though that this engine is many years old, and doesn't bear that much resemblance to *Quake III* or the new Doom. But again, it's a good place to start. You can find a good resource page here.

Really, it's all about money versus time. If you haven't the time to develop a new engine, but don't mind spending the money to use a third-party engine, then go for it. Note that most engine licensing groups these days have very reasonable approaches for teams asking to use their engines. It's in their interests to get as many people using their tech as they can, so experience with it becomes an industry standard.

A glance at any online gaming server statistics reveals that there are more *CounterStrike* servers out there than anything else. There are almost twice as many CS servers out there as its nearest competitor (either *Quake III* or *Unreal Tournament* or any given day).

Game mods all came about from the editing programs that enabled gamers to modify the original .WAD files for DOOM, supplying their own home-brewed level designs and textures. People started playing with these (mostly) home-built tools and found they too could produce levels that other people wanted to play. id noticed this trend and took it a stage further with the Quake series of engines, designing the game so this it was eminently user modifiable. They even went so far as to release their own design tools, instructions, and even -- *gasp* -- some of the code from the game, so aspiring game programmers could play in the Quake Universe. From this it was possible to create your own version of the online experience that was Quake. Many of today's industry gurus came from this early modification experience. Now-famous designers like LevelLord and CliffyB got their start in the industry this way. The crowning glory came from one gentleman named ZOID who produced 3Wave CTF, the first of the 'capture the flag' games, requiring people to play as a team -- one of the first evolutions of

the online experience from the straight deathmatch.

Some games are so popular that they have yearly events occurring. Quake for instance has QuakeCon, a once a year quake convention held in Mesquite Texas, home of id software. People come and bring their PC's for a weekends fragging, or to see the latest mods or Quake engined games on display.

These days any game you produce needs to either have a killer multiplayer experience from day one, or have easily modifiable content so online 'modders' can take your game and make other games out of it. All of this prolongs the life of your game, and hopefully will sell you a few more units as people buy it so they can download the mods to play the latest mod'ed version of *Quake III: The Teachers Strike Back*. But you can't just produce a game, release your tools, and sit back. You do actually have to design the code initially to be easily expandable without requiring coders to be, …well, … John Carmack.

As a developer you need to be out there being visible and providing help and experience to people at home who want to take your game and do something else with it. This support can come in many forms -- a kind word, code snippets, advice, publicity or simply money. It often doesn't matter what the form is as long as it's there.

The choice of which third party tools you use to build content can be critical here. At Raven we've made some development decisions in the past that haven't helped in that area, since we use SoftImage for most of our modeling and all of our animation needs. While it is the best tool to produce the animation we require, it's phenomenally expensive for the home hobbyist. This makes expanding on the content we produce a problem for those at home, so they tend to pass us over and go to games that are easier to produce content. This is definitely something to watch for when building or choosing an engine. In response to the huge interest in doing game mods, Discreet has brought to market a "lite" version of 3D Studio Max called gmax. And best of all, it's free. If you want to play with it, you can grab it here.

Ultimately the success of the online game can often be traced back to the mod community, and so I think it's fair to thank them for a job well done. I've often said in the past that the fastest way to a "real" job in the industry is to start out with a mod, show you have the discipline to finish it and then use that as an interview-getter. Nothing says, "I can do this" like having done it. So get out there and get going. What have you got to lose?

Further reading at JakeWorld! http://www.jakeworld.org

Also here's a very good article by Kenn Hokestra on getting a job in the industry.

The author would like to thank (in no particular order) Tim Sweeney, Jim Dose, Jess Crable, Gil Gribb, Garin Hiebert, and Chris Reed for their words, and will definitely stand his round at the next game developers conference. He would also like to thank Mike Crowns for being an awesome proofreader.

Jake Simpson is a games programmer who's been in this business off and on for about 20 years. He started at age 15 in his native England, in the days of the C64, Sinclair Spectrums and BBC Micros, went through the Amiga and ST, took a bit of time off, then came back to work on Arcade machines for Midway Games back in the mid to late 90's. -He recently worked for Raven Software, home of Soldier of Fortune, Heretic, Hexen, Star Trek : Voyager : Elite force and Jedi Knight II : Outcast, and can now be found at Maxis in Northern California working on titles from Will Wright's stable. In his spare time he codes for the GameBoy Color and Advance, because "it's as far from C++ coding as you can get, besides, as John Carmack said, low level programming is good for a programmers soul".