KAGE: WHAT IT DOESN’T DO

(…and why this is a good thing)

3D game graphics is almost always a balancing act of performance and quality. Raise the quality, and the performance usually suffers. The inverse is also true. Typically, this is due to the polygon counts. More polygons usually mean better-looking graphics, but always means lower frame rates.

Polygon counts usually refer to the number of polygons that are drawn. More precisely, it’s the number of polygons sent to your 3D card (or the software renderer in the game.)

With the newest 3D cards, graphics got better for a few reasons. The cards were designed to do things that were just too slow to do in software (bilinear filtering is a good example) but also, they do it so much faster than software. So today’s 3D engines just keep pumping more and more polygons through the 3D cards, and 3D card manufacturers keep coming up with faster and faster cards. Why? More polygons, of course.

Texture memory is also a consideration. If you want to draw a polygon, you have to make sure that the texture is available to the 3D card. Uploading more textures to your 3D card means even slower frame rates.

OK, stop let’s take a break for a second. I hope that I haven’t said anything you don’t already know. “More polygons” and “faster frame rates” just don’t go together. There is no exception to this fact. ***More polygons always mean slower frame rates.***

So let’s take a trip through a simple rendering pipeline. This is where the engine starts with a list of 3D polygons and ends with a visible image on the screen. Surprisingly, I won’t need to simplify much as there’s not a lot to it. And it’s really important for a good understanding of the problem at hand. So, please bear with me. And who knows? You might even find this a bit interesting. I’m hoping you’ll recognize a few of the terms I use here, and might get a better understanding of what they mean.

Before we get into the guts of an engine, lets first clarify that this is a very naïve engine of the most basic type. The engines in today’s computer games are typically much more complex than this, but the following section should give you a good basis of understanding.

A SIMPLE ENGINE:

In this example, consider a simple level of an office building. You’re in the bathroom and the door is closed. The bathroom is somewhere very near the center of this office building. The whole scene consists of a million polygons (yeah, the level designer went a bit wild with this one… ✍)

1. We’ll start with the list of polygons (let’s use triangles for this example.) Remember that there are a million of them. Three vertices (3D points) per polygon define the size, shape, location, etc. of the polygon in “3-Space” (3D). For this example, these polygons are single-sided, which means that if a polygon is viewed from behind, it shouldn’t be drawn.
2. We’ll need a viewpoint (the location and direction the polygons will be viewed from.)
3. At this point, we’ll want to start building a new list of polygons. We’ve got lots of polygons in the scene, but we’re only interested in the ones we want to draw (for example, we won’t be drawing any polygons behind the viewpoint.) We’ll start with this list full of our original million polygons, and remove what we don’t want.
4. Now it’s time to figure out which polygons are facing the wrong direction. We’ll want to remove any polygons that are facing away from the viewpoint, since those polygons are

not visible. This is typically referred to as “back-face culling.” This tends to cut the count in half. We’re down to 500,000 polygons.

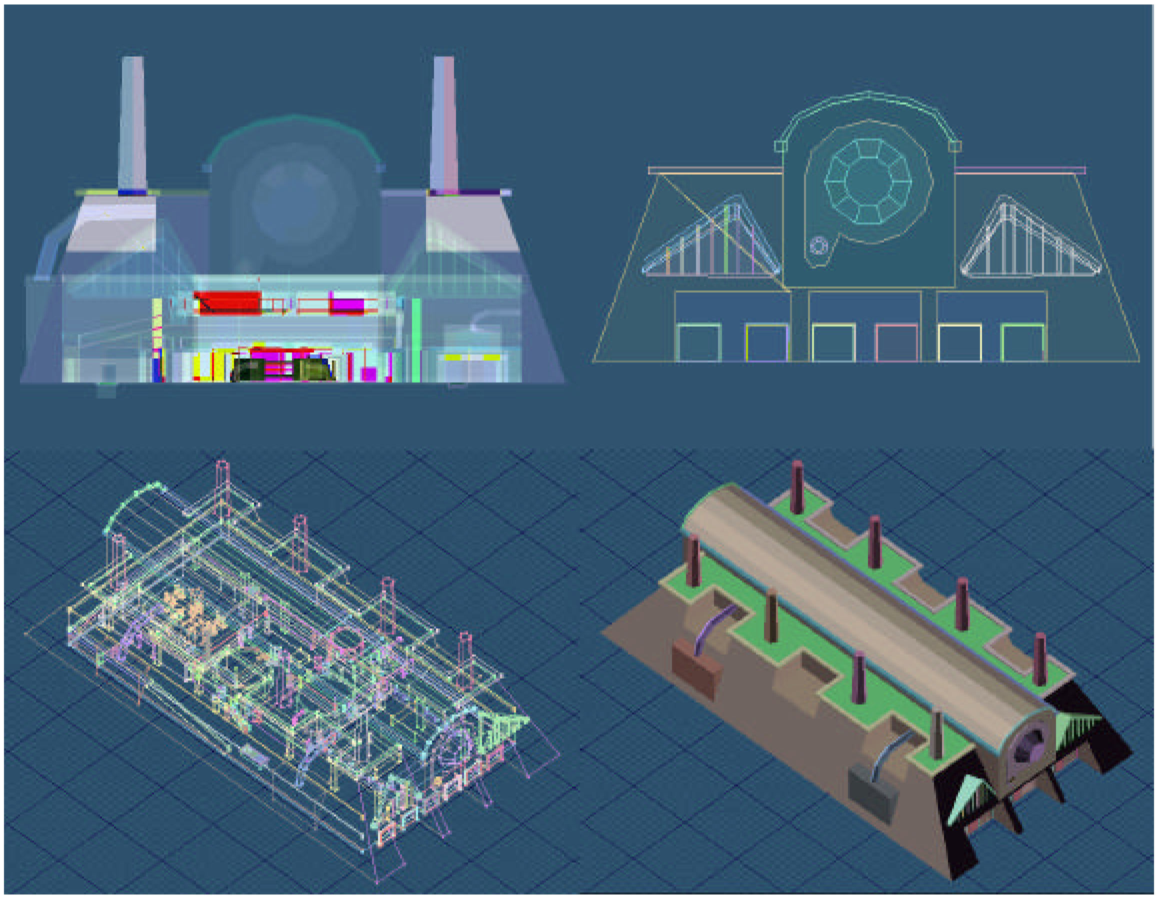
5. Using simple matrix math, we rotate and move the scene into view. If you’re picturing this in your mind, you’ll wonder why anything has to be moved or rotated. The polygons are where they are, and the viewpoint is where it is. But in reality, things work a bit in reverse. It’s a lot like a 2D scroller – the kind where the main character is always in the middle of the screen and everything else scrolls around him/her. So we put the viewpoint at the center of the universe (facing straight ahead) and move/rotate the entire universe until the proper view is obtained. If our player was standing on their head (up side down) then the universe will now be upside down, with the viewpoint right side up. As far as the viewpoint is concerned, however, everything LOOKS right. This process is called “transformation.” And doing so, we’ve just done some pretty simple math on 500,000 polygons. On a fast machine (P2/450) this takes about a 10th of a second. At this point, our maximum frame rate is 10 frames per second.
6. Next we need to do some clipping. The idea is simple. Our viewpoint works just like a camera. The camera has a lens. Different lenses have different viewing angles… like a wide-angle lens versus a zoom lens. In comparison, the viewpoint has an FOV (Field Of View.) Camera lenses and FOVs are pretty much synonymous with each other. We’ll assume our viewpoint has a 90-degree FOV. This means that the viewpoint can see 45-degrees up and 45-degrees down (total of 90 degrees) as well as 45-degrees to the left and right. If you picture this in your mind, you’ll see that this forms a pyramid-like shape. The apex rests on the location of the viewpoint, and the four expanding sides travel away from the viewpoint into space. Clipping is a three-way process. We want to keep everything inside the pyramid, remove everything outside of the pyramid, and SLICE up (clip) all the polygons that cross the pyramid’s boundaries (keeping the interior portion of each polygon.) By the way, this pyramid is referred to as the “view frustum.” This can be a pretty expensive process… and we’ve just done it to our remaining 500,000 polygons. On our P2/450, we’ve just wasted another 10th of a second or more. Our maximum frame rate is now 5 frames per second. But, on the good side, we’ve only got about 83,000 polygons left.
7. What’s left (83,000 polygons) are the “visible polygons.” We’ll be drawing these polygons. So far, our list still contains 3D polygons. We need to “project” them (sometimes referred to as “scaling”) to the 2D screen so that the polygons near the viewpoint appear larger than the ones farther away. This gives you that 3-dimensional look, even though it’s displayed on a 2-dimensional surface (the screen.)
8. We take our list of polygons and send them to the 3D card. We’ll be piping 83,000 polygons to the card, so this’ll take about a second or so, plus the time required for the card to actually draw them, not to mention uploading all the textures. Our frame rate just dropped to about 3 seconds per frame (with enough textures, even a minute per frame.)
9. That’s it… go back to step one, we’re already about 2.9 seconds overdue for the next frame!

If (in our office example) you had your nose to one of the bathroom walls (maybe there was a scratch-and-sniff there or something… ✍) you’re LOOKING at one or two polygons, but we DREW 83,000 of them. This is an extreme example, and I hope the problem has shown itself to you very clearly. But in case you’re still not convinced… let me point out that the problem is actually worse than this. The polygon count I mentioned (83,000) was derived based on the law of averages from a viewpoint in the CENTER of the office building. If you were to place that viewpoint in a corner, looking toward the center, that number could easily become 400,000 to 500,000 polygons sent to your 3D card!

Granted, games these days don’t typically have a million polygons. All games are different, but as a reference, a typical Quake level might have about 10,000 to 20,000 polygons on average.

A VISUAL EXAMPLE:

In the following image, we're looking at a scene of only 5,700 polygons. The upper-left view represents a "complexity" rendering that shows all of the non-visible polygons that have been drawn over and over (in increasing brightness.) The highlighted portions express extreme overdraw.



The upper-right view shows a view with little or no overdraw. The two bottom views were included for reference.

BSP-TREES:

So, how do games solve this problem? Very commonly, they use a Binary Space Partitioning Tree (BSP tree.) I won't bother you with the details of how these things work, but here comes the basic idea….

A BSP tree splits up the entire scene into areas (called "convex spaces".) These are just small areas of 3D space (the small bathroom in our example may be made up of 5 or 6 of these areas.) Once the entire scene is split up into these areas, another program comes along and pre-determines what is visible from each area. It's a smart program (we'll call it "VIS") but it takes a long time to run. It's certainly not impossible to determine what's visible it's just very, very slow. So it takes its time, and makes a list of the visible polygons that it found in each area. This way, when it's time for the engine to do its thing, it doesn't start with a list of all the polygons in the entire scene, it just figures out which area the viewpoint is in, and uses that list of pre-determined, visible polygons.

These pre-determined lists of polygons are called a PVS (Potentially Visible Set.) The key word here is "Potential" since the lists contain more polygons than are actually visible. In graphics terms, this is called "conservative visibility" since it refers the fact that it is very conservative in the way that it returns more polygons than are truly visible so that it takes no chances in NOT drawing a polygon that should be drawn.

There are quite a few drawbacks to using BSP trees that most people aren't aware of. There's quite a few, so let's get started. Before I begin, let me first point out that this information is taken from existing technology used in games today. But every game works differently, some games have solutions to some of the problems I'm about to discuss, other games have partial solutions (ways to make things work a little better or take a little less time.)

It would be interesting to point out that on a typical PC, the VIS program would take months (even a year depending on circumstances) to work out the solution to a million polygons.

Before I can tackle the next point, I need to clarify one issue. As stated earlier, VIS tries to determine exactly which polygons are visible from within each area (so it can build the list of visible polygons for that area.) Note that this means that the list of visible polygons for any area refers to a list of polygons that can be seen from any point within that area. Different locations within each area will result in different visible polygons. So VIS spits out a list of ALL polygons that were visible from ALL points within the area. This, at times, doubles the number of polygons for an area.

In the same way that there are an infinite number of fractional numbers between 0 and 1, there are also an infinite number of 3D points within each of these areas. So rather than spending an infinite amount of time on the task, VIS tries to estimate the result. This estimation may result in a few extra polygons, but more importantly, the estimation may be MISSING a few polygons that are really visible (this is the opposite of "conservative visibility.") If this happens, the rendered scene is wrong (incomplete.) So how do you fix this? Well, when it comes time to render the list of polygons for an area, you don't just render those polygons, you render the polygons from the lists of all the neighboring areas, too. That pretty much covers it, but you've just potentially quadrupled your polygon count.

Let's talk about those extreme cases. A game that runs at good frame rate *most of the time* is not acceptable. Playing a game that cannot maintain a consistent frame rate can really distract from the game's playability. So solving these extreme cases sometimes ends up as a limitation on the level designers to avoid these cases. This means that in many cases, the level designers can't always give you what they want to. Which distracts from the full gaming experience, but in a more fundamental way.

Getting back to our discussion of BSP trees, let's talk about doorways (openings in an architectural environment that inevitably modify which polygons are visible.) If the door is closed in the bathroom when VIS gets around to it, then the VIS program will not "see" the polygons that make up the adjoining room. So all doors must be left open during the VIS process (and all those extra polygons are added.) Otherwise, if you open the door from within the bathroom, you would not see anything beyond it. And, if you're in the bathroom with the door closed, you can't see those polygons making up the adjoining room, so they should not be drawn. Yet they will be, and the polygon count rises even more. Fortunately, the level designers are aware of this fact, and they purposely build windy hallways outside of rooms so that the only extra polygons are the few that make up a hallway. Again, we're placing more limitations on the level designers.

Because of the way the scene is rendered with a BSP tree, there's no easy way to add in other geometry (like players running about.) So the trick is to use a z-buffer. For hardware 3D, this is fine, but for software, it really slows things down.

Since VIS does all of its work prior to shipping the product, the scene is static. Nothing can move. If you move something, then the lists of visible polygons for the different areas all become potentially invalid and VIS needs to be run all over again. So the scene may not change one iota once VIS has been run.

The scene can't change, but you can move actors around the scene. And you can add in doors and that sort of thing, but none of which will affect the rendered polygon count. So if a character runs right up to you, blocking your view of a large portion of the environment, you're still drawing that environment behind him, just to draw right on top of it again when it comes time to draw the polygons that make up the character. I've mentioned this before, it is called "overdraw" *and it's the frame-rates* ***worst enemy****.*

Another major fallback to BSPs is the fact that by their very nature, they tend to slice up the polygons (turning one polygon into multiple pieces) that make up the scene. Modern day games have reduced the number of polygons that get split up to about 5 to 10% or less. But a game that uses a "splitting BSP" is a very common thing. This of course, simply increases the polygon count yet again.

BSPs (when used in conjunction with a PVS) requires a completely closed database. If the database is not closed, rendering errors occur. This means that truly "outdoor" scenes cannot be accomplished, rather creative level designers must simulate them. This is yet another limitation placed on the level designers.

I don't want to sound too negative on BSP trees. They do have their place, and there are very valid reasons for using BSP trees, and when you need a BSP tree, it can be a real life saver.

QUICK NOTE ON PORTALS:

Portals are a totally different technique for accomplishing visibility. But they suffer from their own set of drawbacks. They are also limited to static environments (though, there is a little more freedom here.) They also place their own set of limitations on the level designers, and are completely incapable of dealing with outside environments.

OTHER TECHNIQUES:

There are many other techniques, each of which comes with a plethora of advantages and disadvantages. These are primarily tradeoffs: you give something up to gain something else. This includes the newer hybrid systems that mix portals with BSP trees.

THE KAGE RVS:

Finally, we get to the visibility scheme used by KAGE: the RVS (Real-time Visible Set.)

The RVS solves all the past visibility problems and it does so by determining what's visible *when it's time to render the scene*.

In our bathroom example, the RVS wouldn't do what the typical engine would do. It wouldn't start with a full list of polygons for the entire scene (or even a partial list like a PVS system would) it would start by drawing the visible polygons nearest the viewpoint, and work its way outward from there, stopping when the scene has been fully rendered. This means, that if the "visible scene" consisted of a single polygon, it wouldn't have to perform any of the other work (transformation, clipping, projection, etc.) on any of the remaining million polygons.

Consider the previous paragraph. The RVS doesn't ever touch those polygons beyond what's visible. This being the case, does it matter how many there are? In reality, the number of polygons for the scene might be a billion polygons (or even a trillion!) but that wouldn't affect the frame rate. KAGE only renders what's visible, regardless of how many polygons are out there.

This leads itself into a whole new area… since we don't need ALL of the polygons (just the visible ones) we can just leave them on the hard disk until we DO need them. This allows us to have a game take place on a single, continuous environment rather than pausing to load individual levels. This can cause jumpy frame rates (pulling polygons off of disk just in time to render them) so we do buffer them up a bit with some prediction logic, but the concept doesn't change.

KAGE is also not limited to indoor-only scenes. Nor is it limited to static scenes. As stated earlier, the RVS figures out what's visible *each time it renders an image*. This means that moving polygons do not make one bit of difference to the RVS. Blow a hole in a wall (or remove it completely, or even destroy the building it resides in, if that's your fancy) and KAGE will keep going like nothing has happened.

Earlier in this document, I talked about the extra overhead incurred by trying to add characters to a scene rendered with a BSP tree. I also mentioned that if a player character were to approach the viewpoint and fill a large portion of the screen, that the BSP renderer would end up drawing a lot of environment polygons (hidden behind the character) that would quickly be over-drawn. In the RVS, dynamic polygons act just like environmental polygons. So characters moving around the environment don't offer a slow-down, rather they AID in the visibility. If they "hide" a portion of the environment, then that portion of the environment won't be rendered.

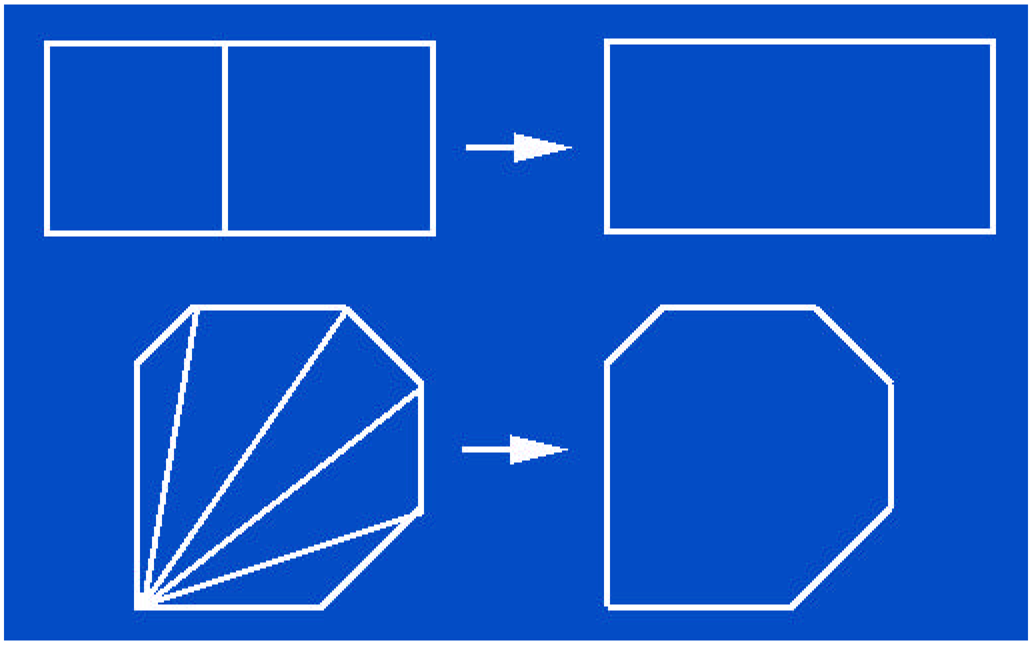
There is no pre-requisite time incurred by any sort of a VIS program (the RVS requires no VIS program at all.) The time it takes to build an RVS data file for KAGE is only a fraction of a second. So level designers can go from building their levels to walking around in them in a matter of seconds.

Since we're on the topic of level designers, lets discuss the limitations placed on them. The RVS will draw what's visible. So the level designers will still be responsible for creating levels that have low polygon counts. However, they will not be forced into re-arranging their levels (adding windy hallways and such.) They will be completely free to give you the full experience they set out to give you, without having to modify their ideas or concepts to suit the underlying technology.

The RVS is also not conservative. It is exact. This means that when the RVS says something needs to be rendered, it IS visible. This cuts our polygon counts down to a fraction of those for other games, yet still giving you the same rendering quality and a MUCH improved frame rate. It's not that we're using simpler geometry, we're just not wasting our time drawing all the polygons that will eventually be over-drawn.

This means we have time for other things. We can add more polygons to the scene to give the player more complex and realistic scenes. We can also add more effects, more characters, etc. and still get great performance.

Let's talk about a different topic: *polygon reduction*. KAGE comes with a tool that is capable of reducing the polygon count for a scene (on average, 50% fewer polygons.) It's not removing any polygons, it simply combines polygons that are side by side. Consider this example:



In the above examples, the geometry did not change but the polygon count was reduced. No visible difference, yet we end up with fewer polygons. Although today's popular 3D rendering hardware can only handle triangles, this reduction plays a large roll in the improvement of the frame rate since the software portion of the engine (the back-face culling, transformation, projection, etc.) has fewer polygons to deal with.

In some cases, the polygons are more complex than the ones I show in the example above (they may contain holes, for example.) In these cases, the original set of polygons still gets reduced, but the result is a set of polygons, rather than one. When this happens, there is almost always a choice of which pieces to combine. The software actually combines them so that the result of the reduction results in the *fewest visible polygons*. This is yet another savings in polygon count that plays a major role in improved frame rates.

ALL THIS AT WHAT COST?

Well, the RVS caused us a lot of problems, to be brutally honest. Because of its flexibility, we couldn't just use a "level editor" like other games. Editors tend to be limited in what they can produce, and we wanted to provide some form of tool for users to create their own levels once we ship a product. Since we cannot ship a tool like 3DS Max with a game, TRI gave me the go-ahead to develop Entropy. Entropy is the KAGE modeler (note, I used the word "modeler" **not** "editor".) It doesn't use brushes, it's a full-blown 3D modeler, and it works a lot like some of the popular 3D tools on the market today.

There was another large factor involved with the development of Entropy. With the nature of the RVS and its ability to have a single, large environment, level designers are faced with the challenge of creating all of that data. We found out quickly that 3DS Max couldn't handle the massive amounts of data as it quickly became very slow and almost entirely unusable on our in-house PCs.

However, the primary reason for writing this document is the following:

In a market that is dominated by 1st person shooters that are all unique in their own way, but all bear the same trademark limitations, it's nearly impossible to get the public to understand that KAGE really is a new technology. I find people asking questions that don't relate to KAGE. Here's an example:

When I demonstrate KAGE to people that make their way into the Terminal Reality offices, they tend to ask many of the same questions. One example might be: "does it support curved surfaces?" And although there is no curved surface code in KAGE at the moment, all it would take to add them is the code required to tessellate the surface into polygons. In contrast to this, adding curved surfaces to BSP systems is a much more complicated process, which could take weeks of development effort. But drop in some generic tessellation code and just make sure the polygons it produces are in the KAGE format and you've got curved surfaces, just that fast.

I'd like to cover one final topic: the misconception that KAGE is a "first-person shooter." KAGE is no better suited for first-person games than it is for third-person, cinematic-camera or any other form of 3D viewing. At the moment, KAGE is none of the above. In the case of KAGE, the game defines the camera angle and the style of the game, not the engine.