

# Ultimate Quake Engine

## *Quake 2 BSP Rendering using Microsoft XNA*

Developed by Jacques Krige (Korax)

### **Credits:**

Very special thanks to Shaun Nirenstein (Crowley9) at NVIDIA for helping out with various aspects and considerations regarding rendering performance and HLSL programming.

## **Quake 2 BSP Renderer**

The past few months I've been working on/off on a project that loads and renders Quake2 (idTech2) BSP files using the .NET language called C# for the program logic and XNA for input and rendering purposes. Initially the idea was to use XNA and build from the ground up game technology to run our titles on using technologies I'm familiar with in the form of the idTech1 and idTech2 engines. Initially the idea was to try and swat two flies at once by developing tech that could carry our game titles on as well as have a clean replacement for our "Unofficial Quake Engine" (UQE) project.

This is the utopia, but it is not possible to accomplish this in a decent timeframe.

On the one hand you want to build technology that is competitive regarding capabilities to other engines or renderers out there, but on the other hand you want to remain faithful to what the technology should be able to do regarding the UQE project. Ultimately its impossible to build a single technology that looks into both directions without having a dated or bloated design. We decided the best was to go, as Excentrax Games, is to utilize XNA for simpler arcade-like titles and for larger titles license a commercial engine like Torque3D and the likes.

So, why are we still working on the Quake2 BSP Renderer?

Since we spent so much time getting it as far as it is, we decided to finish-up the project and release the source code using the GPL license to help other like-minded people to gain access to the source code and hopefully learn from it. Its a much better proposition then stuffing it into our archives to gather dust! There is also another twist to the story; The project turned out to be an experimental renderer for our efforts over at our "idTech" community project, UQE. We are still looking at writing a .NET/XNA version of the idTech2 engine making as much as possible use of XNA to faithfully render the Quake2 game. We will also be looking at building loading and translation paths for Quake1 and Hexen2 at a later stage should the Quake2 project be successful.

The UQE project is a very personal project of mine and with it I'm sharing my passion for the idTech engines with both gamers and game developers interested in the same subject matter. UQE is not one of our primary game development objectives and its development is being managed as time permits us to work on it. The UQE project is great for expanding our knowledge on XNA and helping us build and use game subsystems we can re-use for our XNA targetted projects. It also stands as proof for anyone out there that game development and its related technologies is completely possible in Africa and in showing she is not as dark and illiterate as the media makes her out to be. As far as my research stretches at the time of this writing, Excentrax Games have developed THE most complete XNA-based idTech2 BSP renderer out there. Yet, sadly, our renderer is not complete, it still lacks entity (MD2 and submodels) rendering as well as rendering static lightmap styles.

Lets get right into the technologies the renderer consists of and what you could learn from it...

When the renderer starts-up it loads the BSP level specified in its config.xml file. If this file doesn't exist, it creates one with some default values. The configuration system implemented is a very quick and dirty one just to allow you to change a few variables that would otherwise be hardcoded.

The very first thing Q2BSP does is either load the assets directly from the disk as single files or it loads the assets from the PAK virtual file system. The code managing the PAK file is very basic and just manages the loading from a specific PAK file and is not a fully fledged virtual file system... yet. Even though optionally loading from a .PAK file is great, I think its not the most optimal solution. For a custom virtual file system to work you must be able to stream any data from it for use, but I found it difficult to near impossible to stream a wave file into a form that can be played back by XNA.

I did some reading up on the .XNB format, and it seems it can store multiple files and the XNA API can easily access that file to read data from it. Since we don't want to re-invent a feature that seems to be present, the best thing most likely to do is to build a little conversion tool that can read PAK data and compile XNB data from it, then use the XNB as storage mechanism rather than PAK files.

For this project you'll be able to load any BSP level contained inside a PAK. We have tested it with the pak0.pak file of Quake2 full version, but it should work fine with the demo and other custom PAK files. The only requirement for the loading from a PAK is that ALL assets that has to be loaded needs to be in that very same file, or everything must be external files if the PAK loading option is disabled. Thats how basic the PAK implementation for Q2BSP is. ;)

With the BSP file loaded the renderer is ready to render the world.

The whole world is broken up into surfaces. A surface is a set of vertices that defines a piece of flat geometry. A surface also has a fixed maximum dimensional size limit to easily be broken up for rendering by the PVS and if also an effect of BSP. This means a single flat large wall will be made up by several pieces of surfaces. A surface can have several primitives.

Because of BSP partitioning a leaf can have several surfaces attached to it that forms a piece of geometry. A leaf could for example be that complete large wall, or just a chunk of it. Next we get something we call clusters, or to put it in easier terms, leaf clusters. A cluster is simply a set of possibly visible leafs. So what happens is when you have a specific position in the world, BSP traversing is used to determine which cluster you are in. Once we know the cluster number (index) we look up the PVS and get a list of leaves visible for that cluster. This set of leaves gets drawn to the screen.

If you want to visualize this in your mind's eye, you can imagine a cluster being a volume of space could be big or small, depending on the geometry, you could move your camera in without the geometry PVS changing. Once you move from one cluster to a neighbouring cluster the PVS needs to be re-checked and a new set of leaves needs to be drawn. What is notable is that clusters that are closer together share relatively the same set of leaves. Its very possible that 5 neighbouring clusters would share 70% of the same leaves, depending on how the world is constructed.

One of the performance drawbacks that still remains to be solved is to optimally sort the number of surfaces in the current cluster in such a way that we could make use of Index Buffers and execute draw calls using the `DrawIndexedPrimitives()` function rather than using the `DrawPrimitives()` function that we currently use. `DrawPrimitives()` gets called one for each surface that is visible in the current cluster, but if we could use `DrawIndexedPrimitives()` we could greatly reduce the number of drawing calls that we need to issue. We are still looking at a few scenarios to see where the use of `DrawIndexedPrimitives()` works well and where it breaks functionality.

The first priority is to sort surface rendering by texture (texinfo) then from there build a triangle list based index buffer every time the camera changes PVS cluster. This will change the scene rendering from per-surface `DrawPrimitives()` to per-texture `DrawIndexedPrimitives()`. I'm not sure if it will be jittery if the camera move from one cluster to the next. The only way to know is to code it and find out. Maybe changing the code from `DrawPrimitives()` to `DrawIndexedPrimitives()` frees the CPU enough that you don't notice. For this release the renderer will be rendering the scene primarily using `DrawPrimitives()` until at a later stage we update the code for `DrawIndexPrimitives()` rendering.

The BSP the renderer traverse the BSP nodes and mark surfaces for rendering in two distinct groups namely "solid" surfaces and "translucent/warped" surfaces. The reasoning behind this is to render the world in two phases, phase 1 renders all the solid opaque surfaces in the world, then phase 2 renders all the translucent and warped surfaces. With the solid surface rendering phase it does not really matter if we re-sort the list of surfaces according to texture (texinfo) and not according to its original depth sort order. Most of the surfaces are solid surfaces, and sorting according to texture greatly improves rendering performance. We are not so lucky with translucent surfaces, because we need to render them in the order we get them from the BSP/VIS to make sure we don't introduce rendering anomalies because of the depth and rendering order.

Surfaces with the "warp" flag set gets broken into 64-unit sized sets of sub-surfaces which we call "polygons" within the source code. The reason why warped surfaces gets subdivided is to generate more vertices for the original warped surface to make the ST texture coordinate warping effect possible. This possible large number of primitives using the same texinfo index are being rendered by a call to `DrawIndexedPrimitives()`.

Classic Quake2 lightmapping have also been implemented using multitexturing. In the pixel shader the texture pixels gets multiplied with the lightmap pixels to produce the final pixel for output. We have not yet implemented static lightmap styles, like strobing lights, flickering lights and the likes. There are two types of lightmaps implemented in idTech1, idTech2 and idTech3. They are called "static" lightmaps and "dynamic" lightmaps.

The difference is static lightmaps are pre-processed lightmaps that are fixed and also could have a "style" pre-processed with it, like flickers and strobes and so forth. Then you get "dynamic" lightmaps which gets calculated on-the-fly and gets temporarily blended with the original "static" lightmaps. To give a quick example is when you fire a rocket, it generates a pointlight that lights up the area as it passes through the air and explodes against an object, that light it generates on-the-fly (no pun intended :) ) is a dynamic lightmap.

This dynamic lightmapping looks pretty ugly, especially with idTech1 and idTech2's low resolution lightmaps. We can remedy this by not implementing all the software routines needed to generate and blend this dynamic lightmap by generating dynamic per-pixel lights straight on the GPU... afterall, we already have access to both the texture pixels as well as the lightmap pixels. In the renderer you can switch on/off a per-pixel pointlight that is attached to your camera position. What this pointlight in reality does is it "eats" the color value of only the static lightmap pixel at the pixel shader level and returns a modified static lightmap pixel, which we then multiply with the texture pixel to get our final pixel output result. The calculations isn't 100% perfect yet.

Just for the fun of it we implemented two types of lighting; Classic lightmapped multitexturing and actual per-pixel hardware lighting. The BSP levels are not completely compatible and designed for per-pixel hardware lighting, but its cool to see it sort-of working. Theres a section of code that I left commented that anyone can uncomment which will add per-pixel pointlights for every pointlight entity present in any given BSP level loaded.

The focus of the project was to get good Quake2 BSP rendering up and running with decent performance, not focussing on things like error-checking, good object-orientated design and great performance. When decompressing the archive you'll find a pre-built binary in "release" mode as well as the GPL source code as it stands now at its current state. Included with the binary is a "config.xml" configuration file with a few changeable settings. The build is configured to search and load from PAK0.PAK the "base1.bsp" level. All that needs to be done is to copy the PAK0.PAK of the Quake2 demo or commercial version into the "baseq2" folder located within the "contents" folder.

When Q2BSP starts up it generates a file called "config.txt" which contains a list of the names of all the BSP files that's part of the commercial version of Quake2. This is done to help anyone that doesn't have a PAK reader tool at hand to have a list of BSP files that could be tried out. It is also possible to load some of the deathmatch levels if the commercial version is sufficiently patched, although a PAK editing tool will be needed because the data need to be either merged into a single PAK, or everything needs to be unpacked. This needs to be done since the Q2BSP renderer doesn't have a fully fledged Virtual File System.

Some other features available in the Q2BSP renderer is the ability to switch between solid and wireframe fill modes. The PVS can also be locked to help developers see where the PVS ends from a selected area. Also featuring is the ability to switch the bloom post-process effect on/off. The bloom post-process effect itself is slightly over-emphasized in this experimental renderer, but it demonstrates what the effect can do to make this classic game media slightly (ever so slightly) more current or at least in the right direction.

Lets get to the requirements to be able to execute the Q2BSP renderer successfully:

Microsoft .NET Framework 3.5 Service Pack 1 (or higher)

<http://www.microsoft.com/downloads/details.aspx?FamilyId=AB99342F-5D1A-413D-8319-81DA479AB0D7&displaylang=en>

Microsoft XNA Framework Redistributable 3.0 (or higher)

<http://www.microsoft.com/downloads/details.aspx?familyid=6521D889-5414-49B8-AB32-E3FFF05A4C50&displaylang=en>

Additional (previous/ older) information may be found at the following links:

<http://forums.sagamedev.com/topic.aspx?topicid=555>

<http://forums.sagamedev.com/topic.aspx?topicid=599>

[http://www.sagamedev.com/developerjournal\\_post.aspx?journalid=28](http://www.sagamedev.com/developerjournal_post.aspx?journalid=28)

The project, complete with GPL source code may be downloaded from our UQE website, [www.quake-engine.com](http://www.quake-engine.com)

<http://www.quake-engine.com/download.aspx?game=q2>

Jacques Krige

South African Game Development

<http://www.sagamedev.com>

Ultimate Quake Engine

<http://www.quake-engine.com>