Quake 2 BSP File Format

Return to The Archives

by Max McGuire (07 June 2000)

Introduction

The purpose of this document is to detail the structure of the BSP file format used by Quake 2 to store maps; in particular, the focus is on rendering Quake 2 worlds. Although there is other information contained in the BSP file used for other game elements (such as enemy AI, etc.), I don't have complete information on them so I won't discuss these beyond mentioning them. If you have anything to contribute on these parts please feel free to e-mail at the address listed above. Also, although I've tried my best to minimize errors, if you find anything that is incorrect please let me know.

In addition to just a bare description of the BSP file format, this document also attempts explain the techniques which are at the core of the Quake rendering engine. In general it's assumed that the reader is familiar with the basic principles of 3D graphics, including the BSP tree data structure.

This document covers version 38 of the BSP file format which is the version used by Quake 2. This is also the version of the BSP file format used by Kingpin (which was created using the Quake 2 engine) and therefore all of the information contained is relevant to those files as well. While Quake 1 and Quake 3: Arena use similar BSP formats the formats are not directly compatible. However, because of the similarity in the structure and techniques, this document should be of some use to anyone interested in these formats as well.

My information about the Quake 2 BSP format was all learned by a lot of experimentation in implementing my own Quake 2 BSP renderer along with examining source code for Quake-like engines generously released into the public domain by their authors. These engines were the Twister Engine by Stefano Lanza, the Arnfold 2 Engine by Andrei Fortuna and the Poly Engine by Alexey Goloshubin. The source code for all three of these engines is freely available on the web; if you find something's missing out of this document, I recommend you check out these engines. Id Software has also released the source code for their Quake 2 tools, including the QBSP map file compiler which outputs BSP files, although I didn't find it to be particularly useful in my own exploration.

Legality

Using id Software file formats in a publicly distributed application always seemed a little questionable to me in terms of legality. In preparation of this document I contacted id Software to resolve the issue; John Carmack was kind enough to send along this response:

"We do not legally protect the file formats, but you can't use any of the released tools, or tools derived from them (except the GPL'd Quake 1 tools) to generate content for a commercial venture. If you are writing everything yourself, no problem."

Conventions

To help make this document easier to read I've included C-syntax structures for all of the file structures (with the assumption that there no extra padding bytes between the fields). To simplify the notation and standardize the meaning, the abbreviations int32 and uint32 are used for signed and unsigned 32-bit integers; similar abbreviations are used for 16-bit and 8-bit integers as well. Also, the following two structures are used throughout the BSP file format to store vertices and vectors:

```c
struct point3f
{

    float x;
    float y;
    float z;

};

struct point3s
{

    int16 x;
    int16 y;
    int16 z;

};
```

The second version is used in some places to conserve memory where the precision is unessential (like storing the bounding boxes) since it takes only 6 bytes compared with the 12 byte floating point version. These are regular integer representations, not fixed-point.

Rendering

To give an introduction to the terminology used and the idea behind some of the data structures, this section briefly describes the process of rendering a Quake 2 environment.

A Quake map is carved up into convex regions that become the leaves of the BSP tree, and anytime the camera is positioned within a level it is contained in exactly one of these convex regions. The leaves are grouped together with neighboring leaves to form clusters; exactly how these clusters are formed is determined by the tool that creates the BSP file. For each cluster, a list of all of the other clusters which are potentially visible is stored. This is referred to as the potentially visible set (PVS).

To render a Quake map, first the BSP tree is traversed to determine which leaf the camera is located in. Once we know which leaf the camera is in, we know which cluster it's in (remembering that each leaf is contained in exactly one cluster). The PVS for the cluster is then decompressed giving a list of all the potentially visible clusters from the camera location. Leaves store a bounding box which his used to quickly cull leaves that are not within the viewing frustum.

BSP Tree

Many people incorrectly associate the BSP tree with the visibility algorithm used by Quake and similar engines. As described above, the visible surface determination is done using a precomputed PVS. The BSP tree is primarily used to divide the map into regions and to quickly determine which region the camera is in. As a result, it isn't that fundamental to any of the rendering algorithms used in Quake and any data structure giving a spatial subdivision (like an octree or a k-D tree) could be used instead. BSP trees are very simple however, and they are useful for some of the other non-rendering tasks in the Quake engine.

Traditionally when discussing BSP trees a distinction is made between those BSP trees that store the faces in the leaves (leaf-based BSP trees) and those that store them in the internal nodes (node-based BSP trees). The variation of BSP tree Quake uses is actually both; the reason for this is the dual use of the BSP tree in both rendering and in collision detection. For the purpose of rendering, it is useful to have the faces stored in the leaves because of the way the PVS is used. If you're not interested in performing collision detection, the faces in the nodes can be completely ignored.

File Header

As far as I know all BSP files are stored in a little-endian (Intel) byte order and converted as loaded when used on a big-ending platform. This allows the same map files to be shared by Quake clients running on all different platforms. If you're going to be loading BSP files on a big-endian machine -- like a Macintosh or UNIX machine -- you're going to have to be careful about swapping the byte order.

The Quake BSP file format is organized around a directory structure, where all of the data is contained in "free floating" lumps within the file. There's a directory in the beginning of the file which tells the offset of the start of the lump and the length. This directory comes after the first 8 bytes of the file is a table of contents which gives the location and length of these lumps.

The format for the BSP file header is the bsp_header structure shown below:

```
struct bsp_lump
{

    uint32    offset;     // offset (in bytes) of the data from the beginning of the file
    uint32    length;     // length (in bytes) of the data

};

struct bsp_header
{

    uint32    magic;      // magic number ("IBSP")
    uint32    version;    // version of the BSP format (38)

    bsp_lump  lump[19];   // directory of the lumps

};
```

The first four bytes of the BSP file is a tag which can be used to identify the file as an id Software BSP file. These bytes spell out "IBSP". Following that is a 32-bit unsigned integer specifying the version number. As mentioned earlier, the version described in this document is 38 (decimal).

Below is a table showing the different lumps contained in the BSP file and their index in the lump array in the header. The purpose of lumps marked with a question mark is not known (the names are derived from the QBSP source code), however they are not needed for constructing a simple Quake level renderer.

Index	Name	Description
0	Entities	MAP entity text buffer
1	Planes	Plane array
2	Vertices	Vertex array
3	Visibility	Compressed PVS data and directory for all clusters
4	Nodes	Internal node array for the BSP tree
5	Texture Information	Face texture application array
6	Faces	Face array
7	Lightmaps	Lightmaps
8	Leaves	Internal leaf array of the BSP tree
9	Leaf Face Table	Index lookup table for referencing the face array from a leaf
10	Leaf Brush Table	?
11	Edges	Edge array
12	Face Edge Table	Index lookup table for referencing the edge array from a face

13	Models	?
14	Brushes	?
15	Brush Sides	?
16	Pop	?
17	Areas	?
18	Area Portals	?

Most of the lumps are stored as an array of structures where the structure has a fixed size. For instance the vertex lump is an array of point3f structures. Since each point3f is 12 bytes, the number of vertices can be computed by dividing the size of the vertex lump by 12. Similar computations can be done for the plane, node, texture information, faces, leaf and edge lumps.

In the following sections, the purpose and structure of the known lumps is discussed.

Vertex Lump

The vertex lump is a list of all of the vertices in the world. Each vertex is 3 floats which makes 12 bytes per vertex. You can compute the numbers of vertices by dividing the length of the vertex lump by 12.

Quake uses a coordinate system in which the z-axis is pointing in the "up" direction. Keep in mind that if you modify the coordinates to use a different system, you will also need to adjust the bounding boxes and the plane equations.

Edge Lump

Not only are vertices shared between faces, but edges are as well. Each edge is stored as a pair of indices into the vertex array. The storage is two 16-bit integers, so the number of edges in the edge array is the size of the edge lump divided by 4. There is a little complexity here because an edge could be shared by two faces with different windings, and therefore there is no particular "direction" for an edge. This is further discussed in the section on face edges.

Face Lump

The face lump stores an array of bsp_face structures which have the following format:

```
struct bsp_face
{

    uint16   plane;             // index of the plane the face is parallel to
    uint16   plane_side;        // set if the normal is parallel to the plane normal

    uint32   first_edge;        // index of the first edge (in the face edge array)
    uint16   num_edges;         // number of consecutive edges (in the face edge array)

    uint16   texture_info;      // index of the texture info structure

    uint8    lightmap_syles[4]; // styles (bit flags) for the lightmaps
    uint32   lightmap_offset;   // offset of the lightmap (in bytes) in the lightmap lump

};
```

The size of the bsp_face structure is 20 bytes, the number of faces can be determined by dividing the size of the face lump by 20.

The plane_side is used to determine whether the normal for the face points in the same direction or opposite the plane's normal. This is necessary since coplanar faces which share the same node in the BSP tree also share the same normal, however the true normal for the faces could be different. If plane_side is non-zero, then the face normal points in the opposite direction as the plane's normal.

The details of texture and lightmap coordinate generation are discussed in the section on texture information and lightmap sections.

Face Edge Lump

Instead of directly accessing the edge array, faces contain indices into the face edge array which are in turn used as indices into the edge array. The face edge lump is simply an array of unsigned 32-bit integers. The number of elements in the face edge array is the size of the face edge lump divided by 4 (note that this is not necessarily the same as the number of edges).

Since edges are referenced from multiple sources they don't have any particular direction. If the edge index is positive, then the first point of the edge is the start of the edge; if it's negative, then the second point is used as the start of the edge (and obviously when you look it up in the edge array you drop the negative sign).

Plane Lump

The plane lump stores and array of bsp_plane structures which are used as the splitting planes in the BSP. The format for for the bsp_plane structure is:

```
struct bsp_plane
```

```
{

    point3f   normal;       // A, B, C components of the plane equation
    float     distance;     // D component of the plane equation
    uint32    type;         // ?

};
```

Each bsp_plane structure is 20 bytes, so the number of planes is the size of the plane lump divided by 20.

The x, y and z components of the normal correspond to A, B, C constants and the distance to the D constant in the plane equation:

```
F(x, y, z) = Ax + By + Cz - D
```

A point is on the plane is F(x, y, z) = 0, in front of the plane if F(x, y, z) > 0 and behind the plane if F(x, y, z) < 0. This is used in the traversal of the BSP tree is traversed.

Node Lump

The nodes are stored as an array in the node lump, where the first element is the root of the BSP tree. The format of a node is the bsp_node structure:

```
struct bsp_node
{

    uint32    plane;             // index of the splitting plane (in the plane array)

    int32     front_child;       // index of the front child node or leaf
    int32     back_child;        // index of the back child node or leaf

    point3s   bbox_min;          // minimum x, y and z of the bounding box
    point3s   bbox_max;          // maximum x, y and z of the bounding box

    uint16    first_face;        // index of the first face (in the face array)
    uint16    num_faces;         // number of consecutive edges (in the face array)

};
```

Each bsp_node is 28 bytes, so the number of nodes is the size of the node lump divided by 28.

Since a child of a node may be a leaf and not a node, negative values for the index are used to incate a leaf. The exact position in the leaf array for a negative index is computed as -(index + 1) so that the first negative number maps to 0.

Since the bounding boxes are axis aligned, the eight coordinates of the box can be found from the minimum and maximum coordinates stored in the bbox_min and bbox_max fields.

As mentioned earlier, the faces listed in the node are not used for rendering but rather for collision detection.

Leaf Lump

The leaf lump stores an array of bsp_leaf structures which are the leaves of the BSP tree. The format of the bsp_leaf structure is:

```
struct bsp_leaf
{

    uint32    brush_or;          // ?

    uint16    cluster;           // -1 for cluster indicates no visibility information
    uint16    area;              // ?

    point3s   bbox_min;          // bounding box minimums
    point3s   bbox_max;          // bounding box maximums

    uint16    first_leaf_face;   // index of the first face (in the face leaf array)
    uint16    num_leaf_faces;    // number of consecutive edges (in the face leaf array)
    uint16   first_leaf_brush;   // ?
    uint16   num_leaf_brushes;   // ?

};
```

The bsp_leaf structure is 28 bytes so the number of leaves is the size of the leaf lump divided by 28.

Leaves are grouped into clusters for the purpose of storing the PVS, and the cluster field gives the index into the array stored in the visibility lump. See the Visibility section for more information on this. If the cluster is -1, then the leaf has no visibility information (in this case the leaf is not a place that is reachable by the player).

Leaf Face Lump

Instead of directly accessing the face array, leaves contain indices into the leaf face array which are in turn used as indices into the face array. The face edge lump is simply an array of unsigned 16-bit integers. The number of elements in this array is the size of the leaf face lump divided by 2; this is not necessarily the same as the number of faces in the world.

Texture Information Lump

Texture information structures specify all of the details about how a face is textured. The structure of the a single texture information is the bsp_texinfo structure:

```
struct bsp_texinfo
{

    point3f  u_axis;
    float    u_offset;

    point3f  v_axis;
    float    v_offset;

    uint32   flags;
    uint32   value;

    char     texture_name[32];

    uint32   next_texinfo;

};
```

The bsp_texinfo structure is 76 bytes, so the number of texture information structures is the size of the lump divided by 76.

Textures are applied to faces using a planar texture mapping scheme. Instead of specifying texture coordinates for each of the vertices of the face, two texture axes are specified which define a plane. Texture coordinates are generated by projecting the vertices of the face onto this plane.

While this may seem to add some complexity to the task of the programmer, it greatly reduces the burden of the level designer in aligning textures across multiple faces.

The texture coordinates (u, v) for a point(x, y, z) are found using the following computation:

```
u = x * u_axis.x + y * u_axis.y + z * u_axis.z + u_offset
v = x * v_axis.x + y * v_axis.y + z * v_axis.z + v_offset
```

My own experience has shown that it is perfectly reasonable to generate the coordinates in real-time rather than storing them.

The texture name is stored with a path but without any extension. Typically, if you are loading a Quake 2 map you would append the extension "wal" to the name and then load it from the PAK file. If you're loading a Kingpin map you would append the extension "tga" and then load if from disk (Kingpin stires the textures outside of the PAK file). See the section on the WAL texture format for more details.

Visibility Lump

Leaves are grouped into clusters for the purpose of storing the visibility information. This is to conserve space in the PVS since it's likely that nearby leaves will have similar potentially visible areas. The first 4 bytes of the visibility lump is a 32-bit unsigned integer indicating the number of clusters in the map, and after that is an array of bsp_vis_offset structures with the same number of elements as there are clusters. The bsp_vis_offset structure format is:

```
struct bsp_vis_offset
{

        uint32 pvs;    // offset (in bytes) from the beginning of the visibility lump
        uint32 phs;    // ?

};
```

The rest of the visibility lump is the actual visibility information. For every cluster the visibility state (either visible or occluded) is stored for every other cluster in the world. Clusters are always visible from within themselves. Because this is such a massive amount of data, this array is stored as a bit vector (one bit per element) and 0's are run-length-encoded. Here's an example of a C-routine to decompress the PVS into a byte array (this was adapted from the "Quake Specifications" document):

```
int v = offset;

memset(cluster_visible, 0, num_clusters);

for (int c = 0; c < num_clusters; v++) {

   if (pvs_buffer[v] == 0) {
      v++;
      c += 8 * pvs_buffer[v];
   } else {
      for (uint8 bit = 1; bit != 0; bit *= 2, c++) {
         if (pvs_buffer[v] & bit) {
            cluster_visible[c] = 1;
         }
      }
   }

}
```

Lightmap Lump

Static lighting is handled in the Quake environment using lightmaps, which are low resolution bitmaps that are combined with the texture when rendering a face. Lightmaps are unique to a face (unlike textures which may be used on multiple faces). The maximum size for any lightmap is 16x16 which is enforced by the BSP creation tool by splitting faces which would require larger lightmaps. In Quake 1 lightmaps were limited to grayscale, however Quake 2 allows for 24-bit true color lighting.

Lightmaps are applied either through multi-texturing or multi-pass rendering and should be multiplied with the texture for a face when rendering.

The lightmap lump contains the data for all of the lightmaps in the world stored consecutively. Faces store the offset of their lightmap in this lump in the bsp_face structure's lightmap_offset. The actual lightmap data is stored in a 24-bits-per-pixel RGB format in a left-right, top-down order starting at this offset.

The width and height of a lightmap isn't explicitly stored anywhere and must be computed. If max_u is the maximum u component of all of the texture coordinates for a face, and min_u is the minimum u component of all the texture coordinates for a face (and similarly for the max_v and min_v), then the width and height of the lightmap for the face is given by the computation:

```
lightmap_width  = ceil(max_u / 16) - floor(min_u / 16) + 1
lightmap_height = ceil(max_v / 16) - floor(min_v / 16) + 1
```

For information on computing the texture coordinates for a face see the Texture Information Lump section.

WAL Image Format

Quake 2 stores textures in a proprietary 2D image format called WAL. While this isn't actually part of the BSP file format, it's essential information for loading Quake 2 maps so I've decided to include it. Note that this doesn't apply to Kingpin maps which use textures stored as TGA files.

WAL textures are stored in a 8-bit indexed color format with a specific palette being used by all textures (this palette is stored in the PAK data file that comes with Quake 2). Four mip-map levels are stored for each texture at sizes decreasing by a factor of two. This is mostly for software rendering since most 3D APIs will automatically generate the mip-map levels when you create a texture. Each frame of an animated texture is stored as an individual WAL file, and the animation sequence is encoded by storing the name of the next texture in the sequence for each frame; texture names are stored with paths and without any extension.

The format for the WAL file header is the wal_header structure:

```
struct wal_header
{

    char    name[32];        // name of the texture

    uint32  width;           // width (in pixels) of the largest mipmap level
    uint32  height;          // height (in pixels) of the largest mipmap level

    int32   offset[4];       // byte offset of the start of each of the 4 mipmap levels

    char    next_name[32];   // name of the next texture in the animation

    uint32  flags;           // ?
    uint32  contents;        // ?
    uint32  value;           // ?

};
```

The actual texture data is stored in an 8-bits-per-pixel raw format in a left-right, top-down order.

Entities

The entity section of the BSP file is an ASCII text buffer, formatted in the same manner as the entities in the MAP file (recall that BSP files are compiled from MAP files). Entities are things like the player spawn points, lights, health, ammunition and guns. A comprehensive list of the entities used in Quake 1 (and because of the considerable overlap, Quake 2) is available in the MAP file format section of the "Quake Specifications" document.