Unofficial BSP v30 File Spec

Table of content

Introduction

The following file specification concerns the BSP file format version 30, as it has been designed by the game developper Valve and used in their famous GoldSrc Engine. The file extension is ".bsp".

Important: The following informations do NOT rely on any officially published file specification from Valve Corporation. The file format is still in use by the proprietary Half-Life Engine (better known name of the GoldSrc Engine) implying that there is no public source code of either the file loader or renderer. The following specification has been put together based on informations from the open source project Black Engine as well as the compilers included in the Half-Life SDK which also contains their source.

This file spec uses constructs from the C programming language to describe the different data structures used in the BSP file format. Architecture dependent datatypes like integers are replaced by exact-width integer types of the C99 standard in the stdint.h header file, to provide more flexibillity when using x64 platforms. Basic knowledge about Binary Space Partitioning is recommended. There is a common struct used to represent a point in 3-dimensional space which is used throughout the file spec and the code of the hlbsp project.

```
#include <stdint.h>

typedef struct _VECTOR3D
{
	float x, y, z;
} VECTOR3D;
```

Header

Like almost every file also a BSP file starts with a specific file header which is constucted as follows:

```
#define LUMP_ENTITIES      0
#define LUMP_PLANES        1
#define LUMP_TEXTURES      2
#define LUMP_VERTICES      3
#define LUMP_VISIBILITY    4
#define LUMP_NODES         5
#define LUMP_TEXINFO       6
#define LUMP_FACES         7
#define LUMP_LIGHTING      8
#define LUMP_CLIPNODES     9
#define LUMP_LEAVES        10
#define LUMP_MARKSURFACES  11
#define LUMP_EDGES         12
#define LUMP_SURFEDGES     13
#define LUMP_MODELS        14
#define HEADER_LUMPS       15

typedef struct _BSPHEADER
{
	int32_t nVersion;                // Must be 30 for a valid HL BSP file
	BSPLUMP lump[HEADER_LUMPS]; // Stores the directory of lumps
} BSPHEADER;
```

The file header begins with an 32bit integer containing the file version of the BSP file (the magic number). This should be 30 for a valid BSP file used by the Half-Life Engine. Subseqently, there is an array of entries for the so-called lumps. A lump is more or less a section of the file containing a specific type of data. The lump entries in the file header address these lumps, accessed by the 15 predefined indexes. A lump entry struct is defined as follows:

```
typedef struct _BSPLUMP
{
	int32_t nOffset; // File offset to data
	int32_t nLength; // Length of data
} BSPLUMP;
```

To read the different lumps from the given BSP file, every lump entry file states the beginning of each lump as an offset relativly to the beginning of the file. Additionally, the lump entry also gives the length of the addressed lump in bytes. The Half-Life BSP compilers also define several constants for the maximum size of each lump, as they use static, global arrays to hold the data. The hlbsp project uses malloc() to allocate the required memory for each lump depending on their actual size.

```
#define MAX_MAP_HULLS            4

#define MAX_MAP_MODELS           400
#define MAX_MAP_BRUSHES          4096
#define MAX_MAP_ENTITIES         1024
#define MAX_MAP_ENTSTRING        (128*1024)

#define MAX_MAP_PLANES           32767
#define MAX_MAP_NODES            32767
#define MAX_MAP_CLIPNODES        32767
#define MAX_MAP_LEAFS            8192
#define MAX_MAP_VERTS            65535
#define MAX_MAP_FACES            65535
#define MAX_MAP_MARKSURFACES 65535
#define MAX_MAP_TEXINFO          8192
#define MAX_MAP_EDGES            256000
#define MAX_MAP_SURFEDGES        512000
#define MAX_MAP_TEXTURES         512
#define MAX_MAP_MIPTEX           0x200000
#define MAX_MAP_LIGHTING         0x200000
#define MAX_MAP_VISIBILITY       0x200000

#define MAX_MAP_PORTALS          65536
```

The following sections will focus on every lump of the BSP file.

The Entity Lump (LUMP_ENTITIES)

The entity lump is basically a pure ASCII text section. It consists of the string representations of all entities, which are copied directly from the input file to the output BSP file by the compiler. An entity might look like this:

```
{
"origin" "0 0 -64"
"angles" "0 0 0"
"classname" "info_player_start"
}
```

Every entity begins and ends with curly brackets. Inbetween there are the attributes of the entity, one in each line, which are pairs of strings enclosed by quotes. The first string is the name of the attribute (the key), the second one its value. The attribute "classname" is mandatory for every entity specifiying its type and therefore, how it is interpreted by the engine.

The map compilers also define two constants for the maximum length of key and value:

```
#define MAX_KEY          32
#define MAX_VALUE        1024
```

The Planes Lump (LUMP_PLANES)

This lump is a simple array of binary data structures:

```
#define PLANE_X 0        // Plane is perpendicular to given axis
```

```
#define PLANE_Y 1
#define PLANE_Z 2
#define PLANE_ANYX 3   // Non-axial plane is snapped to the nearest
#define PLANE_ANYY 4
#define PLANE_ANYZ 5

typedef struct _BSPPLANE
{
      VECTOR3D vNormal; // The planes normal vector
      float fDist;          // Plane equation is: vNormal * X = fDist
      int32_t nType;       // Plane type, see #defines
} BSPPLANE;
```

Each of this structures defines a plane in 3-dimensional space by using the Hesse normal form:

normal * point - distance = 0

Where vNormal is the normalized normal vector of the plane and fDist is the distance of the plane to the origin of the coord system. Additionally, the structure also saves an integer describing the orientation of the plane in space. If nType equals PLANE_X, then the normal of the plane will be parallel to the x axis, meaning the plane is perpendicular to the x axis. If nType equals PLANE_ANYX, then the plane's normal is nearer to the x axis then to any other axis. This information is used by the renderer to speed up some computations.

The Texture Lump (LUMP_TEXTURES)

The texture lump is somehow a bit more complex then the other lumps, because it is possible to save textures directly within the BSP file instead of storing them in external WAD files. This lump also starts with a small header:

```
typedef struct _BSPTEXTUREHEADER
{
      uint32_t nMipTextures; // Number of BSPMIPTEX structures
} BSPTEXTUREHEADER;
```

The header only consists of an unsigned 32bit integer indicating the number of stored or referenced textures in the texture lump. After the header follows an array of 32bit offsets pointing to the beginnings of the seperate textures.

```
typedef int32_t BSPMIPTEXOFFSET;
```

Every offset gives the distance in bytes from the beginning of the texture lump to one of the beginnings of the BSPMIPTEX structure, which are equal in count to the value given in the texture header.

```
#define MAXTEXTURENAME 16
#define MIPLEVELS 4
typedef struct _BSPMIPTEX
{
      char szName[MAXTEXTURENAME];    // Name of texture
      uint32_t nWidth, nHeight;       // Extends of the texture
      uint32_t nOffsets[MIPLEVELS]; // Offsets to texture mipmaps BSPMIPTEX;
```

```
} BSPMIPTEX;
```

Each of this structs describes a texture. The name of the texture is a string and may be 16 characters long (including the null-character at the end, char equals a 8bit signed integer). The name of the texture is needed, if the texture has to be found and loaded from an external WAD file. Furthermore, the struct contains the width and height of the texture. The 4 offsets at the end can either be zero, if the texture is stored in an external WAD file, or point to the beginnings of the binary texture data within the texture lump relative to the beginning of it's BSPMIPTEX struct.

The Vertices Lump (LUMP_VERTICES)

This lump simply consists of all vertices of the BSP tree. They are stored as a primitve array of triples of floats.

```
typedef VECTOR3D BSPVERTEX;
```

Each of this triples, obviously, represents a point in 3-dimensional space by giving its three coordinates.

The VIS Lump (LUMP_VISIBILITY)

The VIS lump contains data, which is irrelevant to the actual BSP tree, but offers a way to boost up the speed of the renderer significantly. Especially complex maps profit from the use if this data. This lump contains the so-called Potentially Visible Sets (PVS) (also called VIS lists) in the same amout of leaves of the tree, the user can enter (often referred to as VisLeaves). The visiblilty lists are stored as sequences of bitfields, which are run-length encoded.
Important: The generation of the VIS data is a very time consuming process (several hours) and also done by a seperate compiler. It can therefore be skipped when compiling the map, resulting in BSP files with no VIS data at all!

The Nodes Lump (LUMP_NODES)

This lump is simple again and contains an array of binary structures, the nodes, which are a major part of the BSP tree.

```
typedef struct _BSPNODE
{
      uint32_t iPlane;                  // Index into Planes lump
      int16_t iChildren[2];           // If > 0, then indices into Nodes // otherwise bitwise inverse
indices into Leafs
      int16_t nMins[3], nMaxs[3]; // Defines bounding box
      uint16_t firstFace, nFaces; // Index and count into Faces
} BSPNODE;
```

Every BSPNODE structure represents a node in the BSP tree and every node equals more or less a division step of the BSP algorithm. Therefore, each node has an index (iPlane) referring to a plane in the plane lump which devides the node into its two child nodes. The

childnodes are also stored as indexes. Contrary to the plane index, the node index for the child is signed. If the index is larger than 0, the index indicates a child node. If it is equal to or smaller than zero (no valid array index), the bitwise inversed value of the index gives an index into the leaves lump. Additionally two points (nMins, nMaxs) span the bounding box (AABB, axis aligned bounding box) delimitting the space of the node. Finally firstFace indexes into the face lump and spezifies the first of nFaces surfaces contained in this node.

The Texinfo Lump (LUMP_TEXINFO)

The texinfo lump contains informations about how textures are applied to surfaces. The lump itself is an array of binary data structures.

```
typedef struct _BSPTEXTUREINFO
{
	VECTOR3D vS;
	float fSShift;		// Texture shift in s direction
	VECTOR3D vT;
	float fTShift;		// Texture shift in t direction
	uint32_t iMiptex; // Index into textures array
	uint32_t nFlags;	// Texture flags, seem to always be 0
} BSPTEXTUREINFO;
```

This struct is mainly responsible for the calculation of the texture coordinates (vS, fSShift, vT, fTShift). This values determine the position of the texture on the surface. The iMiptex integer refers to the textures in the texture lump and would be the index in an array of BSPMITEX structs. Finally, there are 4 Bytes used for flags. Somehow they seem to always be 0;

The Faces Lump (LUMP_FACES)

The face lump contains the surfaces of the scene. Once again an array of structs:

```
typedef struct _BSPFACE
{
	uint16_t iPlane;			// Plane the face is parallel to
	uint16_t nPlaneSide;		// Set if different normals orientation
	uint32_t iFirstEdge;		// Index of the first surfedge
	uint16_t nEdges;			// Number of consecutive surfedges
	uint16_t iTextureInfo;		// Index of the texture info structure
	uint8_t nStyles[4];			// Specify lighting styles
	uint32_t nLightmapOffset; // Offsets into the raw lightmap data
} BSPFACE;
```

The first number of this data structure is an index into the planes lump giving a plane which is parallel to this face (meaning they share the same normal). The second value may be seen as a boolean. If nPlaneSide equals 0, then the normal vector of this face equals the one of the parallel plane exactly. Otherwise, the normal of the plane has to be multiplied by -1 to point into the right direction. Afterwards we have an index into the surfedges lump, as well as the count of consecutive surfedges from that position. Furthermore there is an index into the texture info lump, which is used to find the BSPTEXINFO structure needed to calculate the texture coordinates for this face. Afterwards, there are four bytes giving some lighting information (partly used by the renderer to hide sky surfaces). Finally

we have an offset in byes giving the beginning of the binary lightmap data of this face in the lighting lump.

The Lightmap Lump (LUMP_LIGHTING)

This is one of the largest lumps in the BSP file. The lightmap lump stores all lightmaps used in the entire map. The lightmaps are arrays of triples of bytes (3 channel color, RGB) and stored continuously.

The Clipnodes Lump (LUMP_CLIPNODES)

This lump contains the so-called clipnodes, which build a second BSP tree used only for collision detection.

```
typedef struct _BSPCLIPNODE
{
      int32_t iPlane;          // Index into planes
      int16_t iChildren[2]; // negative numbers are contents
} BSPCLIPNODE;
```

This structure is a reduced form of the BSPNODE struct from the nodes lump. Also the BSP tree built by the clipnodes is simpler than the one described by the BSPNODEs to accelerate collision calculations.

The Leaves Lump (LUMP_LEAVES)

The leaves lump contains the leaves of the BSP tree. Another array of binary structs:

```
#define CONTENTS_EMPTY            -1
#define CONTENTS_SOLID            -2
#define CONTENTS_WATER            -3
#define CONTENTS_SLIME            -4
#define CONTENTS_LAVA             -5
#define CONTENTS_SKY               -6
#define CONTENTS_ORIGIN          -7
#define CONTENTS_CLIP              -8
#define CONTENTS_CURRENT_0      -9
#define CONTENTS_CURRENT_90    -10
#define CONTENTS_CURRENT_180   -11
#define CONTENTS_CURRENT_270   -12
#define CONTENTS_CURRENT_UP    -13
#define CONTENTS_CURRENT_DOWN -14
#define CONTENTS_TRANSLUCENT   -15

typedef struct _BSPLEAF
{
      int32_t nContents;                                  // Contents enumeration
      int32_t nVisOffset;                                 // Offset into the visibility lump
      int16_t nMins[3], nMaxs[3];                      // Defines bounding box
      uint16_t iFirstMarkSurface, nMarkSurfaces; // Index and count into marksurfaces array
      uint8_t nAmbientLevels[4];                       // Ambient sound levels
} BSPLEAF;
```

The first entry of this struct is the type of the content of this leaf. It can be one of the predefined values, found in the compiler source codes, and is litte relevant for the actual

rendering process. All the more important is the next integer containing the offset into the vis lump. It defines the start of the raw PVS data for this leaf. If this value equals -1, no VIS lists are available for this leaf, usually if the map has been built without the VIS compiler. The next two 16bit integer triples span the bounding box of this leaf. Furthermore, the struct contains an index pointing into the array of marksurfaces loaded from the marksufaces lump as well as the number of consecutive marksurfaces belonging to this leaf. The marksurfaces are looped through during the rendering process and point to the actual faces. The final 4 bytes somehow spezify the volume of the ambient sounds.

The Marksurfaces Lump (LUMP_MARKSURFACES)

The marksurfaces lump is a simple array of short integers.

```
typedef uint16_t BSPMARKSURFACE;
```

This lump is a simple table for redirecting the marksurfaces indexes in the leafs to the actial face indexes. A leaf inserts it's marksurface indexes into this array and gets the associated faces contained within this leaf.

The Edges Lump (LUMP_EDGES)

The edges are defined as an array of structs:

```
typedef struct _BSPEDGE
{
      uint16_t iVertex[2]; // Indices into vertex array
} BSPEDGE;
```

The edges delimit the face and further refer to the vertices of the face. Each edge is pointing to the start and end vertex of the edge.

The Surfedges Lump (LUMP_SURFEDGES)

Another array of integers.

```
typedef int32_t BSPSURFEDGE;
```

This lump represents pretty much the same mechanism as the marksurfaces. A face can insert its surfedge indexes into this array to get the corresponding edges delimitting the face and further pointing to the vertexes, which are required for rendering. The index can be positive or negative. If the value of the surfedge is positive, the first vertex of the edge is used as vertex for rendering the face, otherwise, the value is multiplied by -1 and the second vertex of the indexed edge is used.

The Models Lump (LUMP_MODELS)

Array of structs:

```
#define MAX_MAP_HULLS 4

typedef struct _BSPMODEL
{
      float nMins[3], nMaxs[3];               // Defines bounding box
      VECTOR3D vOrigin;                           // Coordinates to move the // coordinate system
      int32_t iHeadnodes[MAX_MAP_HULLS]; // Index into nodes array
      int32_t nVisLeafs;                          // ???
      int32_t iFirstFace, nFaces;            // Index and count into faces
} BSPMODEL;
```

A model is kind of a mini BSP tree. Its size is determinded by the bounding box spaned by the first to members of this struct. The major difference between a model and the BSP tree holding the scene is that the models use a local coordinate system for their vertexes and just state its origin in world coordinates. During rendering the coordinate system is translated to the origin of the model (glTranslate()) and moved back after the models BSP tree has been traversed. Furthermore their are 4 indexes into node arrays. The first one has proofed to index the root node of the mini BSP tree used for rendering. The other three indexes could probably be used for collision detection, meaning they point into the clipnodes, but I am not sure about this. The meaning of the next value is also somehow unclear to me. Finally their are direct indexes into the faces array, not taking the redirecting by the marksurfaces.