Source BSP File Format

From Valve Developer Community

This article is based on Rof's "The Source Engine BSP File Format" from October 2005 (http://web.archive.org/web/20050426034532/http://www.geocities.com/cofrdrbob/bspformat.html), retrieved from Geocities before the service shut down. A mirror to the original version can be found here (http://www.bagthorpe.org/bob/cofrdrbob/bspformat.html).

Contents

Introduction

This document describes the structure of the BSP file format used by the Source engine. The format is similar but not identical to the BSP file formats of the Half-Life 1 engine (GoldSrc), which is in turn based on the Quake, Quake II and QuakeWorld file formats, plus that of the later Quake III Arena. Because of this, Max McGuire's article, Quake 2 BSP File Format (http://www.flipcode.com/archives/Quake_2_BSP_File_Format.shtml) is also of invaluable help in understanding the overall structure of the format and the parts of it that have remained the same or similar to its predecessors.

This document is an extension of notes made by Rof during the writing of his Half-Life 2 BSP file decompiler, VMEX. It therefore focuses on those parts of the format necessary to perform map decompilation (conversion of the BSP file back into a VMF file which can be loaded by the Hammer map editor).

Most of the information in this document comes from the Max McGuire article referenced above, from the source code included in the Source SDK (particularly the C header file *public/bspfile.h*), and from Rof's own experimentation during the writing of VMEX.

A certain familiarity with C/C++, geometry, and Source mapping terms is assumed on the part of the reader. Code (mostly C structures) is given in a `fixed width font`. Sometimes the structures as shown are modified from their actual definitions in the SDK header files, for reasons of clarity and consistency.

Overview

The BSP file contains the vast majority of the information needed by the Source engine to render and play a map. This includes the geometry of all the polygons in the level; references to the names and orientation of the textures to be drawn on those polygons; the data used to simulate the physical behaviour of the player and other items during the game; the location and properties of all brush-based, model (prop) based, and non-visible (logical) entities in the map; and the BSP tree and visibility table used to locate the player location in the map geometry and to render the visible map as efficiently as possible. Optionally, the map file can also contain any custom textures and models used on the level, embedded inside the map's Pakfile lump (see below).

Information not stored in the BSP file includes the map description text displayed by multiplayer games (such as Counter-Strike: Source or Half-Life 2: Deathmatch) after loading the map (stored in the file *mapname*.txt) and the AI navigation file used by non-player characters (NPCs) which need to navigate the map (stored in the file *mapname*.nav). Because of the way the Source engine file system works, these external files may also be embedded in the BSP file's Pakfile lump, though usually they are not.

Official map files are stored in the Steam Game Cache File (GCF) format, and are accessed through the Steam file system by the game engine. They can be extracted from the GCF files using Nemesis' GCFScape for perusal outside of Steam. Newer games that are using the VPK file format usually have their maps stored directly on the file system of the operating system.

The data in the BSP file can be stored in little-endian for PC/Mac or in big-endian for PS3/X360. Byte-swapping is required when loading a little-endian file on a big-endian format platform such as Java and vice versa.

BSP file header

The BSP file starts with a header. This structure identifies the file as a Valve Source Engine BSP file, identifies the version of the format, and is then followed by a directory listing of the location, length, and version of up to 64 subsections of the file, known as *lumps*, that store different parts of the map data. Finally, the map revision is given.

The structure of the header is given in the SDK's *public/bspfile.h* header file, a file which will be referencing extensively throughout this document. The header is 1036 bytes long in total:

In Alien Swarm, this struct has been renamed to BSPHeader_t.

```
struct dheader_t
{
    int ident;                // BSP file identifier
    int version;              // BSP file version
    lump_t  lumps[HEADER_LUMPS];  // lump directory array
    int mapRevision;          // the map's revision (iteration, version) number
};
```

Where `ident` is a 4-byte magic number defined as:

```
// little-endian "VBSP"   0x50534256
#define IDBSPHEADER (('P'<<24)+('S'<<16)+('B'<<8)+'V')
```

The first four bytes of the file are thus always `VBSP` (in ASCII). These bytes identify the file as a Valve BSP file; other BSP file formats use a different magic number (such as for id Software's Quake engine games, which start with `IBSP`). The GoldSrc BSP format does not use any magic number at all. The order of the magic number can also be used to determine the endianness of the file: `VBSP` is used for little-endian and `PSBV` for big-endian.

The second integer is the version of the BSP file format (BSPVERSION); for Source games, it ranges from 19 to 21 with exception of VTMB, which uses an earlier version of the format, 17 (see table below). Note that BSP file formats for other engines (HL1, Quake series, etc.) use entirely different version number ranges.

Versions

This table gives an overview over the different BSP versions being used in some games based on the Source Engine.

Version	Games	Notes
17	Vampire: The Masquerade – Bloodlines	modified: dface_t
17-18	Half-Life 2 (Beta)	leaked beta
19	Sin Episodes	
19-20	Half-Life 2	19 when released, partially 20 since Source 2007/2009 update
	Half-Life 2: Deathmatch	
	Counter-Strike: Source	
	Day of Defeat: Source	
20	Half-Life 2: Episode One	
	Half-Life 2: Episode Two	
	Half-Life 2: Lost Coast	
	Garry's Mod	
	Team Fortress 2	modified (newly compiled maps): StaticPropLump_t (version = 10), LZMA compressed game lumps, entity info and PAK files
	Portal	
	Left 4 Dead	modified: StaticPropLump_t (version = 8) and dworldlight_t
	Zeno Clash	modified: StaticPropLump_t (version = 7)
	Dark Messiah	modified: dheader_t, StaticPropLump_t, texinfo_t, dgamelump_t, dmodel_t
	Vindictus	many modified structs
	The Ship	modified: StaticPropLump_t
	Bloody Good Time	modified: StaticPropLump_t
	Black Mesa (Source)	
	`(commercial release)`	modified: StaticPropLump_t (version = 11)
21	Left 4 Dead 2	modified: lump_t, old dbrushside_t
	Alien Swarm	
	Portal 2	modified: StaticPropLump_t (version = 9), dbrushside_t and other structs
	Counter-Strike: Global Offensive	modified: StaticPropLump_t (version = 10)

	Dear Esther `(commercial release)`	modified: StaticPropLump_t
	Insurgency `(commercial release)`	
	The Stanley Parable `(commercial release)`	
	The Beginner's Guide	
Tactical Intervention	256-bit XOR encryption	
22	Dota 2	early betas, modified: dbrushside_t, ddispinfo_t
23		modified: dbrushside_t, ddispinfo_t, doverlay_t
27	Contagion	
29	Titanfall	heavily modified

For more details on game-specific BSP formats, see Source BSP File Format/Game-Specific.

Lump structure

Then follows an array of 16-byte `lump_t` structures. HEADER_LUMPS is defined as 64, so there are 64 entries. However, depending on the game and version, some lumps can be undefined or empty.

Each `lump_t` is defined in *bspfile.h*:

```
struct lump_t
{
    int fileofs;    // offset into file (bytes)
    int filelen;    // length of lump (bytes)
    int version;    // lump format version
    char    fourCC[4];  // lump ident code
};
```

The first two integers contain the byte offset (from the beginning of the bsp file) and byte length of that lump's data block; an integer defining the version number of the format of that lump (usually zero), and then a four byte identifier that is usually 0, 0, 0, 0. For compressed lumps, the fourCC contains the uncompressed lump data size in integer form (see section Lump compression for details). Unused members of the lump_t array (those that have no data to point to) have all elements set to zero.

Lump offsets (and their corresponding data lumps) are always rounded up to the nearest 4-byte boundary, though the lump length may not be.

Lump types

The type of data pointed to by the `lump_t` array is defined by its position in the array; for example, the first lump in the array **(Lump 0)** is always the BSP file's entity data (see below). The actual location of the data in the BSP file is defined by the offset and length entries for that lump, and does not need to be in any particular order in the file; for example, the entity data is usually stored towards the end of the BSP file despite being first in the lump array. The array of lump_t headers is therefore a directory of the actual lump data, which may be located anywhere else in the file.

The order of the lumps in the array is defined as:

Index	Engine	Name	Purpose
0	Source 2004	LUMP_ENTITIES	Map entities
1	Source 2004	LUMP_PLANES	Plane array
2	Source 2004	LUMP_TEXDATA	Index to texture names
3	Source 2004	LUMP_VERTEXES	Vertex array
4	Source 2004	LUMP_VISIBILITY	Compressed visibility bit arrays
5	Source 2004	LUMP_NODES	BSP tree nodes
6	Source 2004	LUMP_TEXINFO	Face texture array
7	Source 2004	LUMP_FACES	Face array
8	Source 2004	LUMP_LIGHTING	Lightmap samples
9	Source 2004	LUMP_OCCLUSION	Occlusion polygons and vertices
10	Source 2004	LUMP_LEAFS	BSP tree leaf nodes
11	Source 2007	LUMP_FACEIDS	Correlates between dfaces and Hammer face IDs. Also used as random seed for detail prop placement.
12	Source 2004	LUMP_EDGES	Edge array
13	Source 2004	LUMP_SURFEDGES	Index of edges
14	Source	LUMP_MODELS	Brush models (geometry of brush entities)

	2004		
15	Source 2004	LUMP_WORLDLIGHTS	Internal world lights converted from the entity lump
16	Source 2004	LUMP_LEAFFACES	Index to faces in each leaf
17	Source 2004	LUMP_LEAFBRUSHES	Index to brushes in each leaf
18	Source 2004	LUMP_BRUSHES	Brush array
19	Source 2004	LUMP_BRUSHSIDES	Brushside array
20	Source 2004	LUMP_AREAS	Area array
21	Source 2004	LUMP_AREAPORTALS	Portals between areas
	Source 2004	LUMP_PORTALS	**Confirm:** Polygons defining the boundary between adjacent leaves?
22	Source 2007	LUMP_UNUSED0	Unused
	Source 2009	LUMP_PROPCOLLISION	Static props convex hull lists
	Source 2004	LUMP_CLUSTERS	Leaves that are enterable by the player
23	Source 2007	LUMP_UNUSED1	Unused
	Source 2009	LUMP_PROPHULLS	Static prop convex hulls
	Source 2004	LUMP_PORTALVERTS	Vertices of portal polygons
24	Source 2007	LUMP_UNUSED2	Unused
	Source 2009	LUMP_PROPHULLVERTS	Static prop collision vertices
25		LUMP_CLUSTERPORTALS	

	Source 2004		**Confirm:** Polygons defining the boundary between adjacent clusters?
	Source 2007	LUMP_UNUSED3	Unused
	Source 2009	LUMP_PROPTRIS	Static prop per hull triangle index start/count
26	Source 2004	LUMP_DISPINFO	Displacement surface array
27	Source 2004	LUMP_ORIGINALFACES	Brush faces array before splitting
28	Source 2007	LUMP_PHYSDISP	Displacement physics collision data
29	Source 2004	LUMP_PHYSCOLLIDE	Physics collision data
30	Source 2004	LUMP_VERTNORMALS	Face plane normals
31	Source 2004	LUMP_VERTNORMALINDICES	Face plane normal index array
32	Source 2004	LUMP_DISP_LIGHTMAP_ALPHAS	Displacement lightmap alphas (unused/empty since Source 2006)
33	Source 2004	LUMP_DISP_VERTS	Vertices of displacement surface meshes
34	Source 2004	LUMP_DISP_LIGHTMAP_SAMPLE_POSITIONS	Displacement lightmap sample positions
35	Source 2004	LUMP_GAME_LUMP	Game-specific data lump
36	Source 2004	LUMP_LEAFWATERDATA	Data for leaf nodes that are inside water
37	Source 2004	LUMP_PRIMITIVES	Water polygon data
38	Source 2004	LUMP_PRIMVERTS	Water polygon vertices
39	Source	LUMP_PRIMINDICES	Water polygon vertex index array

	2004		
40	Source 2004	LUMP_PAKFILE	Embedded uncompressed Zip-format file
41	Source 2004	LUMP_CLIPPORTALVERTS	Clipped portal polygon vertices
42	Source 2004	LUMP_CUBEMAPS	env_cubemap location array
43	Source 2004	LUMP_TEXDATA_STRING_DATA	Texture name data
44	Source 2004	LUMP_TEXDATA_STRING_TABLE	Index array into texdata string data
45	Source 2004	LUMP_OVERLAYS	info_overlay data array
46	Source 2004	LUMP_LEAFMINDISTTOWATER	Distance from leaves to water
47	Source 2004	LUMP_FACE_MACRO_TEXTURE_INFO	Macro texture info for faces
48	Source 2004	LUMP_DISP_TRIS	Displacement surface triangles
49	Source 2004	LUMP_PHYSCOLLIDESURFACE	Compressed win32-specific Havok terrain surface collision data. Deprecated and no longer used.
	Source 2009	LUMP_PROP_BLOB	Static prop triangle and string data
50	Source 2006	LUMP_WATEROVERLAYS	**Confirm:** info_overlay's on water faces?
51	Source 2006	LUMP_LIGHTMAPPAGES	Alternate lightdata implementation for Xbox
	Source 2007	LUMP_LEAF_AMBIENT_INDEX_HDR	Index of LUMP_LEAF_AMBIENT_LIGHTING_HDR
52	Source 2006	LUMP_LIGHTMAPPAGEINFOS	Alternate lightdata indices for Xbox
	Source 2007	LUMP_LEAF_AMBIENT_INDEX	Index of LUMP_LEAF_AMBIENT_LIGHTING
53	Source	LUMP_LIGHTING_HDR	HDR lightmap samples

	2006		
54	Source 2006	LUMP_WORLDLIGHTS_HDR	Internal HDR world lights converted from the entity lump
55	Source 2006	LUMP_LEAF_AMBIENT_LIGHTING_HDR	Per-leaf ambient light samples (HDR)
56	Source 2006	LUMP_LEAF_AMBIENT_LIGHTING	Per-leaf ambient light samples (LDR)
57	Source 2006	LUMP_XZIPPAKFILE	XZip version of pak file for Xbox. Deprecated.
58	Source 2006	LUMP_FACES_HDR	*HDR maps may have different face data*
59	Source 2006	LUMP_MAP_FLAGS	Extended level-wide flags. Not present in all levels.
60	Source 2007	LUMP_OVERLAY_FADES	Fade distances for overlays
61	Source 2008	LUMP_OVERLAY_SYSTEM_LEVELS	System level settings (min/max CPU & GPU to render this overlay)
62	Source 2009	LUMP_PHYSLEVEL	**To do**
63	Source 2010	LUMP_DISP_MULTIBLEND	Displacement multiblend info

Lumps 53-56 are only used in version 20+ BSP files. Lumps 22-25 are unused in version 20.

The structure of the data lumps for the known entries is described below. Many of the lumps are simple arrays of structures; however some are of variable length depending on their content. The maximum size or number of entries in each lump is also defined in the *bspfile.h* file, as MAX_MAP_*.

Finally, the header ends with an integer containing the map revision number. This number is based on the revision number of the map's vmf file (`mapversion`), which is increased each time the map is saved in the Hammer editor.

Immediately following the header is the first data lump. This can be any lump in the preceding list (pointed to using the offset field of that lump), though in practice the first data lump is Lump 1, the plane data array.

Lump compression

BSP files for console platforms such as PlayStation 3 and Xbox 360 usually have their lumps compressed with LZMA. In this case, the lump data starts with the following header (from *public/tier1/lzmaDecoder.h*):

```
struct lzma_header_t
{
	unsigned int	id;
```

```
    unsigned int    actualSize;     // always little endian
    unsigned int    lzmaSize;       // always little endian
    unsigned char   properties[5];
};
```

Where `id` is defined as:

```
// little-endian "LZMA"
#define LZMA_ID (('A'<<24)|('M'<<16)|('Z'<<8)|('L'))
```

There are two special cases for compression: `LUMP_PAKFILE` is never compressed and each of the game lumps in `LUMP_GAME_LUMP` are compressed individually. The compressed size of a game lump can be determined by subtracting the current game lump's offset with that of the next entry. For this reason, the last game lump is always an empty dummy which only contains the offset.

Lumps

Plane

The basis of the BSP geometry is defined by planes, which are used as splitting surfaces across the BSP tree structure.

The plane lump **(Lump 1)** is an array of `dplane_t` structures:

```
struct dplane_t
{
    Vector  normal; // normal vector
    float   dist;   // distance from origin
    int type;   // plane axis identifier
};
```

where the Vector type is a 3-vector defined as:

```
struct Vector
{
    float x;
    float y;
    float z;
};
```

Floats are 4 bytes long; there are thus 20 bytes per plane, and the plane lump should be a multiple of 20 bytes long.

The plane is represented by the element `normal`, a normal vector, which is a unit vector (length 1.0) perpendicular to the plane's surface. The position of the plane is given by `dist`, which is the distance from the map origin (0,0,0) to the nearest point on the plane.

Mathematically, the plane is described by the set of points (x, y, z) which satisfy the equation:

$\mathrm{A}x + \mathrm{B}y + \mathrm{C}z = \mathrm{D}$

where A, B, and C are given by the components `normal.x`, `normal.y` and `normal.z`, and D is `dist`. Each plane is infinite in extent, and divides the whole of the map coordinate volume into three pieces, on the plane (F=0), in front of the plane (F>0), and behind the plane (F<0).

Note that planes have a particular orientation, corresponding to which side is considered "in front" of the plane, and which is "behind". The orientation of a plane can be flipped by negating the A, B, C, and D components.

The `type` member of the structure seems to contain flags that indicate planes that are perpendicular to coordinates axes, but is usually not used.

There can be up to 65536 planes in a map (MAX_MAP_PLANES).

Vertex

The vertex lump **(Lump 3)** is an array of coordinates of all the vertices (corners) of brushes in the map geometry. Each vertex is a `Vector` of 3 floats (x, y, and z), giving 12 bytes per vertex.

Note that vertices can be shared between faces, if the vertices coincide exactly.

There are a maximum of 65536 vertices in a map (MAX_MAP_VERTS).

Edge

The edge lump **(Lump 12)** is an array of `dedge_t` structures:

```
struct dedge_t
{
    unsigned short  v[2];   // vertex indices
};
```

Each edge is simply a pair of vertex indices (which index into the vertex lump array). The edge is defined as the straight line between the two vertices. Usually, the edge array is referenced through the Surfedge array (see below).

As for vertices, edges can be shared between adjacent faces. There is a limit of 256000 edges in a map (MAX_MAP_EDGES).

Surfedge

The Surfedge lump **(Lump 13)**, presumable short for surface edge, is an array of (signed) integers. Surfedges are used to reference the edge array, in a somewhat complex way. The value in the surfedge array can be positive or negative. The absolute value of this number is an index into the edge array: if positive, it means the edge is defined from the first to the second vertex; if negative, from the second to the first vertex.

By this method, the Surfedge array allows edges to be referenced for a particular direction. (See the face lump entry below for more on why this is done).

There is a limit of 512000 (MAX_MAP_SURFEDGES) surfedges per map. Note that the number of surfedges is not necessarily the same as the number of edges in the map.

Face and original face

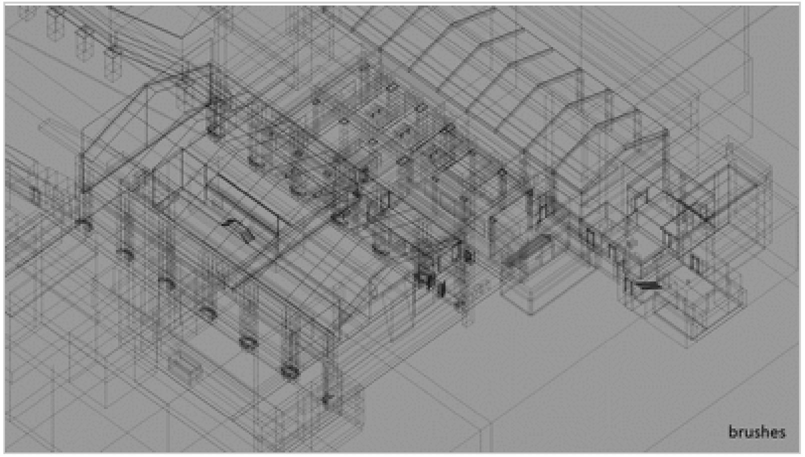
The face lump **(Lump 7)** contains the major geometry of the map, used by the game engine to render the viewpoint of the player. The face lump contains faces after they have undergone the BSP splitting process; they therefore do not directly correspond to the faces of brushes created in Hammer. Faces are always flat, convex polygons, though they can contain edges that are co-linear.

The face lump is one of the more complex structures of the map file. It is an array of `dface_t` entries, each 56 bytes long:

```
struct dface_t
{
    unsigned short  planenum;       // the plane number
    byte        side;           // faces opposite to the node's plane direction
    byte        onNode;         // 1 of on node, 0 if in leaf
    int     firstedge;      // index into surfedges
    short       numedges;       // number of surfedges
    short       texinfo;        // texture info
    short       dispinfo;       // displacement info
    short       surfaceFogVolumeID; // ?
    byte        styles[4];      // switchable lighting info
    int     lightofs;       // offset into lightmap lump
    float       area;           // face area in units^2
    int     LightmapTextureMinsInLuxels[2]; // texture lighting info
    int     LightmapTextureSizeInLuxels[2]; // texture lighting info
    int     origFace;       // original face this was split from
    unsigned short  numPrims;       // primitives
    unsigned short  firstPrimID;
    unsigned int    smoothingGroups;    // lightmap smoothing group
};
```

The first member `planenum` is the plane number, i.e., the index into the plane array that corresponds to the plane that is aligned with this face in the world. `Side` is zero if this plane faces in the same direction as the face (i.e. "out" of the face) or non-zero otherwise.

`Firstedge` is an index into the Surfedge array; this and the following `numedges` entries in the surfedge array define the edges of the face. As mentioned above, whether the value in the surfedge array is positive or negative indicates whether the corresponding pair of vertices listed in the Edge array should be traced from the first vertex to the second, or vice versa. The vertices which make up the face are thus referenced in clockwise order; when looking towards the face, each edge is traced in a clockwise direction. This makes rendering the faces easier, and allows quick culling of faces that face away from the viewpoint.



Animated x-ray view of `d1_trainstation_01` with the three types of geometry data. View in full size here.

`Texinfo` is an index into the Texinfo array (see below), and represents the texture to be drawn on the face. `Dispinfo` is an index into the Dispinfo array is the face is a displacement surface (in which case, the face defines the boundaries of the surface); otherwise, it is -1. `SurfaceFogVolumeID` appears to be related to drawing fogging when the player's viewpoint is underwater or looking through water.

`OrigFace` is the index of the original face which was split to produce this face. `NumPrims` and `firstPrimID` are related to the drawing of "Non-polygonal primitives" (see below). The other members of the structure are used to reference face-lighting info (see the Lighting lump, below).

The face array is limited to 65536 (MAX_MAP_FACES) entries.

The original face lump **(Lump 27)** has the same structure as the face lump, but contains the array of faces before the BSP splitting process is done. These faces are therefore closer to the original brush faces present in the precompile map than the face array, and there are less of them. The `origFace` entry for all original faces is zero. The maximum size of the original face array is also 65536 entries.

Both the face and original face arrays are culled; that is, many faces present before compilation of the map (primarily those that face towards the "void" outside the map) are removed from the array.

Brush and brushside

The brush lump **(Lump 18)** contains all brushes that were present in the original VMF file before compiling. Unlike faces, brushes are constructive solid geometry (CSG) defined by planes instead of edges and vertices. It is the presence of the brush and brushside lumps in Source BSP files that makes decompiling them a much easier job than for GoldSrc files, which lacked this info. The lump is an array of 12-byte `dbrush_t` structures:

```
struct dbrush_t
{
    int firstside;  // first brushside
    int numsides;   // number of brushsides
    int contents;   // contents flags
};
```

The first integer `firstside` is an index into the brushside array lump, this and the following `numsides` brushsides make up all the sides in this brush. The `contents` entry contains bitflags which determine the contents of this brush. The values are binary-ORed together, and are defined in the *public/bspflags.h* file:

Name	Value	Notes
`CONTENTS_EMPTY`	0	No contents
`CONTENTS_SOLID`	0x1	an eye is never valid in a solid
`CONTENTS_WINDOW`	0x2	translucent, but not watery (glass)
`CONTENTS_AUX`	0x4	
`CONTENTS_GRATE`	0x8	alpha-tested "grate" textures. Bullets/sight pass through, but solids don't
`CONTENTS_SLIME`	0x10	
`CONTENTS_WATER`	0x20	
`CONTENTS_MIST`	0x40	
`CONTENTS_OPAQUE`	0x80	block AI line of sight
`CONTENTS_TESTFOGVOLUME`	0x100	things that cannot be seen through (may be non-solid though)
`CONTENTS_UNUSED`	0x200	unused
`CONTENTS_UNUSED6`	0x400	unused
`CONTENTS_TEAM1`	0x800	per team contents used to differentiate collisions between players and objects on different teams
`CONTENTS_TEAM2`	0x1000	
`CONTENTS_IGNORE_NODRAW_OPAQUE`	0x2000	ignore `CONTENTS_OPAQUE` on surfaces that have `SURF_NODRAW`
`CONTENTS_MOVEABLE`	0x4000	hits entities which are `MOVETYPE_PUSH` (doors, plats, etc.)
`CONTENTS_AREAPORTAL`	0x8000	remaining contents are non-visible, and don't eat brushes
`CONTENTS_PLAYERCLIP`	0x10000	
`CONTENTS_MONSTERCLIP`	0x20000	
`CONTENTS_CURRENT_0`	0x40000	currents can be added to any other contents, and may be mixed
`CONTENTS_CURRENT_90`	0x80000	

`CONTENTS_CURRENT_180`	0x100000	
`CONTENTS_CURRENT_270`	0x200000	
`CONTENTS_CURRENT_UP`	0x400000	
`CONTENTS_CURRENT_DOWN`	0x800000	
`CONTENTS_ORIGIN`	0x1000000	removed before bsping an entity
`CONTENTS_MONSTER`	0x2000000	should never be on a brush, only in game
`CONTENTS_DEBRIS`	0x4000000	
`CONTENTS_DETAIL`	0x8000000	brushes to be added after vis leafs
`CONTENTS_TRANSLUCENT`	0x10000000	auto set if any surface has trans
`CONTENTS_LADDER`	0x20000000	
`CONTENTS_HITBOX`	0x40000000	use accurate hitboxes on trace

Some of these flags seem to be inherited from previous game engines and are not used in Source maps. They are also used to describe to contents of the map's leaves (see below). The CONTENTS_DETAIL flag is used to mark brushes that were in func_detail entities before compiling.

The brush array is limited to 8192 entries (MAX_MAP_BRUSHES).

The brushside lump **(Lump 19)** is an array of 8-byte structures:

```
struct dbrushside_t
{
    unsigned short  planenum;   // facing out of the leaf
    short       texinfo;    // texture info
    short       dispinfo;   // displacement info
    short       bevel;      // is the side a bevel plane?
};
```

`Planenum` is an index info the plane array, giving the plane corresponding to this brushside. `Texinfo` and `dispinfo` are references into the texture and displacement info lumps. `Bevel` is zero for normal brush sides, but 1 if the side is a bevel plane (which seem to be used for collision detection).

Unlike the face array, brushsides are not culled (removed) where they touch the void. Void-facing sides do however have their texinfo entry changed to the `tools/toolsnodraw` texture during the compile process. Note there is no direct way of linking brushes and brushsides and the corresponding face array entries which are used to render that brush. Brushsides are used by the engine to calculate all player physics collision with world brushes. (Vphysics objects use lump 29 instead.)

The maximum number of brushsides is 65536 (MAX_MAP_BRUSHSIDES). The maximum number of brushsides on a single brush is 128 (MAX_BRUSH_SIDES).

Node and leaf

The node array **(Lump 5)** and leaf array **(Lump 10)** define the Binary Space Partition (BSP) tree structure of the map. The BSP tree is used by the engine to quickly determine the location of the player's viewpoint with respect to the map geometry, and along with the visibility information (see below), to decide which parts of the map are to be drawn.

The nodes and leaves form a tree structure. Each leaf represents a defined volume of the map, and each node represents the volume which is the sum of all its child nodes and leaves further down the tree.

Each node has exactly two children, which can be either another node or a leaf. A child node has two further children, and so on until all branches of the tree are terminated with leaves, which have no children. Each node also references a plane in the plane array. When determining the player's viewpoint, the engine is trying to find which leaf the viewpoint falls inside. It first compares the coordinates of the point with the plane referenced in the headnode (Node 0). If the point is in front of the plane, it then moves to the first child of the node; otherwise, it moves to the second child. If the child is a leaf, then it has completed its task. If it is another node, it then performs the same check against the plane referenced in this node, and follows the children as before. It therefore traverses the BSP tree until it finds which leaf the viewpoint lies in. The leaves, then, completely divide up the map volume into a set of non-overlapping, convex volumes defined by the planes of their parent nodes.

For more information on how the BSP tree is constructed, see the article "BSP for dummies" (http://web.archive.org/web/20050426034532/http://www.planetquake.com/qxx/bsp/).

The node array consists of 32-byte structures:

```
struct dnode_t
{
    int     planenum;   // index into plane array
    int     children[2];    // negative numbers are -(leafs + 1), not nodes
    short       mins[3];    // for frustum culling
    short       maxs[3];
    unsigned short  firstface;  // index into face array
    unsigned short  numfaces;   // counting both sides
    short       area;       // If all leaves below this node are in the same area, then
            // this is the area index. If not, this is -1.
    short       paddding;   // pad to 32 bytes length
};
```

`Planenum` is the entry in the plane array. The `children[]` members are the two children of this node; if positive, they are node indices; if negative, the value (-1-*child*) is the index into the leaf array (e.g., the value -100 would reference leaf 99).

The members `mins[]` and `maxs[]` are coordinates of a rough bounding box surrounding the contents of this node. The `firstface` and `numfaces` are indices into the face array that show which map faces are contained in this node, or zero if none are. The `area` value is the map area of this node (see below). There can be a maximum of 65536 nodes in a map (MAX_MAP_NODES).

The leaf array is an array with 56 bytes per element:

```
struct dleaf_t
{
    int         contents;       // OR of all brushes (not needed?)
    short           cluster;        // cluster this leaf is in
    short           area:9;         // area this leaf is in
    short           flags:7;        // flags
    short           mins[3];        // for frustum culling
    short           maxs[3];
    unsigned short      firstleafface;      // index into leaffaces
    unsigned short      numleaffaces;
    unsigned short      firstleafbrush;     // index into leafbrushes
    unsigned short      numleafbrushes;
    short           leafWaterDataID;    // -1 for not in water

    //!!! NOTE: for maps of version 19 or lower uncomment this block
    /*
    CompressedLightCube ambientLighting;    // Precaculated light info for entities.
    short           padding;        // padding to 4-byte boundary
    */
};
```

The leaf structure has similar contents to the node structure, except it has no children and no reference plane. Additional entries are the `contents` flags (see the brush lump, above), which shows the contents of any brushes in the leaf, and the `cluster` number of the leaf (see below). The `area` and `flags` members share a 16-bit bitfield and contain the area number and flags relating to the leaf. `Firstleafface` and `numleaffaces` index into the leafface array and show which faces are inside this leaf, if any. `Firstleafbrush` and `numleafbrushes` likewise index brushes inside this leaf through the leafbrush array.

The `ambientLighting` element is related to lighting of objects in the leaf, and consists of a CompressedLightCube structure, which is 24 bytes in length. Version 17 BSP files have a modified dleaf_t structure that omits the ambient lighting data, making the entry for each leaf only 32 bytes in length. The same shortened structure is also used for version 20 BSP files, with the ambient lighting information for LDR and HDR probably contained in the new lumps 55 and 56.

All leaves are convex polyhedra, and are defined by the planes of their parent nodes. They do not overlap. Any point in the coordinate space is in one and only one leaf of the map. A leaf which is not filled with a solid brush and can be entered by the player in the usual course of the game has a cluster number set; this is used in conjunction with the visibility information (below).

There are usually multiple, unconnected BSP trees in a map. Each one corresponds to an entry in model array (see below) and the headnode of each tree is referenced there. The first tree is the worldspawn model, the overall geometry of the level. Successive trees are the models of each brush entity in the map.

The creation of the BSP tree is done by the VBSP program, during the first phase of map compilation. Exactly how the tree is created, and how the map is divided into leaves, can be influenced by the map author by the use of HINT brushes, func_details, and the careful layout of all brushes in the map.

LeafFace and LeafBrush

The leafface lump **(Lump 16)** is an array of unsigned shorts which are used to map from faces referenced in the leaf structure to indices in the face array. The leafbrush lump (also an array of unsigned shorts)**(Lump 17)** does the same thing for brushes referenced in leaves. Their maximum sizes are both 65536 entries (MAX_MAP_LEAFFACES, MAX_MAP_LEAFBRUSHES).

Textures

The texture information in a map is split across a number of different lumps. The Texinfo lump is the most fundamental, referenced by the face and brushside arrays, and it in turn references the other texture lumps.

Texinfo

The texinfo lump **(Lump 6)** contains an array of `texinfo_t` structures:

```
struct texinfo_t
{
    float   textureVecs[2][4];  // [s/t][xyz offset]
    float   lightmapVecs[2][4]; // [s/t][xyz offset] - length is in units of texels/area
    int flags;          // miptex flags overrides
    int texdata;        // Pointer to texture name, size, etc.
}
```

Each texinfo is 72 bytes long.

The first array of floats is in essence two vectors that represent how the texture is orientated and scaled when rendered on the world geometry. The two vectors, **s** and **t**, are the mapping of the left-to-right and down-to-up directions in the texture pixel coordinate space, onto the world. Each vector has an x, y, and z component, plus an offset which is the "shift" of the texture in that direction relative to the world. The length of the vectors represent the scaling of the texture in each direction.

The 2D coordinates (*u, v*) of a texture pixel (or *texel*) are mapped to the world coordinates (*x, y, z*) of a point on a face by:

$u = tv_{0,0} * x + tv_{0,1} * y + tv_{0,2} * z + tv_{0,3}$

$v = tv_{1,0} * x + tv_{1,1} * y + tv_{1,2} * z + tv_{1,3}$

(ie. The dot product of the vectors with the vertex plus the offset in that direction. Where $tv_{A,B}$ is `textureVecs[A][B]`.

Furthermore, after calculating (u, v), to convert them to texture coordinates which you would send to your graphics card, divide u and v by the width and height of the texture respectively.

The `lightmapVecs` float array performs a similar mapping of the lightmap samples of the texture onto the world.

The `flags` entry contains bitflags which are defined in *bspflags.h*:

Name	Value	Notes
`SURF_LIGHT`	0x1	value will hold the light strength
`SURF_SKY2D`	0x2	don't draw, indicates we should skylight + draw 2d sky but not draw the 3D skybox
`SURF_SKY`	0x4	don't draw, but add to skybox
`SURF_WARP`	0x8	turbulent water warp
`SURF_TRANS`	0x10	texture is translucent
`SURF_NOPORTAL`	0x20	the surface can not have a portal placed on it
`SURF_TRIGGER`	0x40	FIXME: This is an xbox hack to work around elimination of trigger surfaces, which breaks occluders
`SURF_NODRAW`	0x80	don't bother referencing the texture
`SURF_HINT`	0x100	make a primary bsp splitter
`SURF_SKIP`	0x200	completely ignore, allowing non-closed brushes
`SURF_NOLIGHT`	0x400	Don't calculate light
`SURF_BUMPLIGHT`	0x800	calculate three lightmaps for the surface for bumpmapping
`SURF_NOSHADOWS`	0x1000	Don't receive shadows
`SURF_NODECALS`	0x2000	Don't receive decals
`SURF_NOCHOP`	0x4000	Don't subdivide patches on this surface
`SURF_HITBOX`	0x8000	surface is part of a hitbox

The flags seem to be derived from the texture's .vmt file contents, and specify special properties of that texture.

Texdata

Finally the `texdata` entry is an index into the Texdata array, and specifies the actual texture.

The index of a Texinfo (referenced from a face or brushside) may be given as -1; this indicates that no texture information is associated with this face. This occurs on compiling brush faces given the SKIP, CLIP, or INVISIBLE type textures in the editor.

The texdata array **(Lump 2)** consists of the structures:

```
struct dtexdata_t
{
    Vector  reflectivity;       // RGB reflectivity
    int nameStringTableID;  // index into TexdataStringTable
    int width, height;      // source image
    int view_width, view_height;
};
```

The `reflectivity` vector corresponds to the RGB components of the reflectivity of the texture, as derived from the material's .vtf file. This is probably used in radiosity (lighting) calculations of what light bounces from the texture's surface. The `nameStringTableID` is an index into the TexdataStringTable array (below). The other members relate to the texture's source image.

TexdataStringData and TexdataStringTable

The TexdataStringTable **(Lump 44)** is an array of integers which are offsets into the TexdataStringData (lump 43). The TexdataStringData lump consists of concatenated null-terminated strings giving the texture name.

There can be a maximum of 12288 texinfos in a map (MAX_MAP_TEXINFO). There is a limit of 2048 texdatas in the array (MAX_MAP_TEXDATA) and up to 256000 bytes in the TexdataStringData data block (MAX_MAP_TEXDATA_STRING_DATA). Texture name strings are limited to 128 characters (TEXTURE_NAME_LENGTH).

Model

A Model, in the terminology of the BSP file format, is a collection of brushes and faces, often called a "bmodel". It should not be confused with the prop models used in Hammer, which are usually called "studiomodels" in the SDK source.

The model lump **(Lump 14)** consists of an array of 48-byte `dmodel_t` structures:

```
struct dmodel_t
{
    Vector  mins, maxs;     // bounding box
    Vector  origin;         // for sounds or lights
    int headnode;       // index into node array
    int firstface, numfaces;    // index into face array
};
```

`Mins` and `maxs` are the bounding points of the model. `Origin` is the coordinates of the model in the world, if set. `Headnode` is the index of the top node in the node array of the BSP tree which describes this model. `Firstface` and `numfaces` index into the face array and give the faces which make up this model.

The first model in the array (Model 0) is always "worldspawn", the overall geometry of the whole map excluding entities (but including func_detail brushes). The subsequent models in the array are associated with brush entities, and referenced from the entity lump.

There is a limit of 1024 models in a map (MAX_MAP_MODELS), including the worldspawn model zero.

Visibility

The visibility lump **(Lump 4)** is in a somewhat different format to the previously mentioned lumps. To understand it, some discussion of how the Source engine's visibility system works in necessary.

As mentioned in the Node and leaf lumps section above, every point in the map falls into exactly one convex volume called a leaf. All leaves that are on the inside of the map (not touching the void), and that are not covered by a solid brush can potentially have the player's viewpoint inside it during normal gameplay. Each of these enterable leaves (also called *visleaves*) gets assigned a cluster number. In Source BSP files, each enterable leaf corresponds to just one cluster.

(The terminology is slightly confusing here. According to the "Quake 2 BSP File Format" article, in the Q2 engine there could be multiple adjacent leaves in each cluster - thus the cluster is so called because it is a cluster of leaves. It seems from the SDK source that this situation may also occur during the compilation of Source maps; however, after the VVIS compile process is finished these adjacent leaves (and their parent nodes) are generally merged into a single leaf. In older Source maps (prior to Counter-Strike: Global Offensive) it seems there is only ever one leaf per cluster. However, in some CS:GO maps multiple leaves can belong to the same cluster in the final compiled BSP. This seems to occur particularly in the 3D skybox of most CS:GO maps, and can be seen in the main playable area of more recently refurbished maps such as de_cbble and de_nuke.)

Each cluster, then, is a volume in the map that the player can potentially be in. To render the map quickly, the game engine draws the geometry of only those clusters which are visible from the current cluster. Clusters which are completely occluded from view from the player's cluster need not be drawn. Calculating cluster-to-cluster visibility is the responsibility of the VVIS compile tool, and the resulting data is stored in the Visibility lump.

Once the engine knows a cluster is visible, the leaf data references all faces present in that cluster, allowing the contents of the cluster to be rendered.

The data is stored as an array of bit-vectors; for each cluster, a list of which other clusters are visible from it are stored as individual bits (1 if visible, 0 if occluded) in an array, with the *n*th bit position corresponding to the *n*th cluster. This is known as the cluster's Potentially Visible Set (PVS). Because of the large size of this data, the bit vectors are compressed by run-length encoding groups of zero bits in each vector.

There is also a Potentially Audible Set (PAS) array created for each cluster; this marks which clusters can hear sounds occurring in other clusters. The PAS seems to be created by merging the PVS bits of all clusters in current cluster's PVS.

The Visibilty lump is defined as:

```
struct dvis_t
{
	int numclusters;
	int byteofs[numclusters][2]
};
```

The first integer is the number of clusters in the map. It is followed by an array of integers giving the byte offset from the start of the lump to the start of the PVS bit array for each cluster, followed by the offset to the PAS array. Immediately following the array are the compressed bit vectors.

The decoding of the run-length compression works as follows: To find the PVS of a given cluster, start at the byte given by the offset in the `byteofs[]` array. If the current byte in the PVS buffer is zero, the following byte multiplied by 8 is the number of clusters to skip that are not visible. If the current byte is non-zero, the bits that are set correspond to clusters that are visible from this cluster. Continue until the number of clusters in the map is reached.

Example C code to decompress the bit vectors can be found in the "Quake 2 BSP File Format" document.

The maximum size of the Visibility lump is 0x1000000 bytes (MAX_MAP_VISIBILITY); that is, 16 Mb.

Entity

See also: Patching levels with lump files

The entity lump **(Lump 0)** is an ASCII text buffer which stores the entity data in a format very similar to the KeyValue format of uncompiled VMF files. Its general form is:

```
{
	"world_maxs" "480 480 480"
	"world_mins" "-480 -480 -224"
	"maxpropscreenwidth" "-1"
	"skyname" "sky_wasteland02"
	"classname" "worldspawn"
}
{
	"origin" "-413.793 -384 -192"
	"angles" "0 0 0"
	"classname" "info_player_start"
}
{
	"model" "*1"
	"targetname" "secret_1"
	"origin" "424 -1536 1800"
	"Solidity" "1"
	"StartDisabled" "0"
	"InputFilter" "0"
	"disablereceiveshadows" "0"
	"disableshadows" "0"
	"rendermode" "0"
	"renderfx" "0"
	"rendercolor" "255 255 255"
	"renderamt" "255"
	"classname" "func_brush"
}
```

Entities are defined between opening and closing braces (`{` and `}`) and list on each line a pair of key/value properties inside quotation marks. The first entity is always worldspawn. The `classname` property gives the entity type, and the `targetname` property gives the entity's name as defined in Hammer (if it has one). The `model` property is slightly special if it starts with an asterisk (*), the following number is an index into the model array (see above) which corresponds to a brush entity 'model'. Otherwise, the value contains the name of a compiled model. Other key/value pairs correspond to the properties of the entity as set in Hammer.

Note: Some entities (including func_detail, env_cubemap, info_overlay and prop_static) are internal, and are stripped from the entity lump during the compile process, normally because they are absorbed by the world.

Depending on the Source engine version, the entity lump can contain 4096 (Source 2004) to 16384 (Alien Swarm) entities (`MAX_MAP_ENTITIES`). These limits are independent from the engine's actual entity limit. Each key can be a maximum of 32 characters (`MAX_KEY`) and the value up to 1024 characters (`MAX_VALUE`).

Game lump

The Game lump **(Lump 35)** seems to be intended to be used for map data that is specific to a particular game using the Source engine, so that the file format can be extended without altering the previously defined format. It starts with a game lump header:

```
struct dgamelumpheader_t
{
    int lumpCount;  // number of game lumps
    dgamelump_t gamelump[lumpCount];
};
```

where the gamelump directory array is defined by:

```
struct dgamelump_t
{
    int     id;     // gamelump ID
    unsigned short  flags;      // flags
    unsigned short  version;    // gamelump version
    int     fileofs;    // offset to this gamelump
    int     filelen;    // length
};
```

The gamelump is identified by the 4-byte `id` member, which defines what data is stored in it, and the byte position of the data and its length is given in `fileofs` and `filelen`. Note that `fileofs` is relative to the beginning of the BSP file, not to the game lump offset. One exception is the console version of Portal 2, where `fileofs` is relative to the game lump offset, as one would expect.

Static props

Of interest is the gamelump which is used to store prop_static entities, which uses the gamelump ID of 'sprp' ASCII (1936749168 decimal). Unlike most other entities, static props are not stored in the entity lump. The gamelump formats used in Source are defined in the *public/gamebspfile.h* header file.

The first element of the static prop game lump is the dictionary; this is an integer count followed by the list of model (prop) names used in the map:

```
struct StaticPropDictLump_t
{
    int dictEntries;
    char    name[dictEntries];  // model name
};
```

Each `name` entry is 128 characters long, null-padded to this length.

Following the dictionary is the leaf array:

```
struct StaticPropLeafLump_t
{
    int leafEntries;
    unsigned short  leaf[leafEntries];
};
```

Presumably, this array is used to index into the leaf lump to locate the leaves that each prop static is located in. Note that a prop static may span several leaves.

Next, an integer giving the number of `StaticPropLump_t` entries, followed by that many structures themselves:

```
struct StaticPropLump_t
{
    // v4
    Vector          Origin;            // origin
    QAngle          Angles;            // orientation (pitch roll yaw)

    // v4
    unsigned short  PropType;          // index into model name dictionary
    unsigned short  FirstLeaf;         // index into leaf array
    unsigned short  LeafCount;
    unsigned char   Solid;             // solidity type
    unsigned char   Flags;
    int             Skin;              // model skin numbers
    float           FadeMinDist;
    float           FadeMaxDist;
    Vector          LightingOrigin;    // for lighting
    // since v5
    float           ForcedFadeScale;   // fade distance scale
    // v6 and v7 only
    unsigned short  MinDXLevel;        // minimum DirectX version to be visible
    unsigned short  MaxDXLevel;        // maximum DirectX version to be visible
    // since v8
    unsigned char   MinCPULevel;
    unsigned char   MaxCPULevel;
    unsigned char   MinGPULevel;
    unsigned char   MaxGPULevel;
    // since v7
    color32         DiffuseModulation; // per instance color and alpha modulation
    // v9 and v10 only
    bool            DisableX360;       // if true, don't show on XBox 360 (4-bytes long)
    // since v10
    unisgned int    FlagsEx;           // Further bitflags.
    // since v11
    float           UniformScale;      // Prop scale
};
```

The coordinates of the prop are given by the `Origin` member; its orientation (pitch, roll, yaw) is given by the `Angles` entry, which is a 3-float vector. The `PropType` element is an index into the dictionary of prop model names, given above. The other elements correspond to the location of the prop in the BSP structure of the map, its lighting, and other entity properties as set in Hammer. The other elements (`ForcedFadeScale`, etc.) are only present in the static prop structure if the gamelump's specified version is high enough (see `dgamelump_t.version`); both version 4 and version 5 static prop gamelumps are used in official HL2 maps. Version 6 has been encountered in TF2. Version 7 is used in some Left 4 Dead maps, and a modified version 7 is present in Zeno Clash maps. Version 8 is used predominantly in Left 4 Dead, and version 9 in Left 4 Dead 2. The new version 10 appears in Tactical Intervention. Version 11 is used in Counter-Strike: Global Offensive since the addition of uniform prop scaling (before this it was version 10). After version 7, DX level options were removed. In version 11 XBox 360 flags were removed.

Other

Other gamelumps used in Source BSP files are the detail prop gamelump (`dprp`), and the detail prop lighting lump (`dplt` for LDR and `dplh` for HDR). These are used for prop_detail entities (grass tufts, etc.) automatically emitted by certain textures when placed on displacement surfaces.

There does not seem to be a specified limit on the size of the game lump.

Displacements

Displacement surfaces are the most complex parts of a BSP file, and only part of their format is covered here. Their data is split over a number of different data lumps in the file, but the fundamental reference to them is through the dispinfo lump **(Lump 26)**. Dispinfos are referenced from the face, original face, and brushside arrays.

DispInfo

```
struct ddispinfo_t
{
    Vector          startPosition;      // start position used for orientation
    int         DispVertStart;      // Index into LUMP_DISP_VERTS.
    int         DispTriStart;       // Index into LUMP_DISP_TRIS.
    int         power;          // power - indicates size of surface (2^power   1)
    int         minTess;        // minimum tesselation allowed
    float           smoothingAngle;     // lighting smoothing angle
    int         contents;       // surface contents
    unsigned short      MapFace;        // Which map face this displacement comes from.
    int         LightmapAlphaStart; // Index into ddisplightmapalpha.
    int         LightmapSamplePositionStart;    // Index into LUMP_DISP_LIGHTMAP_SAMPLE_POSITIONS.
    CDispNeighbor       EdgeNeighbors[4];   // Indexed by NEIGHBOREDGE_ defines.
    CDispCornerNeighbors    CornerNeighbors[4]; // Indexed by CORNER_ defines.
    unsigned int        AllowedVerts[10];   // active verticies
};
```

The structure is 176 bytes long. The `startPosition` element is the coordinates of the first corner of the displacement. `DispVertStart` and `DispTriStart` are indices into the DispVerts and DispTris lumps. The `power` entry gives the number of subdivisions in the displacement surface - allowed values are 2, 3 and 4, and these correspond to 4, 8 and 16 subdivisions on each side of the displacement surface. The structure also references any neighbouring displacements on the sides or the corners of this displacement through the `EdgeNeighbors` and `CornerNeighbors` members. There are complex rules governing the order that these neighbour displacements are given; see the comments in *bspfile.h* for more. The `MapFace` value is an index into the face array and is face that was turned into a displacement surface. This face is used to set the texture and overall physical location and boundaries of the displacement.

DispVerts

The DispVerts lump **(Lump 33)** contains the vertex data of the displacements. It is given by:

```
struct dDispVert
{
    Vector  vec;    // Vector field defining displacement volume.
    float   dist;   // Displacement distances.
    float   alpha;  // "per vertex" alpha values.
};
```

where `vec` is the normalized vector of the offset of each displacement vertex from its original (flat) position; `dist` is the distance the offset has taken place; and `alpha` is the alpha-blending of the texture at that vertex.

A displacement of power *p* references (2^*p* 1)^2 dispverts in the array, starting from the `DispVertStart` index.

DispTris

The DispTris lump **(Lump 48)** contains "triangle tags" or flags related to the properties of a particular triangle in the displacement mesh:

```
struct dDispTri
{
    unsigned short Tags;    // Displacement triangle tags.
};
```

where the flags are:

Name	Value

`DISPTRI_TAG_SURFACE`	0x1
`DISPTRI_TAG_WALKABLE`	0x2
`DISPTRI_TAG_BUILDABLE`	0x4
`DISPTRI_FLAG_SURFPROP1`	0x8
`DISPTRI_FLAG_SURFPROP2`	0x10

There are 2x(2^*p*)^2 DispTri entries for a displacement of power *p*. They are presumably used to indicate properties for each triangle of the displacement such as whether the surface is walkable at that point (not too steep to climb).

There are a limit of 2048 Dispinfos per map, and the limits of DispVerts and DispTris are such that all 2048 displacements could be of power 4 (maximally subdivided).

Other displacement-related data are the DispLightmapAlphas **(Lump 32)** and DispLightmapSamplePos **(Lump 34)** lumps, which seem to relate to lighting of each displacement surface.

Pakfile

The Pakfile lump **(Lump 40)** is a special lump that can contains multiple files which are embedded into the bsp file. Usually, they contain special texture (.vtf) and material (.vmt) files which are used to store the reflection maps from env_cubemap entities in the map; these files are built and placed in the Pakfile lump when the `buildcubemaps` console command is executed. The Pakfile can optionally contain such things as custom textures and prop models used in the map, and are placed into the bsp file by using the BSPZIP program (or alternate programs such as Pakrat). These files are integrated into the game engine's file system and will be loaded preferentially before externally located files are used.

The format of the Pakfile lump is identical to that used by the Zip compression utility when no compression is specified (i.e., the individual files are stored in uncompressed format). If the Pakfile lump is extracted and written to a file, it can therefore be opened with WinZip and similar programs.

The header *public/zip_uncompressed.h* defines the structures present in the Pakfile lump. The last element in the lump is a `ZIP_EndOfCentralDirRecord` structure. This points to an array of `ZIP_FileHeader` structures immediately preceeding it, one for each file present in the Pak. Each of these headers then point to `ZIP_LocalFileHeader` structures that are followed by that file's data.

The Pakfile lump is usually the last element of the bsp file.

Cubemap

The Cubemap lump **(Lump 42)** is an array of 16-byte `dcubemapsample_t` structures:

```
struct dcubemapsample_t
{
    int     origin[3];  // position of light snapped to the nearest integer
    int         size;       // resolution of cubemap, 0 - default
};
```

The `dcubemapsample_t` structure defines the location of a env_cubemap entity in the map. The `origin` member contains integer x,y,z coordinates of the cubemap, and the `size` member is resolution of the cubemap, specified as 2^(*size*-1) pixels square. If set as 0, the default size of 6 (32x32 pixels) is used. There can be a maximum of 1024 (MAX_MAP_CUBEMAPSAMPLES) cubemaps in a file.

When the "buildcubemaps" console command is performed, six snapshots of the map (one for each direction) are taken at the location of each env_cubemap entity. These snapshots are stored in a multi-frame texture (vtf) file, which is added to the Pakfile lump (see above). The textures are named `cX_Y_Z.vtf`, where (X,Y,Z) are the (integer) coordinates of the corresponding cubemap.

Faces containing materials that are environment mapped (e.g. shiny textures) reference their assigned cubemap through their material name. A face with a material named (e.g.) `walls/shiny.vmt` is altered (new Texinfo & Texdata entries are created) to refer to a renamed material `maps/`*`mapname`*`/walls/shiny_X_Y_Z.vmt`, where (X,Y,Z) are the cubemap coordinates as before. This .vmt file is also stored in the Pakfile, and references the cubemap .vtf file through its `$envmap` property.

Version 20 files contain extra `cX_Y_Z.hdr.vtf` files in the Pakfile lump, containing HDR texture files in RGBA16161616F (16-bit per channel) format.

Overlay

Unlike the simpler decals (infodecal entities), info_overlays are removed from the entity lump and stored separately in the Overlay lump **(Lump 45)**. The structure is reflects the properties of the entity in Hammer almost exactly:

```
struct doverlay_t
{
    int     Id;
    short       TexInfo;
    unsigned short  FaceCountAndRenderOrder;
    int     Ofaces[OVERLAY_BSP_FACE_COUNT];
    float       U[2];
    float       V[2];
    Vector      UVPoints[4];
    Vector      Origin;
    Vector      BasisNormal;
};
```

The `FaceCountAndRenderOrder` member is split into two parts; the lower 14 bits are the number of faces that the overlay appears on, with the top 2 bits being the render order of the overlay (for overlapping decals). The `Ofaces` array, which is 64 elements in size (OVERLAY_BSP_FACE_COUNT) are the indices into the face array indicating which map faces the overlay should be displayed on. The other elements set the texture, scale, and orientation of the overlay decal. There can be a maximum of 512 overlays per file (MAX_MAP_OVERLAYS). In Dota 2 this limit on the number of overlays is increased significantly.

Lighting

The lighting lump **(Lump 8)** is used to store the static lightmap samples of map faces. Each lightmap sample is a colour tint that multiplies the colours of the underlying texture pixels, to produce lighting of varying intensity. These lightmaps are created during the VRAD phase of map compilation and are referenced from the `dface_t` structure. The current lighting lump version is 1.

Each `dface_t` may have a up to four lightstyles defined in its `styles[]` array (which contains 255 to represent no lightstyle). The number of luxels in each direction of the face is given by the two `LightmapTextureSizeInLuxels[]` members (plus 1), and the total number of luxels per face is thus:

```
(LightmapTextureSizeInLuxels[0] + 1) * (LightmapTextureSizeInLuxels[1] + 1)
```

Each face gives a byte offset into the lighting lump in its `lightofs` member (if no lighting information is used for this face e.g. faces with skybox, nodraw and invisible textures, `lightofs` is -1.) There are (*number of lightstyles*)*(*number of luxels*) lightmap samples for each face, where each sample is a 4-byte ColorRGBExp32 structure:

```
struct ColorRGBExp32
{
    byte r, g, b;
    signed char exponent;
};
```

Standard RGB format can be obtained from this by multiplying each colour component by 2^(*exponent*). For faces with bumpmapped textures, there are four times the usual number of lightmap samples, presumably containing samples used to compute the bumpmapping.

Immediately preceeding the `lightofs`-referenced sample group, there are single samples containing the average lighting on the face, one for each lightstyle, in reverse order from that given in the `styles[]` array.

Version 20 BSP files contain a second, identically sized lighting lump **(Lump 53)**. This is presumed to store more accurate (higher-precision) HDR data for each lightmap sample. The format is currently unknown, but is also 32 bits per sample.

The maximum size of the lighting lump is 0x1000000 bytes, i.e. 16 Mb (MAX_MAP_LIGHTING).

Ambient Lighting

The ambient lighting lumps **(Lumps 55 and 56)** are present in BSP version 20 and later. Lump 55 is used for HDR lighting, and Lump 56 is used for LDR lighting. These lumps are used to store the volumetric ambient lighting information in each leaf (i.e. lighting information for entities such as NPCs, the viewmodel, and non-static props). Prior to version 20, this data was stored in the leaf lump (Lump 10), in the `dleaf_t` structure, with far less precision than this newer lump allows.

Both ambient lighting lumps are arrays of `dleafambientlighting_t` structures:

```
struct dleafambientlighting_t
{
    CompressedLightCube cube;
    byte x;     // fixed point fraction of leaf bounds
    byte y;     // fixed point fraction of leaf bounds
    byte z;     // fixed point fraction of leaf bounds
    byte pad;   // unused
};
```

Each leaf is associated with a number of these `dleafambientlighting_t` structures, each of which contains a cube of ambient light data at the position specified by the `x`, `y`, and `z` members. These coordinates are stored as fractions of the leaf's bounding box, i.e. a value of 0 for `x` represents the westmost position in the leaf, a value of 255 represents the eastmost position in the leaf, and a value of 128 represents the center of the leaf.

The lighting data for each sample is represented by a `CompressedLightCube` structure, which is simply an array of 6 `ColorRGBExp32` structures as described in the previous section:

```
struct CompressedLightCube
{
    ColorRGBExp32 m_Color[6];
};
```

Each lighting sample in the array corresponds to the amount of lighting being received from each cardinal direction in 3D space.

At compile time, VRAD will randomly generate a number of ambient light sample positions for each leaf, and store the lighting information at each sample point in a `dleafambientlighting_t` structure. To associate each leaf with its collection of ambient samples, the ambient lighting index lumps **(Lumps 51 and 52)** are used. Lump 51 stores ambient light index information for HDR, and lump 52 stores the same information for LDR.

The ambient light index lumps are arrays of `dleafambientindex_t` structures:

```
struct dleafambientindex_t
{
    unsigned short ambientSampleCount;
    unsigned short firstAmbientSample;
};
```

The Nth `dleafambientindex_t` structure in this array always corresponds to the Nth leaf in the `dleaf_t` array. The `ambientSampleCount` field is the number of ambient samples associated with the corresponding leaf, and `firstAmbientSample` is an index into the ambient lighting array, referring to the first sample in the array that is associated with this leaf.

Occlusion

The occlusion lump **(Lump 9)** contains the polygon geometry and some flags used by func_occluder entities. Unlike other brush entities, func_occluders don't use the 'model' key in the entity lump. Instead, the brushes are split from the entities during the compile process and numeric occluder keys are assigned as 'occludernum'. Brush sides textured with `tools/toolsoccluder` or `tools/toolstrigger` are then stored together with the occluder keys and some additional info in this lump.

The lump is divided into three parts and begins with a integer value with the total number of occluders, followed by an array of `doccluderdata_t` fields of the same size. The next part begins with another integer value, this time for the total number of occluder polygons, as well as an array of `doccluderpolydata_t` fields of equal size. Part three begins with another integer value for the amount of occluder polygon vertices, followed by an array of integer values for the vertex indices, again of the same size.

```
struct doccluder_t
{
    int         count;
    doccluderdata_t     data[count];
    int         polyDataCount;
    doccluderpolydata_t polyData[polyDataCount];
    int         vertexIndexCount;
    int         vertexIndices[vertexIndexCount];
};
```

The `doccluderdata_t` structure contains flags and dimensions of the occluder, as well as the area where it remains. `firstpoly` is the first index into the `doccluderpolydata_t` with a total of `polycount` entries.

```
struct doccluderdata_t
{
    int flags;
    int firstpoly;  // index into doccluderpolys
    int polycount;  // amount of polygons
    Vector  mins;           // minima of all vertices
    Vector  maxs;           // maxima of all vertices
    // since v1
    int area;
};
```

Occluder polygons are stored in the `doccluderpolydata_t` structure and contain the `firstvertexindex` field, which is the first index into the vertex array of the occluder, which are again indices for the vertex array of the vertex lump **(Lump 3)**. The total number of vertex indices is stored in `vertexcount`.

```
struct doccluderpolydata_t
{
    int firstvertexindex;   // index into doccludervertindices
    int vertexcount;        // amount of vertex indices
    int planenum;
};
```

Physics

The physcollide lump **(Lump 29)** contains physics data for the world.

The lump consists of a sequence of *models*, where each model consists of:

- a dphysmodel_t header

```
struct dphysmodel_t
{
    int modelIndex;  // Perhaps the index of the model to which this physics model applies?
    int dataSize;    // Total size of the collision data sections
    int keydataSize; // Size of the text section
    int solidCount;  // Number of collision data sections
};
```

- a series of collision data sections (including compactsurfaceheader_t header)
- a text section

The lump is terminated by a dphysmodel_t structure with the modelIndex set to -1.

The last two parts appear to be identical to the PHY file format, which means their exact contents are unknown. Note that the compactsurfaceheader_t structure contains the data size of each collision data section (including the rest of the header), so the lump can be parsed as follows:

```
while(true) {
    header = readHeader();
    if(header.modelIndex == -1)
        break;

    for(int k = 0; k < header.solidCount; k++) {
        size = read4ByteInt();
        collisionData = readBytes(size);
    }

    textData = readBytes(header.keydataSize);
}
```

Other

To do: Some of these information are likely guesses and need further research.

- The Worldlights lump **(Lump 15)** contains information on each static light entity in the world, and seems to be used to provide semi-dynamic lighting for moving entities.
- The Areas lump **(Lump 20)** references the Areaportals lump **(Lump 21)** and is used with func_areaportal and func_areaportalwindow entities to define sections of the map that can be switched to render or not render.
- The Portals **(Lump 22)**, Clusters **(Lump 23)**, PortalVerts **(Lump 24)**, ClusterPortals **(Lump 25)**, and ClipPortalVerts **(Lump 41)** lumps are used by the VVIS phase of the compile to ascertain which clusters can see which other clusters. A cluster is a player-enterable leaf volume in the map (see above). A "portal" is a polygon boundary between a cluster or leaf and an adjacent cluster or leaf. Most of this information is also used by the VRAD program to calculate static lighting, and then is removed from the bsp file.
- PhysCollide Lumps **(Lump 29)** and PhysCollideSurface **(Lump 49)** lumps seem to be related to the physical simulation of entity collisions in the game engine.
- The VertNormal **(Lump 30)** and VertNormalIndices **(Lump 31)** lumps may be related to smoothing of lightmaps on faces.
- The FaceMacroTextureInfo lump **(Lump 47)** is a short array containing the same number of members as the number of faces in the map. If the entry for a face contains anything other than `-1` (0xFFFF), it is an index of a texture name in the TexDataStringTable. In VRAD, the corresponding texture is mapped onto the world extents, and used to modulate the lightmaps of that face. There is also a base macro texture (located at `materials/macro/`*`mapname`*`/base.vtf`) that is applied to all faces if found. Only maps in VTMB seem to make any use of macro textures.
- LeafWaterData **(Lump 36)** and LeafMinDistToWater **(Lump 46)** lumps may be used to determine player position with respect to water volumes.
- The Primitives **(Lump 37)**, PrimVerts **(Lump 38)** and PrimIndices **(Lump 39)** lumps are used in reference to "non-polygonal primitives". They are also sometimes called "waterstrips", "waterverts" and "waterindices" in the SDK Source, since they were originally only used to subdivide water meshes. They are now used to prevent the appearance of cracks between adjacent faces, if the face edges contain a "T-junction" (a vertex collinearly between two other vertices). The PrimIndices lump defines a set of triangles between face vertices, that tessellate the face. They are referenced from the Primatives lump, which is in turn referenced by the face lump data. Current maps do not seem to use the PrimVerts lump at all. (Ref. (http://web.archive.org/web/20071110230828/http://www.chatbear.com/board.plm?a=viewthread&b=4991&t=137,1118051039,3296&s=0&id=862840))
- Version 20 files containing HDR lighting information have four extra lumps, the contents of which are currently uncertain. Lump 53 is always the same size as the standard lighting lump **(Lump 8)** and probably contains higher-precision data for each lightmap sample. Lump 54 is the same size as the worldlight lump **(Lump 15)** and presumably contains HDR-related data for each light entity.

Retrieved from "https://developer.valvesoftware.com/w/index.php?title=Source_BSP_File_Format&oldid=223416"

Category: File formats

- This page was last modified on 16 March 2019, at 02:18.