Valve Map Format

From Valve Developer Community

The **Valve Map Format** (**VMF**) is a proprietary format that stores raw (pre-compile) map data, used by the Valve Hammer Editor to save production-stage maps and prefabs. It contains information on all map brushes and entities in a keyvalue format stored with a ".vmf" extension.

A .vmf file is the source code of a map designed for use in the engine. Like any source file it must be compiled to be used and is thus designed to be easy to read and work with rather than optimized for execution. Because of this a .vmf file is written in an easy to understand form of coding language common in dealings with the source engine. This is to be the documentation of everything defined within a .vmf file detailing what is what and how it's shown. This is not a tutorial with an end result, it is more like a lesson in the theory behind it. This documentation is written using files generated by Hammer 4.1 within the Jan 1st 2006 Source SDK Beta.

Contents

The structure of .vmf

The code structure in a .vmf is simple and easy to understand, it's also common in many other factors of the engine. This is a simple review of the structure for those first exposed and to set out the terms that things will be referred to as.

```
// This is a comment.
ClassName_1
{
      "Property_1" "Value_1"
      "Property_2" "Value_2"
      ClassName_2
      {
            "Property_1" "Value_1"
      }
}
```

The given names are merely placeholders using the naming techniques they will be referenced with; Classes, Properties and Values. Within this documentation each section will be laid out the same. It will have a title of the Class it is explaining followed by a simple introduction of what that Class defines. Underneath will be a code sample, and then each property and the valid data type will be stated along with an explanation. There are only a few data types a .vmf file could contain, they are generally standard data types and values but are listed for reference.

Notation	Description
int	Any integer value.
dec	Any decimal value.
sz	Any string of characters.
bool	A Boolean value of true/false in binary.
vertex	An XYZ location given by 3 decimal values separated by spaces.
rgb	A color value using 3 integers between 0 and 255 separated by spaces corresponding to the Red channel, Green channel, and Blue channel, respectively.

The Properties and values will always come right under the code sample and be laid out in the following manner. The explanation will be in brief and cover only that Property.

- **property_1 (data_type):**

  This will explain what the property represents and any further notes on possible values.

Sometimes the explanation cannot be brief, or a Property for one Class is in fact another different Class. When one line requires its own explanation it will be written as a link below the code sample. This merely indicates it is explained under its own title and is not included in the definitions for that Class. After each property is defined the Class will be discussed as a whole, along with the reasons behind the properties and how they affect the end result will be detailed.

Usual Structure

The usual structure of a vmf is as follows:

```
versioninfo{}
visgroups{}
world{}
entity{}
hidden{}
cameras{}
cordon{}
```

- versioninfo{}
- visgroups{}
- viewsettings{}
- world{}
- entity{}
- hidden{}
- cameras{}
- cordon{}

See each of these for what they do. The order does not matter, but Hammer will put them in this order when it saves.

Version info

This Class details information for Hammer on what version created the file and how many times it has been saved. It is basically the file header and contains no information relevant to the map itself.

```
versioninfo
{
	"editorversion" "400"
	"editorbuild" "3325"
	"mapversion" "0"
	"formatversion" "100"
	"prefab" "0"
}
```

- **editorversion (int)**

  The version of Hammer used to create the file, version 4.00 is "400".

- **editorbuild (int)**

  The patch number of Hammer the file was generated with.

- **mapversion (int)**

  It represents how many times you've saved the file, useful for comparing old or new versions.

- **formatversion (100)**

  Unknown. (Most likely the VMF file format version)

- **prefab (bool)**

  Whether it is a full map or simply a collection of prefabricated objects.

What the `formatversion` represents is unknown due to the fact no different values are ever generated. Any changes in value yield no results and are simply overridden back to "100" on save. The `prefab` property is used to define if the file contains an object like a sofa made from other brushes and entities, rather than a full blown map. If any of this information is missing Hammer will simply recreate it from its current form.

VisGroups

This Class itself is usually empty; however, it contains all the unique VIS group definitions in Hammer, along with their structure and properties. The first separated section in the code sample is one unique group. The entire code sample is used for example.

```
visgroups
{

	visgroup
	{
		"name" "Tree_1"
		"visgroupid" "5"
		"color" "65 45 0"
	}

	visgroup
	{
		"name" "Tree_2"
		"visgroupid" "1"
		"color" "60 35 0"
		visgroup
		{
			"name" "Branch_1"
			"visgroupid" "2"
			"color" "0 192 0"
		}
		visgroup
		{
			"name" "Branch_2"
			"visgroupid" "3"
			"color" "0 255 0"
			visgroup
			{
				"name" "Leaf"
				"visgroupid" "4"
				"color" "255 0 0"
			}
		}
	}
}
```

- **name (sz)**

  The name of the group, it's the only identifier within in Hammer so it should be unique for your own benefit.

- **visgroupid (int)**

  A unique value among all other visgroup ids, identical values will cause weird minor errors in Hammer.

- **color (rgb)**

  A color for the group, can be applied to brush outlines in Hammer.

VIS groups are unrelated to the VIS part of a compile and in the new Hammer there are 2 types; auto, and custom. Auto refers to the automatic ones Hammer creates to single out types of brushes or entities, the auto groups are not defined in the file as Hammer generates them. Custom refers to groups that a user creates and they are defined within this Class. In older versions all groups were originally listed here. The groups themselves have a hierarchical tree like structure, meaning that one group may be a parent and have other groups be its children. When something is done to a parent children are affected, but children do not affect parents. To establish one group as a child to another group you define that group within the definition of the parent group. The code sample given is an example of this structure. Tree_1 is the parent of Branches and Branch_2 is the parent of Leaf_1. There's no limit to the complexity of this Class and its subclasses; however, there is a limit of 128 groups before Hammer begins to function improperly. It is possible to have more than one visgroups class, but Hammer will condense them into one, adding the additional vis groups into the main visgroup.

Viewsettings

This class contains the map specific view properties used in Hammer, any of the other properties are not saved into the map.

```
viewsettings
{
	"bSnapToGrid" "1"
	"bShowGrid" "1"
	"bShowLogicalGrid" "0"
	"nGridSpacing" "64"
	"bShow3DGrid" "0"
}
```

- **bSnapToGrid (bool)**

  Whether the map has the grid snapping feature enabled.

- **bShowGrid (bool)**

  Whether the map is showing the 2D grid.

- **bShowLogicalGrid (bool)**

Changes whether the hidden "Logical View" should show a grid

- **nGridSpacing (int)**

  The value the grid lines are spaced at.

- **bShow3DGrid (bool)**

  Whether the map is showing the 3D grid.

If `bSnapToGrid` is true, Hammer will force all points the user interacts with to a multiple of `nGridSpacing`; this does not affect loaded vertices. Any other property set within Hammer will only be set for Hammer and not for the map. If any of this information is missing Hammer will simply recreate it from defaults so it is not essential.

World

In a VMF file, the world Class contains all the world brush information for Hammer. World brushes are the brushes without any entity information attached to them. The `world` Class takes on the same form as other entities but has some unique factors.

```
world
{
	"id" "1"
	"mapversion" "1"
	"classname" "worldspawn"
	"skyname" "sky_wasteland02"
	Solid{}
	Hidden{}
	Group{}
}
```

- **id (int)**

  A unique value among other world Class ids.

- **mapversion (sz)**

  A reiteration of the mapversion from the versioninfo Class.

- **classname (worldspawn)**

  States what type of entity the world is.

- **skyname (sz)**

  The name of the skybox to be used.

- solid{}
- group{}
- hidden{}

The definition of the World class actually takes the form of an entity. This causes the possibility for a large number of properties to be defined, they have been omitted for simplicities sake as they followed standard entity notation. As entities are discussed later only the following information will be given for now. The `classname` for World **always** has to be "worldspawn". Without it the map is not properly defined and cannot be compiled properly, "worldspawn" is a special type of entity the game has to find here. The only property that must be included is the `skyname` to avoid errors in compiling. It is possible to define multiple world Classes, but Hammer will condense them into one single definition, adding the solids and groups of the additional world classes into the first, and using the "latest" properties; that is, if the first world has ("A" "1") and ("B" "2") but the second world only has ("B" "3"), the new world will have ("A" "1") and ("B" "3").

Solid

This Class represents 1 single brush in Hammer. It contains 1 property and 2 subclasses. One brush is defined simply by the total of all sides that make up its shape. All brushes have to be saved as a valid shape or Hammer will either not load the solid or reshape it so that it becomes valid.

```
solid
{
	"id" "1"
	side{}
	editor{}
}
```

- **id (int)**

  A unique value among other solids' IDs, Identical values will cause weird minor errors.

- side{}
- editor{}

Within the `solid` Class you must define multiple sides, at least four. As mentioned this is because each stored brush should be valid, due to the way Hammer calculates and represents brushes a non valid brush will cause errors either in Hammer or in game.

Side

This Class defines all the data relevant to one side and just to that side. It details its orientation, texture, and properties. The plane property defines the orientation of the face. Where a face intersects with other planes is used to define all of its borders, thus they are not saved into the file. The u-axis and v axis properties are complex and require an independent explanation.

```
side
{
	"id" "6"
	"plane" "(512 -512 -512) (-512 -512 -512) (-512 -512 512)"
	"material" "BRICK/BRICKFLOOR001A"
	"uaxis" "[1 0 0 0] 0.25"
	"vaxis" "[0 0 -1 0] 0.25"
	"rotation" "0"
	"lightmapscale" "16"
	"smoothing_groups" "0"
	dispinfo{}
}
```

- **id (int)**

  A unique value among other sides ids. Identical values will cause weird minor errors.

- plane ()
- **material (sz)**

  The directory and name of the texture the side has applied to it.

- uaxis ()
- vaxis ()
- **rotation (dec)**

  The rotation of the given texture on the side. The rotation is built into the uaxis and vaxis, this value is merely displayed to the user, changing it does not change the texture's rotation, just the value displayed to the user.

- **lightmapscale (int)**

  The light map resolution on the face.

- **Smoothing_groups (int)**

  Select a smoothing group to use for lighting on the face.

- dispinfo{}

The id values should be unique even between solids. That is because the face identifier needs to be unique for entities like cubemaps that reference one specific face. The `lightmapscale` Property refers to the size you define for the lighting patches to be applied to the face during compiling. The `smoothing_groups` Property indicates which smoothing groups the face will be placed in. All adjacent faces in a smoothing group will have the differences in light between them bled together. The easiest way to determine what groups the value indicates is by converting it back to binary. A high in the least significant bit will mean it's in the first group, a high in the last bit will mean it's in the 32nd group.

Planes

A plane is a fundamental two-dimensional object. It can be visualized as a flat infinite sheet of paper. That piece of paper is then positioned in a three-dimensional world. Where planes intersect forms a brushes edges and vertices.

```
"plane" "(0 0 0) (0 0 0) (0 0 0)"
```

- **plane ((vertex) (vertex) (vertex))**

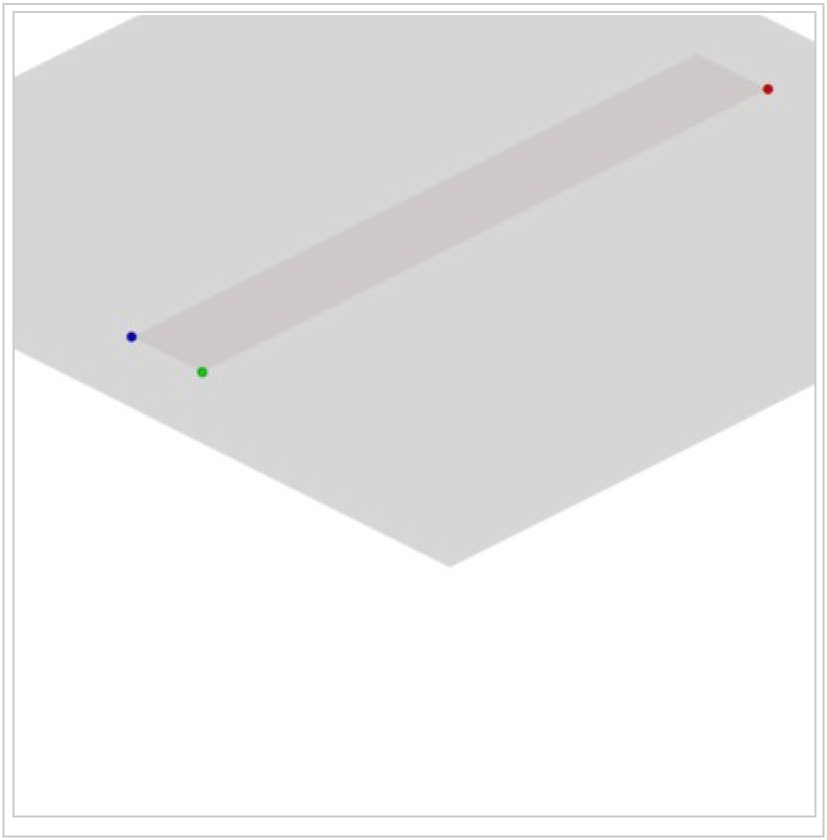
  This defines the three points that will be used to set the plane's orientation and position in the three-dimensional world.

To define the orientation three sequential points are used. Think of these as three objects the sheet is placed upon. If one is higher or lower than the others it will cause the paper to sit at an angle. By choosing how high or low the other points are you can rotates the sheet in any direction. The paper passes through all three defined points, this means they can be used to represent the orientation of the plane. The direction the plane faces is determined by the order you define them in. Defining them in a clockwise order when looking from one direction order will cause the face to be visible from that direction. The following example will generate a plane for a square 32x32 floor, a plane flat in the z-axis that will face upwards.

```
"plane" "(-16 -16 0) (16 -16 0) (16 16  0)"
```

The plane is defined using these three points. The first marks the bottom left of the face, the second marks the top left, and the third marks the top right. When viewed from the top this forms a clockwise order, this means the plane will be facing upwards. The vertex positions leave a space of 32 units for each side and lay out 3 corners of a square. This will form the required floor.

On the right is an animation of a simple brush being build from six planes via CSG. The first, second and third plane point is represented by red, green and blue dots. Below are the plane points for this brush:



```
"plane" "(-128 32 128) (128 32 128) (128 0 128)"
"plane" "(-128 0 0) (128 0 0) (128 32 0)"
"plane" "(-128 32 128) (-128 0 128) (-128 0 0)"
"plane" "(128 32 0) (128 0 0) (128 0 128)"
"plane" "(128 32 128) (-128 32 128) (-128 32 0)"
"plane" "(128 0 0) (-128 0 0) (-128 0 128)"
```

The next example details a crucial note.

```
"plane" "(-32 -32 0) (32 -32 0) (32 32 0)"
"plane" "(32 -32 0) (32 32  0) (-32 32 0)"
"plane" "(32 32  0) (-32 32 0) (-32 -32 0)"
"plane" "(-32 32 0) (-32 -32 0) (32 -32  0)"
```

All of the above definitions will create the same plane. The sequential points chosen to define the plane will form the same plane regardless, if choosing different points on the same face forms a different plane then that brush is invalid. The vertices that make up all the edges of a face are determined using the intersection of all the defined planes of a brush. Vertices mark the point where three or more planes intersect and edges mark the area where two planes intersect along a line. This is almost a base definition of how they are supposed to exist and cannot be altered. The problem comes with the fact that vertices are used to define the planes, as such the points of intersection have already been defined and must be adhered to. Failure to do so will generate many varied shapes but will always results in an invalid brush. When defining a plane any other defined vertices must be able to be placed on that plane or another plane defined within the brush.

U/V Axis

The u-axis and v-axis are the texture specific axes. They hold their own designations as they are independent of the true x-axis and y-axis, but it's easier to think of them in terms of x and y. The u-axis represents the x-axis and the v-axis represents the y-axis. These two code lines work together to define how the texture is displayed upon a face.

```
"uaxis" "[1 0 0 0] 0.25"
"vaxis" "[0 1 0 0] 0.25"
```

- **u/v axis ([x y z dec] dec)**

  The x, y, and z are decimal values representing that axis. The following value is the translation and the last decimal is the total scaling.

The values for the x y z set represent the amount of the texture to be shown in that axis. They modify how much of the texture is shown within the normal space the texture would occupy on the face. A value of one means one repetition in the normal space, a value of two would mean two repetitions in the normal space. The key is that this only applies to one axis of the texture and is applied relative to true x, y, and z axes. That means that it is possible to set the texture to appear along an axis the plane doesn't exist in. With a flat plane in the x-axis and the y-axis definitions are only needed for those axes as any definition in the z-axis has no surface to appear on. The code sample shows how this would be generated by Hammer. The texture's x-axis (the u-axis) is displayed in the real x-axis and the texture's y-axis (v-axis) is displayed in the real y-axis. Which of the x, y, and z axes are showing the axes of the texture depends entirely upon the faces orientation. Negative values will flip textures and Hammer will generate errors about textures shown in a plane the surface does not occupy. The second to last value is the simple translation of the textures origin within that axis; this value is actually a decimal though Hammer will round it up when displaying Hammer will still save the correct decimal value. The last decimal is the overall scaling of the entire axis including what is defined in the x y z group. With an effective scale of 1, 1 pixel on the texture corresponds to 1 Hammer Unit.

Dispinfo

The `dispinfo` Class deals with all the information for a displacement map. Information is stored on each vertex and thus creates a large amount of information to be sorted. This leaves a lot of subclasses to help organize all the information. The presence of the `dispinfo` Class itself tells Hammer that the face is a displacement map and the Class therefore should not be included if the face is not a displacement. All the properties are applied over top of the original side's properties.

```
dispinfo
{
	"power" "2"
	"startposition" "[-512 -512 0]"
	"elevation" "0"
	"subdiv" "0"
	normals{}
	distances{}
	offsets{}
	offset_normals{}
	alphas{}
	triangle_tags{}
	allowed_verts{}
}
```

- **power (2,3,4)**

  Used to calculate the number of rows and columns, only the three given values will compile properly. If omitted Hammer assumes 4. If the data is built around another power value, the vertex data will be squished into the starting corner.

- **startposition (vertex)**

  The position of the bottom left corner in an actual x y z position.

- **elevation (float)**

  A universal displacement in the direction of the vertex's normal added to all of the points.

- **Subdiv (bool)**

  Marks whether or not the displacement is being sub divided.

- normals{}
- distances{}
- offsets{}
- offset_normals{}
- alpha{}
- traingle_tags{}
- allowed_verts{}

There's also a temporary value called "scale" within in Hammer. This value is actually not saved into the file and only affects the scaling of current movements. All grid based subclasses follow a similar system, the equation 2^*n* + 1 is used to determine the number of rows where *n* is the power. This determines the number of rows and columns to be used. The rows are numbered and stored starting from 0 at the vertex `startposition` defines. Each Property will simply be row# where # is the rows number. Sometimes a column will have multiple parts and lead to there being 2 or 3 times as many values.

Normals

The Class defines the normal line for each vertex. This is a line that points outwards from the face and is used for light shading and vertex placement. Each column contains a group of 3 values.

```
normals
{
      "row0" "X0 Y0 Z0 X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3 X4 Y4 Z4"
      "row1" "0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
      "row2" "0 0 1 0 0 1 0 0 1 0 0 1 0 0 1"
      "row3" "0 0 -1 0 0 -1 0 0 -1 0 0 -1 0 0 -1"
      "row4" "0 0 1 0 0 -1 0 0 -1 0 0 -1 0 0 1"
}
```

- **row# ((x y z) (x y z) ...)**

  Each group of decimals is used to define the normal line for that vertex.

The normal line is an imaginary one unit long line drawn from the vertex. The decimal values represent the length of the line that exists in that axis. A line straight along one axis will be a value of 1 in that axis, if however the line is sloped part of it will be in one axis and another part will be another axis. That results in two different decimal values in the different axes of the same column. This line is important for two reasons, the line is used as a guide on how to light the displacement map, and the line is used to guide the vertex to its defined position. In the code sample it shows how normal lines may appear in the last three rows. The first row is all straight up, and the second row is all straight down. In the third row the inside three points face down and the two outside points face up.

Note that having negative distances with an up normal (0, 0, 1) is the same as having positive distances with a down (0, 0, -1) normal.

Distances

The distance values represent how much the vertex is moved along the normal line. With an undefined normal line the distance is not applied.

```
distances
{
      "row0" "#0 #1 #2 #3 #4"
      "row1" "0 0 0 0 0 "
      "row2" "64 64 64 64 64"
      "row3" "64 64 64 64 64"
      "row4" "32 32 32 32 32"
}
```

- **row# (dec dec ...)**

  Any decimal value representing the final distance a vertex is moved along the normal line.

Once a distance value is set the vertex is moved along the normal line for the given value. This then gives the final position for the vertex. Although negative values are accepted and compiled it is better for the normal line to handle the direction of the displacement. Applying the values from the `normals` Class to this Class gives a displacement map its form. If the code samples for rows 2, 3, and 4 were actually used they would create a displacement map with three unique rows. One row would be 64 units high, the next 64 units deep. The third would have three 32 unit deep points and the two outside points 32 units high.

Offsets

This Class lists all the default positions for each vertex in a displacement map. The `distances` are judged from the points the `offsets` Class defines. Each column contains a group of 3 values.

```
offsets
{
      "row0" "X0 Y0 Z0 X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3 X4 Y4 Z4"
      "row1" "0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
      "row2" "0 0 64 0 0 64 0 0 64 0 0 64 0 0 64"
      "row3" "0 0 -64 0 0 -64 0 0 -64 0 0 -64 0 0 -64"
      "row4" "0 0 32 0 0 -32 0 0 -32 0 0 -32 0 0 32"
}
```

- **row# (vertex vertex ...)**

  Each group is the vertex position in x, y, and z relative to the original position.

The `offsets` Class defines new default vertex positions relative to the original calculated positions. In the code sample above just the contents of this Class recreates an identical displacement map to the previous two Classes combined. Hammer does not modify these points in any way.

offset_normals

This Class is almost identical to the `normals` Class. The data structure is identical except the name, thus the code sample has been omitted to save space. The only differences lie in what it does and what it affects. This class defines the default normal lines that the `normals` Class is based from. It does not modify the `offsets` Class. Hammer has no options to modify these values, it can only set them based upon the faces orientation.

Alpha

This Class contains a value for each vertex that represents how much of which texture to shown in blended materials. The difference between the values is merged linearly across the displacement map.

```
alphas
{
      "row0" "#0 #1 #2 #3 #4"
      "row1" "0 0 0 0 0 "
      "row2" "255 255 255 255 255"
      "row3" "0 0 0 0 0"
      "row4" "128 0 0 0 128"
}
```

- **row# (dec dec ...)**

  It represents a decimal between 0 and 255.

The value simply marks how much of the secondary texture is visible overtop the base texture. The differences in values are blended via linear interpolation so only the key values at vertices are stored. As the last 3 rows in the code sample provide an example. The first row is pure secondary texture, and the second row is pure base. The third row has the two outside points showing half of each texture and the inner 3 showing all base texture.

triangle_tags

This Class contains information specific to each triangle in the displacement, rather than every vertex. This means that the size and number of its rows will differ. It follows the pattern of 2^*n* where *n* is the power. The row and column numbers are still defined from the starting point outwards.

```
triangle_tags
{
	"row0" "#0 #0 #1 #1 #2 #2 #3 #3 #4 #4"
	"row1" "9 9 9 9 9 9 9 9"
	"row2" "0 0 0 0 0 0 0 0"
	"row3" "1 1 1 1 1 1 1 1"
	"row4" "9 9 9 9 9 9 9 9"
}
```

- **row# (((0,1,9) (0,1,9)) ((0,1,9) (0,1,9))...)**

  Each group represents one square consisting of two triangles. Only values of 0, 1, or 9 will compile.

The value assigned to the triangle represents its orientation. A value of "9" means the triangles has little or no slope in the z-axis. A value of "1" means it has a significant slope in the z-axis but a player is still able to walk on it. A value of "0" means it has a large slope in the z-axis and the player cannot walk up it. Any value is accepted by Hammer, a value of "2" will even produce green highlights in the Display walkable area view, but any such values will not compile.

allowed_verts

This affects the in-game tesselation of the displacement map, stating which vertices share an edge with another displacement map but do not share a vertex.

```
allowed_verts
{
	"10" "-1 -1 -1 -1 -1 -1 -1 -1 -1 -1"
}
```

- **10 (int, int, int, int, int, int, int, int, int, int)**

A set of binary flags which corresponds to which vertices are allowed in the displacement map. A false flag removes the vertex from the compiled map. Note that -1 is all bits set to true.

This by definition has no effect in Hammer, it modifies properties for compiled maps which the editor does not need to consider. A quick way to view the "disallowed" vertices is to turn on the Show collapsible vertices button. The vertices will now be shown as pink cubes. When the displacement maps are compiled the disabled vertices are interpolated with its neighbors to fit with the line. See in hammer and after compiling.

Editor

All information within this Class is for Hammer only and bears no significance to the map itself.

```
editor
{
	"color" "0 255 0"
	"visgroupid" "2"
	"groupid" "7"
	"visgroupshown" "1"
	"visgroupautoshown" "1"
	"comments" "Only exists on entities."
	"logicalpos" "[34 28]"
}
```

- **color (rgb)**

  A color for the brushes outline and flat color.

- **visgroupid (int)**

  The id of the visgroup this solid belongs to.

- **group (int)**

  The id of the brush group this solid belongs to.

- **visgroupshown (bool)**

  Whether or not that group is shown.

- **visgroupautoshown (bool)**

  Whether or not the automatic group is shown.

- **comments (string)**

  A comment string that Hammer will show in the Class Info tab pane.

- **logicalpos (2D vector)**

  The 2D position of the entity in the hidden "Logical View".

The `visgroupid` is special in that you can define the property as many times as you want in order to assign the brush to multiple visgroups. The group property defines which brush group the solid belongs to in Hammer. The number is the id of the first child group the brush belonged to. The `visgroupshown` is one misplaced item, why include the definition of a group being visible within a brush for that group and for every brush in that group. All it does is allow a brush to be visible when it shouldn't be. The `comments` property allows you to make comments on your entities for yourself or other map developers. You can find the comments in the lower right of the Class Info pane in Entity Properties. Note that this only works on entities, and will be ignored and deleted by Hammer if placed on anything else. The "Logical View" is a partially-implemented view available in some versions of Hammer, displaying all entities and their IO connections in a flowchart-like fashion. `logicalpos` stores the position of each entity in this display. Hammer appears to initialize this to a semi-random location along the positive X axis.

Group

This class sets the brush groups that exist and their properties. It also defines their hierarchy.

```
group
{
	"id" "7"
	editor{}
}
```

- **id(int)**

  This is a unique number among other group ids.

- editor{}

A group is assigned one id, the brush itself states whether or not it is within the group. The definition for the `editor` Class is included again. This is important because it is how the hierarchy of all the groups is maintained. If one group is a child to another it then includes the group id within its `editor` subclass. That defines it is a sub group of the parent group, so when the parent groups break all the child groups will remain intact.

Hidden

There are two versions of the hidden class, but both include classes which have the `visgroupshown` or `autovisgroupshown` in `editor` set to "0".

The first type is in the `world` and `entity` class

```
hidden
{
	solid{}
}
```

- solid{}

This is very simple, just every hidden solid that would usually be held under `world` or `entity` is instead held under this. There's no other difference. Hammer itself will show solids under "hidden" if both of the visgroupshown properties are true.

The second type is under the file itself:

```
hidden
{
	entity{}
}
```

- entity{}

As with the other version this is simple, it simply holds the entity which is hidden. No other difference is visible. Again, Hammer will show entities with both `visgroupshown` properties set to true, even if it is under "hidden".

Entity

Both brush and point based entities are defined in the same way. Both are defined outside of the `world` class as the `world` class itself is an entity. All entities work in the same way: you define its properties and then either give the entity solids to inhabit or a point to exist at.

```
entity
{
	"id" "19"
	"classname" "func_detail"
	"spawnflags" "0"
	______
	connections{}
	solid{}
	hidden{}
	"origin" "-512 0 0"
	editor{}
}
```

- **id(int)**

  This is a unique number among other entity ids.

- **classname(sz)**

  This is the name of the entity class.

- **spawnflags(int)**

  Indicates which flags are enabled on the entity.

- **origin(vertex)**

This is the point where the point entity exists.

- connections{}
- solid{}
- editor{}
- hidden{}

There are several lines that are important. First is the underscores, this is a placeholder for all the entities properties and values. These are the exact same thing seen with smart edit disabled in Hammer. The properties and values can be anything, but only valid properties will mean anything when compiled. It may be wise to review the definition of the `world` Class and the Worldspawn entity. With entities the possible values and properties extend far beyond the scope of this document and should be reviewed independently. Second is the `connections{}` Class. This is all the i/o information and is optional. Third, is the solid line. Defining a `solid` subclass will make it a brush based entity; having both an origin and solid will make a brush based entity with an origin property. Defining only an `origin` will make it a point based entity. Any missing entity properties will be recreated from default upon compiling and do not have to be saved.

Connections

This is where all the outputs for an entity are stored. Inputs are not stored as they are simply traced from the outputs. Therefore only the outputs are stored in this optional class.

```
connections
{
	"OnTrigger" "bob,Color,255 255 0,1.23,1"
	"OnTrigger" "bob,ToggleSprite,,3.14,-1"
}
```

- **Output(sz,Input,sz,dec,bool)**

  The entire output event stored in 1 string.

The string is pretty simple to decompose. The property is the output event triggering another entities input. The value on the other hand, is either a comma separated string (Source Engine Games without Squirrel/Lua Support) or a escape character (byte: 0x1b) separated string of the rest of the output's properties. First is the target entity name or class. Second is the Input being triggered on the target object (s). Third is the parameter override, leave blank for no value. Fourth being the delay before triggering. The final parameter is the number of times to fire, or -1 to always fire. To create multiple outputs you simply add more properties.

Note that this is one string which Hammer does not make assumptions about. If you omit any value (except the last one) Hammer will crash with an unhelpful error message. If you omit the last value, which is the number of times to fire, Hammer sets it to 1 or Fire Once.

Cameras

Used for the 3D viewport cameras used in Hammer, created with the Camera Tool. These cameras are not used in-engine and are not compiled into the final .BSP.

```
cameras
{
    "activecamera" "1"
    camera
    {
        "position" "[-1093.7 1844.91 408.455]"
        "look" "[-853.42 1937.5 175.863]"
    }
    camera
    {
        "position" "[692.788 1394.95 339.652]"
        "look" "[508.378 1493 347.127]"
    }
    camera
    {
        "position" "[-4613.89 2528.77 -2834.88]"
        "look" "[-4533.53 2950.1 -2896.85]"
    }
}
```

```
cameras
{
    "activecamera" "-1"
}
```

- **activecamera (int)**

  Sets the currently active camera used for the Hammer 3D View. When no cameras are present in the .VMF, this value is set to -1 and the camera position defaults to the world origin, facing North.

- **position (vertex)**

  The eye position of the camera in the map.

- **look (vertex)**

  The position of the camera target -- the point the camera is looking toward.

Cordon

This stores all the information Hammer needs and uses for its cordon tool.

```
cordon
{
     "mins" "(99999 99999 99999)"
     "maxs" "(-99999 -99999 -99999)"
     "active" "0"
}
```

- **mins (vertex)**

  The top right upper co-ordinate.

- **maxs (vertex)**

The bottom left lower co-ordinate.

- **active (bool)**

  Whether or not the cordon is turned on.

After the 2 points are defined, a rectangular prism is drawn between the two points. This will make any object that is outside the prism excluded from rendering or compiling.

In newer Hammer versions (L4D onwards)

```
cordons
{
	"active" "0"
	cordon
	{
		"name" "cordon"
		"active" "1"
		box
		{
			"mins" "(-1204 -1512 -748)"
			"maxs" "(836 444 1128)"
		}
	}
}
```

Cordons are in a group cordons and a cordon itself is a set of boxes defined by mins and maxs. (See above)

Conclusion

That's all the information on the .vmf file format that Hammer will generate. This hopefully covers all possible questions and enables anyone to do what they wish with the file format. Below are some files for use as reference or used for the experiments that revealed how the format is laid out.

Retrieved from "https://developer.valvesoftware.com/w/index.php?title=Valve_Map_Format&oldid=220347"

Categories: File formats | Programming | Level Design

- This page was last modified on 24 October 2018, at 23:21.