

# .MAP files

## file format description, algorithms, and code

By Stefan Hajnoczi  
[stefanha@pon.s.uio.no](mailto:stefanha@pon.s.uio.no)

### Table Of Contents

- **Disclaimer**
- **Introduction**
- **.MAP files**
  - The .MAP file format
  - .MAP parser design
  - Turning a brush into a set of polygons
  - Calculating texture coordinates
  - Sorting vertices
  - Performing constructive solid geometry union on all brushes
  - Classifying a polygon against a plane
  - Splitting a polygon by a plane
- **Source code**
- **Special thanks**
- **Bibliography/Links**

### Disclaimer

Although I [Stefan Hajnoczi] have put a lot of work into this article, I cannot guarantee that there are no mistakes. I do not take responsibility for any damage or problems caused by the code provided with this article (both compiled binary and source code).

I am in not way part of iD Software or Valve Software and this document is not an official guide to WorldCraft or any of the file formats described. Because of this, I cannot guarantee for the accuracy of the information provided.

In order to understand this article, you should be familiar with WorldCraft 3.3, C++, vectors, and planes. Finally, I'd also like to say that a lot of code dealing with .MAP files and .WAD files can be found in the Half-Life SDK (<http://www.valvesoftware.com/hlsdk.htm>).

### Introduction

After you have created a map in WorldCraft, click on File | Export to .MAP to save the map as a .MAP file. The saved .MAP file contains all the brushes and entities in the level. Since you are making your own engine, you will want to write your own map compile tools. The first step to making custom compile tools is parsing the .MAP file to get a list of all entities, entity properties, and polygons. Unfortunately, parsing .MAP files is not easy. This paper provides an overview of how to do it, as well as descriptions of every step. If you download this paper, the source code of my .MAP file parser is included.

The way I learnt all this was to read all available information about WorldCraft. Unfortunately a lot of information was repeated, while other parts were missing. I had to figure out the .MAP file format myself, because the only online information about it was out-of-date. That is why I compiled this document. I hope this document will serve as a comprehensive paper about how to modify WorldCraft in order to use it with your own engine. A few people did help me along the way, and I would like to thank these people in the special thanks section.

### The .MAP file format

I think the best way to learn the .MAP file format is just to jump straight in to a simple example. This is an example .MAP file containing just a box:

```
{
"classname" "worldspawn"
"mapversion" "220"
"wad" "\games\half-life\cstrike\cstrike.wad;\games\half-life\valve\halflife.wad"
{
( 0 64 64 ) ( 64 64 64 ) ( 64 0 64 ) BC RATE02 [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 0 0 0 ) ( 64 0 0 ) ( 64 64 0 ) BC RATE02 [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 0 64 64 ) ( 0 0 64 ) ( 0 0 0 ) BC RATE02 [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 64 64 0 ) ( 64 0 0 ) ( 64 0 64 ) BC RATE02 [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 64 64 64 ) ( 0 64 64 ) ( 0 64 0 ) BC RATE02 [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 64 0 0 ) ( 0 0 0 ) ( 0 0 64 ) BC RATE02 [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
}
}
```

Curly brackets ('{' and '}') are put around every **entity**, hence the curly bracket at the beginning of the file. Everything in a .MAP file is part of an entity. Obvious entities are doors, lights, player starts, etc. as these are called entities in WorldCraft. However, even the basic level geometry, which seems not to be part of an entity is, in fact, part of an entity. Every entity can consist of two things: **properties** and **brushes**. Properties consist of a **name** (color coded in green), which tells you which type of property this is and a **value** (color coded in pink). Properties help you store non-geometric data about an object. This could include things like health, for a monster, or the filename of a sound used when an something is triggered. Brushes are enclosed in curly brackets ('{' and '}'). Brushes represent the actual geometry of a level. This include walls, floors, tables, etc. (everything that consists of polygons). Each brush consists of four or more **faces**. A face is a plane, texture name, and texturing information. I will go into more detail about brushes later.

There is one property that every entity must have: **classname**. Classname specifies which type of entity is being described. Examples of this would be: "classname" "worldspawn", "classname" "light", or "classname" "button". Classname is the first property of any entity.

**Worldspawn** is an entity which must exist in any .MAP file. Worldspawn is the first entity in any .MAP file. It contains all brushes that aren't any special entities. In a typical level, worldspawn will contain walls, floors, ceilings, etc. No matter how big or small a level is, it must have one worldspawn. Worldspawn doesn't only contain level geometry, but also the .MAP file version information and the .WAD files used. The .MAP file version is given in the **mapversion** property. For WorldCraft 3.3, the version is 220. The .WAD files used are listed in the **wad** property. Different .WAD files are separated by semicolons (;).

Brushes consist of four or more faces. There is no limit to the number of brushes that can be in an entity. Brushes are always convex, which is a very useful property (as will be seen later on). Instead of being described as a set of polygons, brushes are described as a set of faces. An example face is:

```
( 0 64 64 ) ( 64 64 64 ) ( 64 0 64 ) BC RATE02 [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
```

It starts with a **three point plane description**. Then the **texture name** is given. Next the **U and V texture axis** are given. Then the **texture rotation** is given (which is useless information because the texture axis are already rotated). Last, the **U and V texture scale** is given.

I will clarify the texture axis. The first one is the U axis and the second one is the V axis. The first three numbers of a texture axis give the texture axis' normal. The fourth number gives the texture shift (or offset). Each texture axis can be imagined as a plane.

A typical .MAP file will consist of one big worldspawn and many (200+) entities representing lights, doors, switches, triggers, etc. A final example .MAP file shows multiple entities and brushes. It is a quick Counter-Strike level I made. It has one room, a light, a target, and a player start. Don't concentrate on what every entity does or means, but on the syntax of the .MAP file.

```
{
"classname" "worldspawn"
```

```

"MaxRange" "4096"
"mapversion" "220"
"wad" "\games\half-life\cstrike\cstrike.wad;\games\half-life\valve\halflife.wad"
{
( 0 0 256 ) ( 0 256 256 ) ( 256 256 256 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 0 256 224 ) ( 0 256 256 ) ( 0 0 256 ) AAATRIGGER [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 256 0 224 ) ( 256 0 256 ) ( 256 256 256 ) AAATRIGGER [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 256 256 224 ) ( 256 256 256 ) ( 0 256 256 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 0 0 224 ) ( 0 0 256 ) ( 256 0 256 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 0 256 224 ) ( 0 0 224 ) ( 256 0 224 ) C1A0_W2 [ 1 0 0 0 ][ 0 -1 0 160 ] 0 2 1.6
}
{
( 0 256 0 ) ( 0 0 0 ) ( 256 0 0 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 0 0 32 ) ( 0 0 0 ) ( 0 256 0 ) AAATRIGGER [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 256 256 32 ) ( 256 256 0 ) ( 256 0 0 ) AAATRIGGER [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 0 256 32 ) ( 0 256 0 ) ( 256 256 0 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 256 0 32 ) ( 256 0 0 ) ( 0 0 0 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 0 0 32 ) ( 0 256 32 ) ( 256 256 32 ) C1A0_LABFLRB [ 1 0 0 0 ][ 0 -1 0 128 ] 0 2 2
}
{
( 0 0 224 ) ( 0 0 32 ) ( 0 256 32 ) AAATRIGGER [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 0 256 224 ) ( 0 256 32 ) ( 32 256 32 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 32 0 32 ) ( 0 0 32 ) ( 0 0 224 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 0 0 224 ) ( 0 256 224 ) ( 32 256 224 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 0 256 32 ) ( 0 0 32 ) ( 32 0 32 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 32 256 224 ) ( 32 256 32 ) ( 32 0 32 ) C1A0_W1D5 [ 0 1 0 0 ][ 0 0 -1 26.6667 ] 0 2 1.2
}
{
( 256 256 224 ) ( 256 256 32 ) ( 256 0 32 ) AAATRIGGER [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 224 256 32 ) ( 256 256 32 ) ( 256 256 224 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 256 0 224 ) ( 256 0 32 ) ( 224 0 32 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 224 256 224 ) ( 256 256 224 ) ( 256 0 224 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 224 0 32 ) ( 256 0 32 ) ( 256 256 32 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 224 0 224 ) ( 224 0 32 ) ( 224 256 32 ) C1A0_W1D5 [ 0 1 0 0 ][ 0 0 -1 26.6667 ] 0 2 1.2
}
{
( 32 256 32 ) ( 224 256 32 ) ( 224 256 224 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 32 224 224 ) ( 32 256 224 ) ( 224 256 224 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 224 256 32 ) ( 32 256 32 ) ( 32 224 32 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 32 224 32 ) ( 32 256 32 ) ( 32 256 224 ) BCRA02 [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 224 256 224 ) ( 224 256 32 ) ( 224 224 32 ) BCRA02 [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 224 224 32 ) ( 32 224 32 ) ( 32 224 224 ) C1A0_W1D5 [ 1 0 0 -21.3333 ] [ 0 0 -1 26.6667 ] 0 1.5 1.2
}
{
( 224 0 32 ) ( 32 0 32 ) ( 32 0 224 ) AAATRIGGER [ 1 0 0 0 ][ 0 0 -1 0 ] 0 1 1
( 224 0 224 ) ( 32 0 224 ) ( 32 32 224 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 32 32 32 ) ( 32 0 32 ) ( 224 0 32 ) AAATRIGGER [ 1 0 0 0 ][ 0 -1 0 0 ] 0 1 1
( 32 0 224 ) ( 32 0 32 ) ( 32 32 32 ) BCRA02 [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 224 32 32 ) ( 224 0 32 ) ( 224 0 224 ) BCRA02 [ 0 1 0 0 ][ 0 0 -1 0 ] 0 1 1
( 32 32 32 ) ( 224 32 32 ) ( 224 32 224 ) C1A0_W1D5 [ 1 0 0 -21.3333 ] [ 0 0 -1 26.6667 ] 0 1.5 1.2
}
}
{
"classname" "info_player_start"
"angles" "0 0 0"
"origin" "64 64 80"
}
{
"classname" "light"
"_light" "255 255 128 200"
"target" "Light01.Target"
"targetname" "Light01"
"origin" "128 128 208"
}
{
"classname" "info_target"
"targetname" "Light01.Target"
"origin" "128 128 32"
}
}

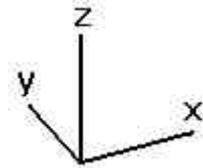
```

## .MAP file parser design

Now that you are familiar with the .MAP file format, we can start thinking about how we are going to code a .MAP file parser. The output from parsing the .MAP file should be a list of entities. Each entity should have properties and polygons. Notice that this isn't much different than what .MAP files are. The

problem is that instead of storing geometry as a list of polygons, .MAP files consist of brushes which are convex polyhedra defined by planes. So the task of a .MAP file parser is to load the .MAP file into memory, to convert the brushes in an entity into a set of polygons, and to the refined entity list.

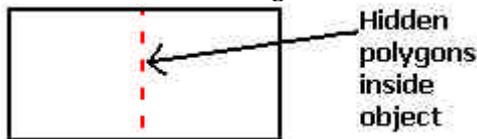
The coordinate system that WorldCraft uses looks like this:



To convert from WorldCraft coordinates to the left-handed coordinate system, you need to switch the y and z components of every coordinate.

You may ask yourself how complicated buildings and objects can be built if brushes are convex. First, as you already know, each entity can have more than one brush. If you put two convex polyhedra next to each other, you can form a more complicated concave polyhedron. However, there will be polygons inside the actual object.

To help you get the idea, imagine two cubes put next to each other so that they are touching. They form a rectangular polyhedron that is longer on one axis than on the others. However, the two touching sides are inside the new object. Firstly, it is a waste of memory to store these polygons because they can never be seen. Secondly, this sort of geometry may be illegal for BSP compilation. Here is a picture of the situation I am describing:



This problem can be solved by using the Constructive Solid Geometry (CSG) union operation. The union operation will eliminate any polygons inside an object and allow you to join up two polyhedra while removing any unnecessary geometry.

Another problem you face when you write your .MAP file parser is accuracy. From the start, I decided to use the 'double' data type instead of 'float'. This allows for more accuracy. Still, rounding errors creep in, and eventually coordinates may become inaccurate. To fight this problem, epsilons are introduced. Instead of testing for precise values such as zero, you test if the value is within a certain range to zero. This is necessary whether you like it or not. I did not use epsilons in the pseudo code throughout this paper because they depend on your data type as well as your world scale and dimensions. To get an idea of how and when they are used, look through the source code, especially the math and classification routines. 'Fuzzy' compares can be used not only when comparing two numbers, but also when comparing two vectors. For example, to check if two normalized vectors are equal,  $\vec{a} \cdot \vec{b} \approx 1$ . A way to do this in pseudo-code is:

```
double Angle = a.Dot ( b ) - 1;

if ( ( Angle > -epsilon ) && ( Angle < epsilon ) )
{
    ...they are equal...
}
```

These are the major steps in parsing a .MAP file. I will list them again to make sure you understand.

- Load .MAP file into memory
- Create polygons out of brushes
- Perform CSG union on every brush in an entity
- Save the list of entities to whatever format you like

## Turning a brush into a set of polygons

Once an entity list is in memory, the first task is to convert the brushes into polygons. There are two ways to accomplish this. The first is by the **intersection of 3 planes**. This method is fast and requires relatively little code. The second method is to create huge, potential polygons along every face's plane, and then to clip them against each other. I personally dislike that method, so I chose to use the intersection of 3 planes.

The pseudo code to turn a brush into a polygon soup is:

Faces is an array of all the faces in the brush

Create an array Polys which has as many members as Faces

```
For i = 0 to NumberOfFaces - 1
{
    For j = 0 to NumberOfFaces - 1
    {
        For k = 0 to NumberOfFaces - 1
        {
            if ( i != j != k )
            {
                Polys[ i ].AddVertex ( GetIntersection ( i, j, k ) );
            }
        }
    }
}
```

This really glosses over the details because I left out the GetIntersection function which calculates the intersection of 3 planes. Also, this version is very unoptimized. Lastly, there is still one problem which I will show you later. The way it works is that every combination of 3 faces is checked for an intersection. The resulting vertices are stored in the polygons corresponding to the planes that caused the intersection. Here is the intersection of 3 planes formula:

$$\bar{p} = \frac{-d_1(\bar{n}_2 \times \bar{n}_3) - d_2(\bar{n}_3 \times \bar{n}_1) - d_3(\bar{n}_1 \times \bar{n}_2)}{\bar{n}_1 \bullet (\bar{n}_2 \times \bar{n}_3)}$$

Each plane consists of a normal, n, and the distance from the plane to the origin, d. this should be fairly logical, as all these variables come from the plane equation  $ax + by + cz + d = 0$  (or more compactly  $n \bullet x + d = 0$ ). Note that if the denominator is 0, there is no intersection (the planes are parallel). The pseudo code to find the intersection of 3 planes would be:

```
bool GetIntersection ( n1, n2, n3, d1, d2, d3, &p )
{
    double denom = n1.Dot ( n2.Cross ( n3 ) );

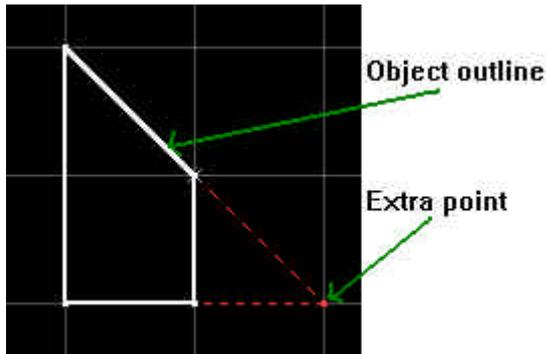
    if ( denom == 0 )
    {
        return false;
    }

    p = -d1 * ( n2.Cross ( n3 ) ) - d2 * ( n3.Cross ( n1 ) ) - d3 * ( n1.Cross ( n2 ) ) / denom;

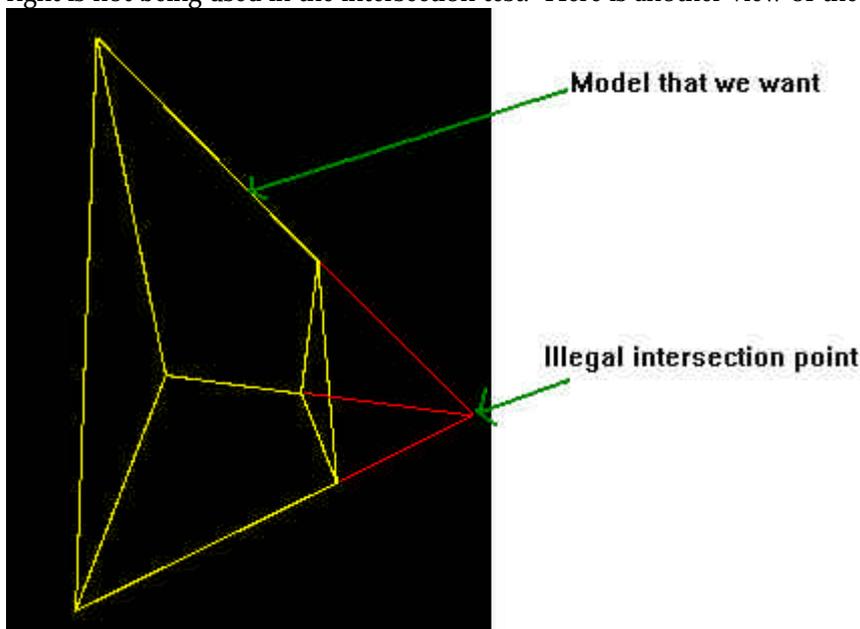
    return true;
}
```

The function returns true and sets p if there was an intersection. Otherwise, it returns false.

There is a common gotcha that must be solved when the intersection of 3 planes is used. Look at the following situation:



An extra point will be produced because three planes intersect. Unfortunately this point is not part of the object. The first plane being intersected is the horizontal line. The second plane is the diagonal line. The third plane is another plane that cannot be seen because this is a 2D view. Although there is an intersection, the point lies outside the object. The reason for this error is that the vertical plane to the right is not being used in the intersection test. Here is another view of the same object (this time, in 3D):



The **yellow** lines show the model that we want. The **red** lines show where the illegal intersection point is produced. Now that you hopefully understand the problem, I will show you the solution.

Before adding the point to the newly created polygon, a test should be made if the point is outside the object. This test can be made easily because every brush is convex. That means that if any point lies in front of any plane in the brush, it is outside the model. Think about how this is true for any convex brush. The improved pseudo code is:

Faces is an array of all the faces in the brush

Create an array Polys which has as many members as Faces

```

for i = 0 to NumberOfFaces - 1
{
    for j = 0 to NumberOfFaces - 1
    {
        for k = 0 to NumberOfFaces - 1
        {
            if ( i != j != k )
            {
                illegal = false;
                newVertex = GetIntersection ( i, j, k );

                for m = 0 to NumberOfFaces - 1
                {

```

```

        if ( DotProduct ( Faces[ m ].normal, newVertex ) + Faces[ m ].d > 0 )
        {
            illegal = true;
        }
    }
    if ( illegal == false )
    {
        Polys[ i ].AddVertex ( newVertex );
    }
}
}
}

```

There is room for optimization, although this can be data-structure dependant. One idea is instead of checking every face against all other faces in every possible combination, to have the first for loop to run through all faces from 0 to NumberOfFaces - 3. The other two for loops run through i to NumberOfFaces - 2, and j to NumberOfFaces - 1 respectively. Instead of adding the new vertex to only one polygon at a time, it can be added to all three polygons at a time. Optimized pseudo code would be:

Faces is an array of all the faces in the brush

Create an array Polys which has as many members as Faces

For i = 0 to NumberOfFaces - 3

```

{
    For j = i to NumberOfFaces - 2
    {
        For k = j to NumberOfFaces - 1
        {
            if ( i != j != k )
            {
                legal = true;
                newVertex = GetIntersection ( i, j, k );

                for m = 0 to NumberOfFaces - 1
                {
                    // Test if the point is outside the brush
                    if ( DotProduct ( Faces[ m ].normal, newVertex ) + Faces[ m ].d > 0 )
                    {
                        legal = false;
                    }
                }

                if ( legal )
                {
                    Polys[ i ].AddVertex ( newVertex ); // Add vertex to
                    Polys[ j ].AddVertex ( newVertex ); // 3 polygons
                    Polys[ k ].AddVertex ( newVertex ); // at a time
                }
            }
        }
    }
}
}

```

That wraps up how to get the polygons from a brush. The next step is to calculate the texture coordinates for every vertex.

## Calculating texture coordinates

Calculating texture coordinates for any polygon is not at all hard. I will just throw the formula at you:

$$t_u = \frac{\vec{v} \cdot \vec{n}_u}{w} / s_u + \frac{o_u}{w}$$

$$t_v = \frac{\vec{v} \cdot \vec{n}_v}{h} / s_v + \frac{o_v}{h}$$

$tu$  and  $tv$  are the two texture coordinates.  $v$  is the vertex that the texture coordinate is being calculated for.  $nu$  and  $nv$  are the texture axis' normals.  $w$  and  $h$  are the width and height, respectively, of the texture.  $su$  and  $sv$  are the texture scale.  $ou$  and  $ov$  are the texture offset or shift. To find the dimensions of the texture being used, you will have to find the texture in one of the .WAD files used by the level. To see code doing this, look at my paper on .WAD files. You can always look at how my code does it.

The problem with these texture coordinates is that they are not **normalized**. With normalized I mean as close to 0 as possible. Some graphics cards may not be able to handle extremely large texture coordinates well. If you look at the texture coordinates generated using this formula, you will see that they are correct (as long as you have texture wrapping on), but need to be put into the  $-1$  to  $+1$  range.

The way to do this is to find the coordinate closest to 0.  $U$  and  $V$  are normalized separately. If, however, a coordinate is within the  $-1$  to  $+1$  range, then they coordinates are already normalized. Normalizing is done at the polygon level. The point is to find the texture coordinate nearest to 0 (but not in the  $-1$  to  $+1$  range) and then to subtract it from every vertex in the polygon. The pseudo code to do this (only normalizes the  $U$  axis):

```
float Nearest = poly.verts[ 0 ].u;
int IndexOfNearest = 0;

for i = 0 to NumberOfVertices - 1
{
    if ( fabs ( poly.verts[ i ].u ) > 1 )
    {
        if ( fabs ( poly.verts[ i ].u ) < fabs ( Nearest ) )
        {
            IndexOfNearest = i;
            Nearest = poly.verts[ i ].u;
        }
    }
    else
    {
        // the coordinates are already normalized
        return;
    }
}

for i = 0 to NumberOfVertices - 1
{
    poly.verts[ i ].u = poly.verts[ i ].u - Nearest;
}
```

This must be done for both the  $U$  and the  $V$  axis. Again, to see this in actual C++ code, check my source code.

## Sorting vertices

If you have valid polygons with texture coordinates for every vertex, you are already pretty far. Your vertices, however, are in more or less random order. To take advantage of back face culling, the vertices of every polygon will need to be sorted in either clockwise or counterclockwise order. I prefer sorting in clockwise order, so that counterclockwise polygons can be culled. The method I describe can also be used, with some simple adjustments, to sort vertices in counterclockwise order. In order to sort vertices, you already have to know the polygon's normal. This is easy because you can just take the normal of the face that the polygon comes from.

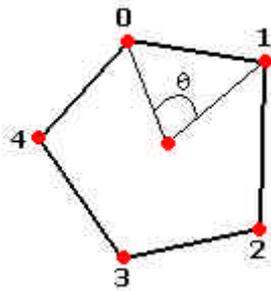
First, you must calculate the center of the polygon. This is nothing more than the average of the vertices. Pseudo code to do this would be:

```
Point center;

for n = 0 to NumberOfVertices - 1
{
    center += vertices[ n ];
}

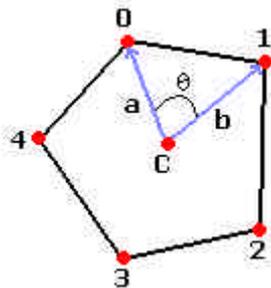
center /= NumberOfVertices;
```

Using the center of the polygon as reference, we can find the vertex with the smallest angle to the current vertex. This is repeated until the third-to-last vertex. Here is a diagram of what we are trying to find:



$\theta$  is the smallest angle. The vertex with the smallest angle to vertex 1 is vertex 2. The vertex with the smallest angle to vertex 2 is vertex 3. Now our sorted vertices list should be 0, 1, 2, 3. Only vertex 4 is left over, so it must be last.

Although this is easy to understand, how can we do it mathematically? The dot product operation can be used to find the angle between two vectors. Here is another diagram showing how to compute the angle between two vertices:



We need to find vectors **a** and **b**. This can be done by subtracting the center from vertices 0 and 1, respectively. The resulting vectors **a** and **b** should then be normalized. This all works mathematically as:

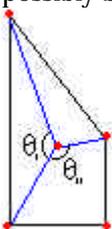
$$\vec{a} = \vec{v}_0 - \vec{c}$$

$$\vec{b} = \vec{v}_1 - \vec{c}$$

The dot product of two normalized vectors, **a** and **b**, results in a number from -1 to +1. -1 means that the vectors are at an angle of 180 degrees. 0 means that the vectors are at an angle of 90 degrees. +1 means that the vectors are at an angle of 0 degrees. So to find the angle,  $\hat{A}$ , between normalized vectors, **a** and **b**:

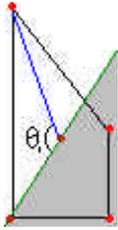
$$\hat{A} = \vec{a} \cdot \vec{b}$$

Unfortunately, there is a case when this method will fail. To eliminate the problem vertices that can't possibly be next are skipped. First, I'll show you the problem case:



Since  $\hat{q}_1 > \hat{q}_2$ , the wrong vertex will be chosen. The solution to this problem is to skip all vertices that are more than  $180^\circ$  from  $\vec{a}$ . We cannot do this using the vector dot product operation, since its result only goes from  $0^\circ$  to  $180^\circ$  (we need a full  $360^\circ$  range). However, it can be done by creating a plane on  $\vec{a}$  and along the polygon normal. The plane is perpendicular to the polygon's plane. All points under the

vertices can be skipped. The points that are not skipped will be within  $0^\circ - 180^\circ$  from  $\bar{a}$ . To help you visualize this (in a 2D case):



The points in the gray region have been skipped, since they are more than  $180^\circ$  from  $\bar{a}$ . Although I am not sure, I think this method only works for convex polygons. In pseudo-code, this whole algorithm looks like this:

```

for n = 0 to NumberOfVertices - 3
{
    Vector3 a = Normalize ( vertices[ n ] - center );
    Plane p = PlaneFromPoints ( vertices[ n ], center, center + normal );

    double SmallestAngle = -1;
    int Smallest = -1;

    for m = n + 1 to NumberOfVertices - 1
    {
        if ( p.ClassifyPoint ( vertices[ m ] ) != BACK )
        {
            Vector3 b = Normalize ( vertices[ m ] - center );
            double Angle = a.Dot ( b );

            if ( Angle > SmallestAngle )
            {
                SmallestAngle = Angle;
                Smallest = m;
            }
        }
    }

    swap ( vertices[ n + 1 ], vertices[ Smallest ] );
}

```

Although the vertices are now sorted, they may still not be in the correct order. The reason is that this code doesn't take into account the way the polygon is facing, i.e. the normal. If the polygon happens to be back facing, then the vertices are in the wrong order. This can be checked by the following pseudo code:

```

Normal n = CalculateNormal ( vertices );

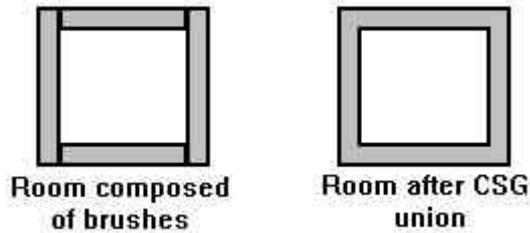
if ( n.Dot ( normal ) < 0 )
{
    ReverseVertexOrder ( vertices );
}

```

CalculateNormal will calculate the normal of a given set of vertices. ReverseVertexOrder will reverse the order of a given set of vertices. You can take a look at the function I made to sort vertices in clockwise order. It is called SortVerticesCW and can be found in the source code. If you don't know how to calculate a polygon's normal, check out <http://www.thecore.de/avgplane.pdf>.

## Performing constructive solid geometry union on all brushes

Now that we have the correct polygons (with texture coordinates) for every brush, we need to generate the correct polygons. To get the correct polygons, **constructive solid geometry (CSG) union** must be performed on all brushes in an entity. Take a look at the following diagram to see why CSG union is necessary.

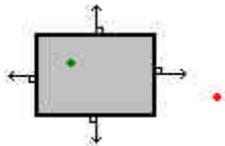


What CSG union did is it removed all overlapping polygons and polygons that were inside other brushes. Not only does this reduce the polygon count, but it also generates geometry that can be used to create BSP trees. Most engines using .MAP files for levels use BSP trees. Here is another diagram that shows what CSG union does.



There are two wide-spread algorithms that perform CSG operations on polyhedra. The **Laidlaw** approach performs CSG operations using just the polygon lists. The **Naylor** approach performs CSG operations using BSP trees. A BSP tree is created for every brush, and the BSP trees are then clipped against each other. I chose the Naylor approach, and believe it is the best approach for this situation. You may notice that the following code is very similar to the tutorial “Removing Illegal Geometry from Data imported from Quake .MAP Files” by Gerald Filimonov aka K9Megahertz (<http://www.3dtechdev.com/tutorials/illegalgeometry/illegalgeometrytut.html>). My code and ideas are based on his tutorial. It may help you to read both his tutorial and this.

Don't worry if you don't know how to create BSP trees, because in our case they are unnecessary. Both CSG algorithms were developed for any concave or convex polyhedra. However, our brushes are **convex** polyhedra. This gives us a big advantage. We can skip the whole idea of BSP trees because we know that if a point is in front of any polygon in a brush, the point lies outside the brush.



This diagram shows that if a point is in front of any polygon in the brush, it is outside the brush. If a point is in back of all polygons in a brush, it is inside the brush. This is a very neat trick we can now determine if any point, line, or polygon is inside, outside, or intersecting a brush.

The pseudo code for CSG union goes like this:

```
Brush *Brush::CSGUnion ( )
{
    ClippedBrushes = CopyList ( );           // Copy the brush array

    bool bClipOnPlane;

    for i = 0 to NumberOfBrushes - 1
    {
        bClipOnPlane = false;

        for j = 0 to NumberOfBrushes - 1
        {
            if ( i != j ) // Make sure we don't clip a brush against itself
            {
                ClippedBrushes[ i ].ClipToBrush ( Brushes[ j ], bClipOnPlane );
            }
            else
            {
                bClipOnPlane = true;
            }
        }
    }
}
```

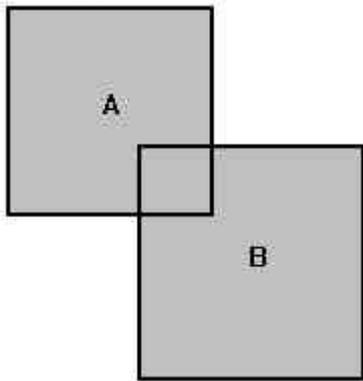
```

    }
}
// Brushes is no longer needed
return ClippedBrushes; // ClippedBrushes contains all the correct polygons
}

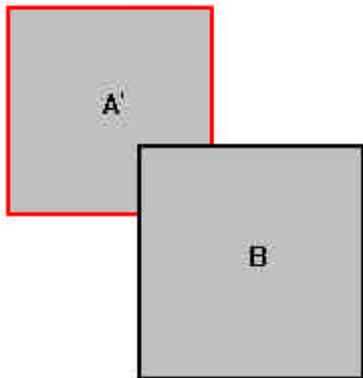
```

First, a copy of the brushes must be made, because one copy is used to clip brushes, while the other copy is clipped by brushes. Next, we must loop through all clipped brushes and brushes. It is important to check if we are trying to clip the brush against itself. If the brush is not clipping itself, then we can clip it. Once the loops are finished, the brush array is not needed anymore. The clipped brushes array contains all the correct polygons. One thing that may seem unclear is what `bClipOnPlane` does. `bClipOnPlane` is a flag that keeps track if polygons on the same plane should be kept or deleted. If every polygon on the same plane was kept, there would be multiple polygons covering the same area. If every polygon on the same plane was deleted, then there would be missing polygons. In later functions, you will see how `bClipOnPlane` is used.

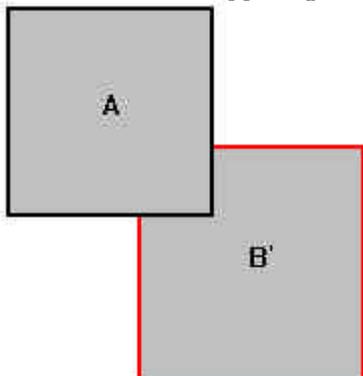
To help you visualize this process, I will draw diagrams:



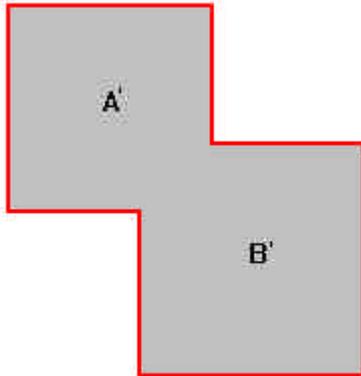
These two brushes will be clipped against each other. A copy of brushes A and B are made, brushes A' and B' respectively. First, brush A' is clipped to brush B.



Next, brush B' is clipped against A.



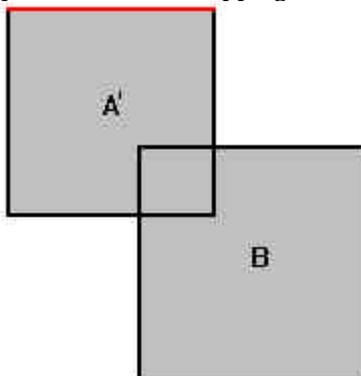
Since there are no more brushes, we are done. If we take a look at A' and B', they are correct.



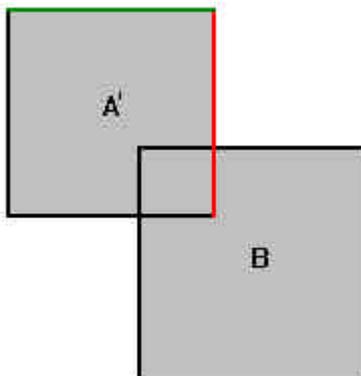
Take a look at the pseudo-code again to make sure you understand how this works. One question is left open, though. How does ClipToBrush work? ClipToBrush works like this:

```
void Brush::ClipToBrush ( Brush *pBrush, bool bClipOnPlane )
{
    for i = 0 to NumberOfPolygons - 1
    {
        Polygons[ i ] = Polygons[ i ].ClipToList ( pBrush.Polygons, bClipOnPlane );
    }
}
```

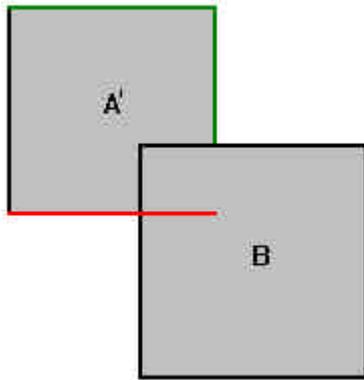
This is a simplified version of how to clip a brush to another brush. Each polygon in the brush is clipped against each polygon in the other brush. The resulting polygons (that have been clipped) are then replaced with the original polygons. I know this is confusing, but bear with me. I will try to visualize this process. We are clipping A' to B.



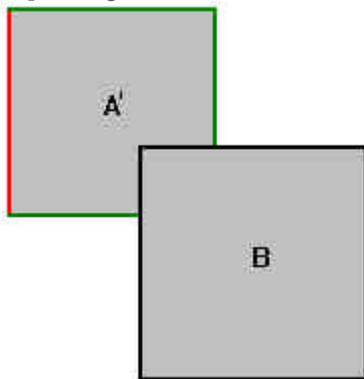
The first polygon to be clipped against B is marked in red. Since it is in front of one of B's polygons, it must be outside the brush. So we go to the next polygon in A'.



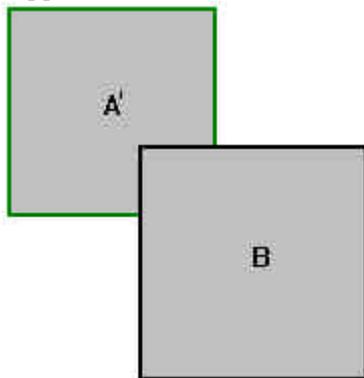
The polygon we have already clipped is green. The polygon currently being clipped is red. This polygon spans across brush B. It will have to be split. Only the split fragment(s) outside brush B are returned. In this case, it is only one fragment. On to the next polygon.



You should be starting to get a feel for this. I will continue to the next step without any further explaining.



Again we now have a simple case where the polygon is outside brush B. We are done and A' has been clipped.



The next question is how a polygon can be clipped against a list of polygons. The polygon list is a set of convex polygons forming a convex hull (the brush being clipped against). A recursive function can clip a polygon to the list. This is the pseudo-code for a function that clips a polygon to a polygon list:

```

Polygon *Polygon::ClipToList ( Polygon *pPolygon, bool bClipOnPlane )
{
    switch ( ClassifyPolygon ( polygon ) )
    {
        case FRONT:           // pPolygon is outside this brush
            return polygon;

        case BACK:
            if ( !IsLast ( ) ) // pPolygon is inside this brush
            {
                return NULL;
            }

            return pNext->ClipToList ( polygon, bClipOnPlane );

        case ONPLANE:
    }
}

```

```

double Angle = polygon.normal.Dot ( normal ) - 1;

if ( ( Angle > -epsilon ) && ( Angle < epsilon ) )
{
    if ( !bClipOnPlane )
    {
        return polygon;
    }
}

if ( !IsLast ( ) )
{
    return NULL;
}

return pNext->ClipToList ( polygon, bClipOnPlane );

case SPANNING:
    SplitPoly ( polygon, front, back );

    if ( !IsLast ( ) ) // back must be inside the brush, so only return the front split
    {
        return front;
    }

    BackFragments = pNext->ClipToList ( back, bClipOnPlane );// Clip back fragment to see if any of it will survive

    If ( BackFragments == NULL )
    {
        return front;
    }

    if ( BackFragments == back ) // There is no need to split this polygon, since all of it is in front of the brush
    {
        return polygon;
    }

    front.AddPolygon ( BackFragments ); // Add the back fragments to the front list

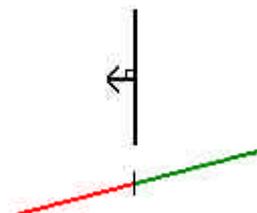
    return front;
}
}

```

The function starts by classifying the polygon to be clipped against the plane of the polygon that it is being clipped against. If the polygon is in front of the plane, the polygon can be returned unchanged, because the polygon lies outside the brush. If the polygon is in back of the plane, then it is clipped against the next polygon in the list. However, if the end of the list has been reached, it means that the polygon is inside the brush, and should be deleted.

If the polygon is on the plane, then it needs to be clipped against the next polygon in the list. If the end of the list has been reached, then the polygon can be deleted. If the polygon's normal is the same as the plane normal and bClipOnPlane is false, then the original polygon gets returned.

If the polygon is spanning the plane, then it needs to be split. The front fragment will definitely be valid, but the back fragment needs to be clipped against the next polygon in the list. If the end of the list is reached, then only the front fragment is returned, since the back fragment must be inside the brush. If none of the back fragment is returned after it has been clipped against the rest of the list, it means that the back fragment was inside the brush and only the front fragment should be returned. If the back fragment is the same as the polygons returned from clipping the back fragment, then it is unnecessary to split the polygon. To show you why, look at this case:



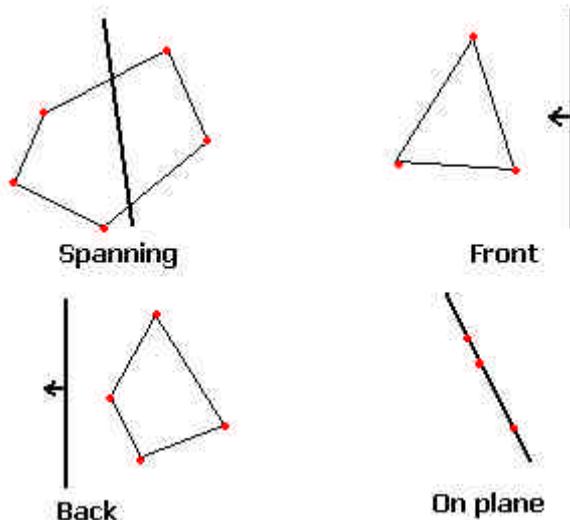
There is no point in splitting the polygon since it does not intersect the brush. The original polygon can be returned instead of the front and back splits. Doing this helps keep the polygon count to a minimum. If the end of the list has not been reached and the back fragment is not equal to the fragments returned from clipping the back fragment, then the front and the fragments returned from clipping the back fragment need to be returned.

I would like to mention that in the pseudo-code, I sometimes treated the polygon lists as arrays and sometimes as linked lists. The reason I did this was to keep the pseudo-code simple. In for loops, it is easier to use arrays, while it is easier to add an item to a linked list than to re-allocate an array to add an element. I would recommend that you use arrays to store polygon lists due to simplicity. However, you will need to write functions that deal with adding polygons to the list. In the source code, I used linked lists. However, they were harder to debug and resulted in more code (especially in for loops).

In order to really understand the algorithm, I took a piece of paper and used the algorithm step-by-step on simple cases. That really helped me understand how it works (especially when a polygon is on the plane).

## Classifying a polygon against a plane

Classifying a polygon against a plane is a very common thing to do. What you want to figure out is whether the polygon is in **front** of, in **back** of, **on**, or **spanning** the plane. Here is a 2D illustration of these four cases:



It is hard to draw the on plane case, but I hope you understand it. Classifying a polygon against a plane is pretty simple. Using the plan equation,  $\vec{n} \cdot \vec{x} + d = 0$ , we can classify any point against a plane. If the result is positive, the point is in front of the plane. If the result is negative, the point is behind the plane. If the result is zero, then the point is on the plane. The pseudo code to classify a polygon against a plane is:

```

int      iFront, iBack, iOnPlane;

for n = 0 to NumberOfVertices - 1
{
    double  result = n.Dot ( vertices[ n ] ) + d;

    if ( result > 0 )
    {
        iFront++;
    }
    else if ( result < 0 )
    {
        iBack++;
    }
    else
    {
        iOnPlane++;
    }
}

```

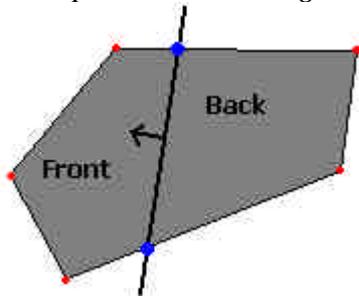
```

}
if ( iFront == NumberOfVertices )
{
    return FRONT;
}
if ( iBack == NumberOfVertices )
{
    return BACK;
}
if ( iOnPlane == NumberOfVertices )
{
    return ONPLANE;
}
return SPANNING;

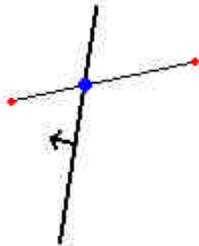
```

## Splitting a polygon by a plane

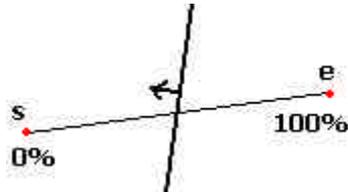
If a polygon spans a plane, you may want to split it. Splitting a polygon by a plane results in two fragments, the front and the back fragment. The front fragment is the part of the polygon that is in front of the plane. The back fragment is the part of the polygon that is in back of the plane.



Not only will the vertices have to be assigned to the correct fragment, but new vertices may have to be generated for the points where the polygon intersects the plane. These are marked as big blue dots on the above diagram. To generate a new vertex where the polygon intersects the plane, we will need to know the edge where the polygon spans the plane. We can consider this edge as a line segment which must be split by the plane.



To compute the new vertex, we first find the percentage of length of the line segment where the intersection occurs:



The formula to find this percentage is  $p = \frac{-\vec{n} \cdot \vec{s} + D}{\vec{n} \cdot \vec{d}}$ , where  $\mathbf{n}$  and  $D$  represent the plane,  $\mathbf{s}$  is the first vertex, and  $\mathbf{d}$  is the direction of the line segment.  $\mathbf{d}$  can be calculated by subtracting  $\mathbf{e}$  from  $\mathbf{s}$  and normalizing the result. Note that if the denominator is 0, then there is no intersection.  $p$  is then used to calculate the new vertex,  $\vec{v} = \vec{s} + p\vec{d}$ . If  $p$  is to be used for other calculations (such as calculating new

texture coordinates), it is necessary to normalize  $p$ ,  $p = \frac{P}{|\vec{e} - \vec{s}|}$ , where  $||$  denote the magnitude of a vector.

Now you know how to find the intersection of a line segment with a plane. It is time to show you the code to a function that will split a polygon by a plane.

```
void SplitPolygon ( Polygon* pPoly_, Plane* pPlane_, Polygon* pFront_, Polygon* pBack_ )
{
    //
    // Classify all vertices
    //
    Classify Positions[ NumberOfVertices ];

    for n = 0 to NumberOfVertices - 1
    {
        Positions[ n ] = pPlane_->ClassifyPoint ( pPoly_->vertices[ n ] );
    }

    //
    // Build front and back fragments
    //
    for n = 0 to NumberOfVertices - 1
    {
        int     m = n + 1;
        bool    bIgnore = false ;

        if ( n == NumberOfVertices - 1 )
        {
            m = 0;
        }

        switch ( Positions[ n ] )
        {
        case FRONT:
            pFront->AddVertex ( pPoly_->vertices[ n ] );

        case BACK:
            pBack->AddVertex ( pPoly_->vertices[ n ] );

        case ONPLANE:
            pFront->AddVertex ( pPoly_->vertices[ n ] );
            pBack->AddVertex ( pPoly_->vertices[ n ] );
        }

        if ( ( Positions[ n ] == ONPLANE ) && ( Positions[ m ] != ONPLANE ) )
        {
            bIgnore = true;
        }
        else if ( ( Positions[ m ] == ONPLANE ) && ( Positions[ n ] != ONPLANE ) )
        {
            bIgnore = true;
        }

        if ( ( !bIgnore ) && ( Positions[ n ] != Positions[ m ] ) )
        {
            //
            // Calculate new vertex
            //
            Vector3 d = ( pPoly_->vertices[ n ] - pPoly_->vertices[ m ] ).Normalize ( );
            double  denom = pPlane_->n.Dot ( d );

            if ( denom == 0 )
            {
                continue;
            }

            double  p = -( pPlane_->n.Dot ( pPoly_->vertices[ n ] ) + pPlane_->d ) / denom;
            Vertex  v = pPoly_->vertices[ n ] + ( p * d );

            p = p / ( End - Start ).Magnitude ( );

            //
            // Calculate new vertex's texture coordinates
            //

```

```

double du = pPoly_->vertices[ m ].tu - pPoly_->vertices[ n ].tu ;
double dv = pPoly_->vertices[ m ].tv - pPoly_->vertices[ n ].tv ;
du = du / sqrt ( du * du + dv * dv );
dv = dv / sqrt( du * du + dv * dv );

v.tu = pPoly_->vertices[ n ].tu + ( p * du );
v.tv = pPoly_->vertices[ n ].tv + ( p * dv );

//
// Add the vertex to the fragments
//
pFront_->AddVertex ( v );
pBack_->AddVertex ( v );
}
}
}

```

## Source code

The source code to this paper as well as the paper itself can be downloaded from my homepage at <http://www.uio.no/~stefanha>. The source code is my fully functional .MAP parser. It takes a .MAP file, calculates the polygons for every brush, converts the coordinates to left-handed coordinate system, scales geometry 128:1, calculates texture coordinates, performs CSG union on all brushes of every entity, and saves the entities in a .CMF file. I made the .CMF file format myself. It is just a binary file format to store a bunch of entities. Every entity can have properties and polygons. The reason I scale the geometry is that I prefer smaller coordinates.

I have also included a program called CMFView that I used to debug my CSG program with. CMFView loads .CMF files and displays them. However, I can't guarantee that CMFView will work on every PC. I also included a description of the .CMF file format. If you want, you can use my CSG program instead of writing your own. To compile a .MAP file go to DOS and type "csg mapname.map mapname.cmf", where "mapname" is your map's name. Please read the readme before using the source code or programs.

## Special Thanks

Special thanks go out to:

- The guys from the Mr. Game-Maker bulletin board. Telemachos, Gazza, Nitro, Proof, Magnus, and the rest (you know who you are). They helped me find bugs and explained lots of things to me.
- Niki from the Mr. Game-Maker bulletin board. Helped me fix bugs with sorting vertices and showed me how to calculate the average plane of a polygon (<http://www.thecore.de/avgplane.pdf>).
- Gerald Filimonov (aka K9Megahertz on the Mr. Game-Maker bulletin board) found bugs in the vertex sorting pseudocode. Go to his homepage at <http://www.3dtechdev.com>.
- BlackFiend and MasterPoi (from the Mr. Game-Maker bulletin board) found bugs and helped me fix them.
- Sean Cavanaugh from Gearbox Software. He was the first person that helped me out with .MAP files.
- Luke Hodorowicz for replying to my questions. Check out his projects at <http://gollum.flipcode.com>.
- Yahn Bernier from Valve Software. He showed me where texture coordinates were calculated in Half-Life's code.
- Ty Matthews, one of the authors of Wally, for giving me his .WAD loading code.

## Bibliography/Links

- Half-Life SDK (<http://www.valvesoftware.com/hlsdk.htm>)
- Mr. Game-Maker bulletin board (<http://www.mrgame-maker.com>)
- "Removing Illegal Geometry from Data imported from Quake .MAP Files" by Gerald Filimonov aka K9Megahertz (<http://www.3dtechdev.com/tutorials/illegalgeometry/illegalgeometrytut.html>)

- “Level Editing” by Luke Hodorowicz ([http://www.flipcode.com/tutorials/tut\\_levedit.shtml](http://www.flipcode.com/tutorials/tut_levedit.shtml))
- “Finding the Average Plane for a Set of Points” by Niki (<http://www.thecore.de/avgplane.pdf>)