VMF Files
converting brushes into a 3D mesh

jakgor471

Abstract

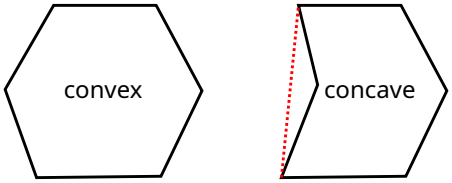
This article is meant to explain in details the process of converting brushes (convex polyhedra) defined as a set of hyperplanes into a triangulated 3D mesh using the Quake algorithm as well as describe, although partially, the structure of VMF file format. The article is based on my JavaScript implementation of the process. The links for the source code and the interactive demo are provided in the *Links* section.

The article references a work of Stefan Hajnoczi regarding the .MAP files (see *Bibliography*)

Contents

1 Introduction

Brush is a term most known from mapping software for Quake as well as Hammer Editor - a map editor for Valve's Source Engine. These programs feature a mapping paradigm sometimes called a **brush-based design** where brushes are units of map geometry and building blocks - they may be manipulated, created, deleted, made into a "physical" object or a logic one (i.e. trigger). In principle a brush is a **convex polyhedron**, meaning a 3D figure comprised of flat polygonal faces with no "indentations". The easiest way to explain convexity is to imagine wrapping the shape with a shrink wrap - the shape is convex if the shrink wrap contacts the entirety of it's surface area. If there are gaps between the surface of a shape and the shrink wrap it means the shape is **concave** and cannot be a valid brush.



Concave shape can be split into two or more convex pieces, which are in turn valid brushes, thus the convexity requirement poses no bigger limitation to mappers, a slight inconvenience at most. Additional requirement for a proper brush is that all it's sides are flat, otherwise the brush will be degenerated on construction.

Apart from flat faces there exist a method of creating smooth, organic looking geometry by subdividing all, or only selected, brush sides - this feature is called **displacements**. Displacements are the closest thing there is in Hammer Editor to **mesh-based design**, although this mechanism is burdened with severe limitations addressed later in the *Displacements* section.

2 Two main algorithms of creating 3D mesh

2.1 Quake algorithm

This article will focus mainly on the Quake method of obtaining a 3D mesh of a brush but it is worth mentioning a second method, described in details in Stefan Hajnoczi's article. The Quake algorithm constructs a polygon for each side of the brush by first creating a flat quadrilateral projected onto the plane of a given side. The quadrilateral is then repeatedly clipped against planes of each other side of the brush. Finally, an intended shape is obtained and the process is repeated for each side.

2.2 Intersection algorithm

Algorithm described by Stefan Hajnoczi takes a different approach - for each side of the brush the process is as follows: firstly the vertices are obtained by calculating the intersection points between the given side's plane and all distinct pairs of other sides' planes. Those vertices are not guaranteed to be in a proper order (clockwise or anticlockwise), thus need to be sorted, additionally, if the adjacent sides are not perpendicular (the brush is not a cuboid) the "invalid" intersection points will be generated and need to be removed. This adds a significant amount of computational complexity as well as making the algorithm itself more complex to design. Once the vertices are sorted and "invalid" vertices are removed the intended shape is obtained.

In a demo JavaScript application loading a test VMF file containing a 256 sided "sphere" is around 45 times slower using the intersection method, compared to Quake's clipping method. A question of time complexity of those two algorithms will be addressed later in the article.

3 VMF format

VMF (Valve Map Format) is a text-based file format used by **Hammer Editor** - a mapping software dedicated to Source Engine games like Garry's Mod, Half-Life 2 etc. A similar format (.MAP) is used by older editors (i.e. WorldCraft). For the sake of this article I will not describe the entirety of the format specification, as I am focusing only on the construction of a 3D mesh, thus it is not necessary to go into details.

The most important aspect of the VMF files is the way the geometry is stored - as it was described in the *Introduction* section the Hammer Editor uses brushes as units of geometry. In the VMF file a brush is defined as follows:

```
solid
{
    "id" "2"
    side
    {
        "id" "1"
        "plane" "(-3072 9216 2560) (-3072 9216 1536) (-3072 8192 1536)"
        "material" "TOOLS/TOOLSNODRAW"
        "uaxis" "[0 -1 0 0] 0.25"
        "vaxis" "[0 0 -1 0] 0.25"
        "rotation" "0"
        "lightmapscale" "16"
        "smoothing_groups" "0"
    }
    ...
    side
    {
        "id" "6"
        "plane" "(-5119.99 8192 2560) (-5119.99 8192 1536) (-4095.99 9216 1536)"
        "material" "TOOLS/TOOLSNODRAW"
        "uaxis" "[0 1 0 0] 0.25"
        "vaxis" "[0 0 -1 0] 0.25"
        "rotation" "0"
        "lightmapscale" "16"
        "smoothing_groups" "0"
    }
}
```

The definition of a brush starts with a *solid* class containing multiple *side* classes. Each side defines a *plane* described using three 3D vectors representing the vertices of a triangle. This triangle is then used to construct a plane. Apart from that a side contains information about the applied material, orientation in texture space, 2D rotation in texture space etc. which will be briefly explained later.
In pseudo-code examples the *side* class is represented as a structure:

```
struct uvaxis {
    axis : vec3,
    scale : number,
    offset : number
}

struct side {
    plane : struct plane,
    uaxis : struct uvaxis,
    vaxis : struct uvaxis,
    dispinfo : struct dispinfo,
    draw : bool
}
```

dispinfo and *plane* structures will be explained in later sections.

Note: vectors are defined in a coordinate system where right is +X, up is +Z and forward is +Y. In my code example I convert those coordinates like so:

$$\vec{p}_{\text{converted}} = \langle \vec{p}_x, \vec{p}_z, -\vec{p}_y \rangle$$

4 Essential math concepts

4.1 The dot product

The dot product is a commutative operation on two vectors, in computer graphics 3D and 2D vectors are most commonly used. The result of a dot product is a scalar, meaning a single number. The algebraic formula for 3D vectors is:

$$\vec{a} \cdot \vec{b} = a_x \times b_x + a_y \times b_y + a_z \times b_z$$

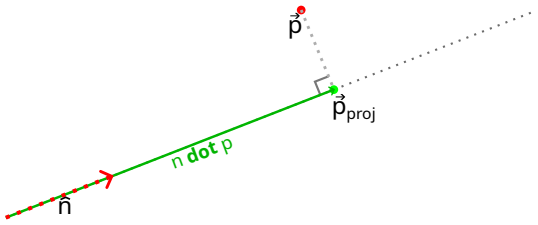
From the geometric definition we can observe that the dot product is influenced by the lengths of both vectors and a cosine of an angle θ between them:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos\theta$$

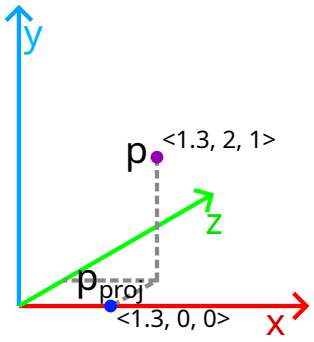
If one of those vectors has a length of 1, meaning it is a unit vector, a dot product simply tells how much does the one vector extend onto an other. This property can be used to calculate the coordinates of a vector projected onto an axis:

$$(\hat{n} \cdot \vec{p})\hat{n} = \vec{p}_{\text{proj}}$$

where $\hat{n}$ is a unit vector representing an axis, $\vec{p}$ is a vector representing a point in space and $\vec{p}_{\text{proj}}$ is a projection of $\vec{p}$ onto $\hat{n}$



Very simple way to fully understand how it works is to project a point onto a cardinal axis, let's say the X axis. A 3D coordinate space is defined by 3 axes, in most cases, perpendicular to each other. The components of a vector describe the extent along respective axes, for example:
$\vec{p} = \langle 1.3, 2, 1 \rangle$ means the vector spans 1.3 units along the X axis, 2 units along the Y and 1 unit along the Z axis. The unit vector of the X axis is $\hat{n}_x = \langle 1, 0, 0 \rangle$, thus
$\hat{n}_x \cdot \vec{p} = 1 \times 1.3 + 0 \times 2 + 0 \times 1 = 1.3$, meaning the extent of $\vec{p}$ along the X axis is 1.3. To get the coordinates of a projected point the unit vector of an axis is multiplied by the calculated dot product: $\hat{n}_x \times 1.3 = \langle 1.3, 0, 0 \rangle$.

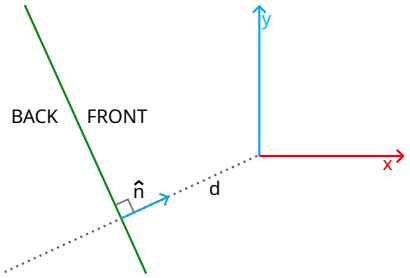


Not always a given vector we want to project a point onto happens to be a unit vector, thus it is necessary to know how to obtain such vector from an arbitrary one. A unit vector is a vector of a magnitude equal to 1. Any non-zero length vector can be turned into a unit vector by normalizing it, that is dividing each component (x, y, z) by the length of the vector:

$$\hat{a} = \frac{\vec{a}}{|\vec{a}|} = \langle \frac{a_x}{|\vec{a}|}, \frac{a_y}{|\vec{a}|}, \frac{a_z}{|\vec{a}|} \rangle$$

4.2 Plane equation

The most critical mathematical concept needed to fully understand the topic is the plane equation. In 3D graphics a plane is often conceptualized as a flat quadrilateral consisting of 4 vertices. Such representation is useful for rendering but of little use in regards to operations like clipping. In such circumstances it is more suitable to represent a plane as an equation that is true for every point laying on the surface of that plane. In such representation a plane is described using two values: a normal vector $\hat{n}$ being perpendicular to the surface of the plane and a distance d defining how far along the normal vector $\hat{n}$ the plane is positioned from the origin. Just like a 2D line divides the entire 2D space into two subspaces, a plane divides the 3D space into two distinct subspaces.



From the illustration above we notice a normal vector $\hat{n}$ plays another crucial role - it defines the front and back side of a plane. Later, during clipping this information is used to determine which portion of the polygon should be discarded.

With that in mind, let's define an equation of a plane as follows:

$$Ax + By + Cz - d = 0$$

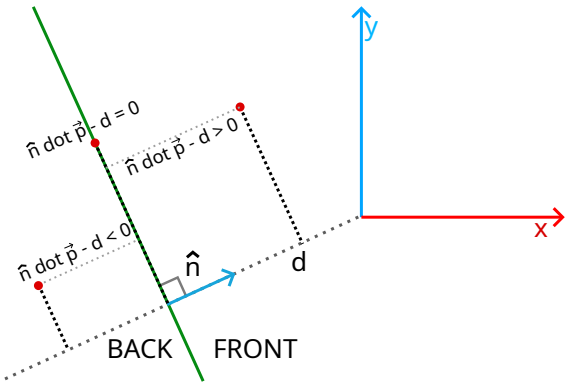
A, B and C are X, Y and Z components of a normal vector $\hat{n}$, x, y and z are coordinates of a point $\vec{p}$ in space and d is the distance. We can rewrite that equation using the dot product:

$$\hat{n} \cdot \vec{p} - d = 0$$

Substituting 0 with d_p we get a formula for calculating the distance of a point from the surface of the plane:

$$d_p = \hat{n} \cdot \vec{p} - d$$

From that we conclude that the point $\vec{p}$ lays on the surface of a plane if and only if it's distance to the given plane d_p is equal to 0. Additionally if the distance d_p is negative - the point $\vec{p}$ is in the back subspace, or in the front backspace if the distance d_p is positive.



4.3 The cross product

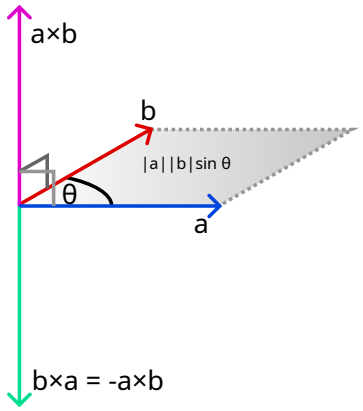
The cross product, similarly to the dot product, is an operation on two vectors but it is not commutative, meaning the order of operands matter. The result of the cross product is not a scalar but a vector. Cross products are often used in 3D graphics to calculate normal vectors of a

triangle, because given two vectors it calculates a vector perpendicular to them. The algebraic formula for 3D vectors is:

$$\vec{a} \times \vec{b} = -\vec{b} \times \vec{a} = \langle a_y b_z - a_z b_y,\, a_z b_x - a_x b_z,\, a_x b_y - a_y b_x \rangle$$

From the geometric definition we notice the resulting vector is influenced by the sine of an angle θ between the two vectors, as well as their lengths. You may recognize this as a formula for the area of parallelogram, thus the length of resulting vector is equal to an area of a parallelogram formed by two vectors:

$$\vec{a} \times \vec{b} = |\vec{a}||\vec{b}| \sin\theta$$



This property comes in handy for calculating the surface area of a triangle given it's two sides (by halving the length of their cross product). Knowing that, while calculating normal vectors, we need to be careful and almost always normalize the resulting vector, even if the operands are normalized beforehand (unless we are certain the angle between them is 90°).

To calculate the normal of a triangle defined by 3 vertices, $\vec{p}_1$, $\vec{p}_2$ and $\vec{p}_3$, first we calculate the vector from $\vec{p}_1$ to $\vec{p}_2$:

$$\vec{v}_1 = \vec{p}_2 - \vec{p}_1$$

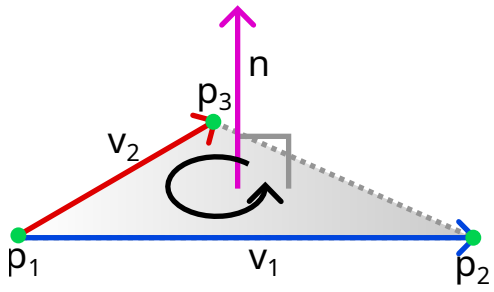
then the vector from $\vec{p}_1$ to $\vec{p}_3$:

$$\vec{v}_2 = \vec{p}_3 - \vec{p}_1$$

finally we calculate the normal $\hat{n}$ vector using the cross product, not forgetting to normalize the resulting vector afterwards:

$$\vec{n} = \vec{v}_1 \times \vec{v}_2$$

$$\hat{n} = \frac{\vec{n}}{|\vec{n}|}$$

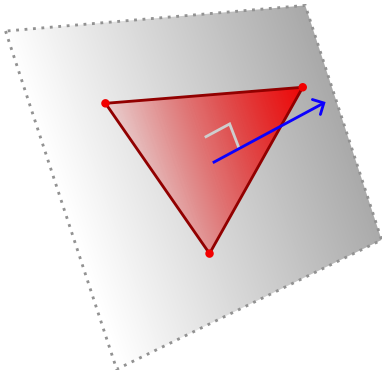


Notice the vertices are defined in an anticlockwise order when looking at the front of a triangle. This is very important in 3D graphics for the sake of back-face culling - flipping the order of vertices also flips the normal vector.

5 Constructing the sides

5.1 Constructing the planes

Armed with the knowledge about dot products, cross products and plane equation it is time to tackle the question of constructing planes from triangles defined in VMF.
One of properties of a triangle is it's vertices always lay on the same plane, thus it can be used to represent that plane - for that a normal vector $\hat{n}$ and a distance d is needed. We already know how to calculate the normal vector (see *The cross product*).
Once it is calculated we calculate the distance by projecting any given vertex of the triangle onto an axis defined by the normal vector $\hat{n}$ (see *The dot product*).

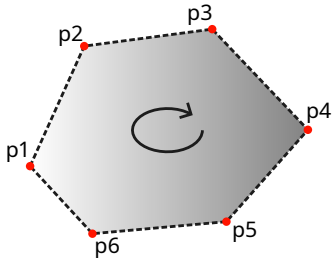


In pseudo-code a plane can be represented as a structure and constructed from 3 points as such:

```
struct plane {
    normal : vec3,
    distance : number
}

function plane::fromTriangle(p1 : vec3, p2 : vec3, p3 : vec3) : struct plane{
    let plane : struct plane = struct plane {};
    let v1 : vec3 = vec3::subtract(p2, p1);
    let v2 : vec3 = vec3::subtract(p3, p1);

    plane.normal = vec3::crossProduct(v1, v2);
    plane.distance = vec3::dotProduct(plane.normal, p1);

    return plane;
}
```

5.2 Windings

In Quake 2 Tools' source code a polygon is represented using a structure called *winding_t* (see *common/polylib.h*). A winding is a set of vertices in order (clockwise or anticlockwise) which can be then trivially triangulated using e.g. fan triangulation algorithm (assuming the represented polygon is convex). In this article windings are in clockwise order but are then triangulated into anticlockwise triangles.



[p1, p2, p3, p4, p5, p6]

To construct a mesh of a brush's side a quadrilateral winding needs to be created first based on the side's plane. Such winding is called the **base winding**. Since brushes are positioned and constructed in worldspace coordinates, the base winding must span at least the entire map volume.

Let's consider a side aligned with the XY plane of the coordinate system. To calculate the position of the winding's vertices the up and right direction vectors are needed in the side's, or rather it's plane's, coordinate space. In our simplified example those vectors are trivial to find: the up vector is $\hat{v}_{\text{up}} = \langle 0, 1, 0 \rangle$ and the right vector is $\hat{v}_{\text{right}} = \langle 1, 0, 0 \rangle$. Consecutive vertex positions are (anticlockwise order):

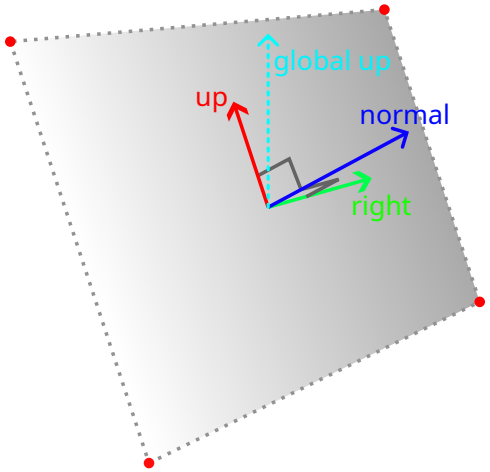
$$\vec{p}_1 = (-\hat{v}_{\text{right}} + \hat{v}_{\text{up}}) \times M$$

$$\vec{p}_2 = (-\hat{v}_{\text{right}} - \hat{v}_{\text{up}}) \times M$$

$$\vec{p}_3 = (\hat{v}_{\text{right}} - \hat{v}_{\text{up}}) \times M$$

$$\vec{p}_4 = (\hat{v}_{\text{right}} + \hat{v}_{\text{up}}) \times M$$

where M is the halved map size (assuming the map volume is a cube of side-length equal to $2M$). In Quake 2 Tools' source code the M is equal to 8192 units, meaning the map volume is 16384^3 units.
Since the plane in our example is positioned flush on the XY plane it is not necessary to offset the vertices along the plane's normal.
In practical use we cannot rely on planes being aligned with the coordinate system, thus a generalized approach for finding those vectors is needed based only on the normal vector.



Firstly we notice the up and right vectors are perpendicular to the normal vector, the right vector is perpendicular to the global up vector, thus can be constructed by taking the cross product of normal and global up. Plane's up vector is then constructed by calculating the cross product between the normal vector and the right vector.
A problem with this approach arrives when the global up and the normal vector are collinear (or very close to being collinear due to floating point inaccuracy) - in such case the area of a parallelogram formed by these two vectors will be zero, thus the resulting right vector will be of zero length. The solution is to make the angle between the normal vector and the global up vector as close as possible to 90°. The easiest way to ensure that is to first determine the **major axis** of the normal vector, that is - the axis along which the vector spans the most. If the major axis is X or Z - the global up vector is $\langle 0, 1, 0 \rangle$, otherwise it is set to $\langle 1, 0, 0 \rangle$. In pseudo-code this process is as follows:

```
function getGlobalUp(normal : vec3) : vec3{
    let global_up : vec3 = vec3(0, 1, 0);
    let axis = -1;
    let max = -MAXVALUE;

    for(let i = 0; i < 3; ++i){
        //vector components can be accessed using array subscript
        let absval = abs(normal[i]);
```

```
        if(absval > max){
            max = absval;
            axis = i;
        }
    }

    if(axis == -1) //Invalid vector!
        return null;

    if(axis == 1) //if Y is major axis - change the global_up
        global_up = vec3(1, 0, 0);

    return global_up;
}
```

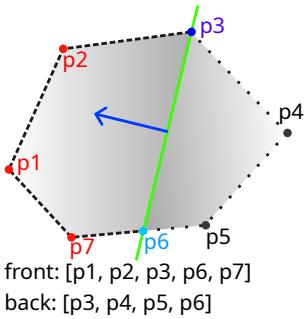
Having the two direction vectors calculated the coordinates of four points are similar to the above example:

$$\vec{p_1} = (-\hat{v}_{\text{right}} + \hat{v}_{\text{up}}) \times M + \hat{n}d$$
$$\vec{p_2} = (-\hat{v}_{\text{right}} - \hat{v}_{\text{up}}) \times M + \hat{n}d$$
$$\vec{p_3} = (\hat{v}_{\text{right}} - \hat{v}_{\text{up}}) \times M + \hat{n}d$$
$$\vec{p_4} = (\hat{v}_{\text{right}} + \hat{v}_{\text{up}}) \times M + \hat{n}d$$

The only difference is this time the points are offset along the normal vector $\hat{n}$ by the distance d. Vertices of the constructed winding are in clockwise order.

5.3 Clipping

Once the winding is constructed it is ready to be clipped against all other sides of a brush it is a part of. By leaving only the "front portions" of a winding with every cut we finally obtain the intended shape of the side. You can see the process in details in the JavaScript demo by running the animation.

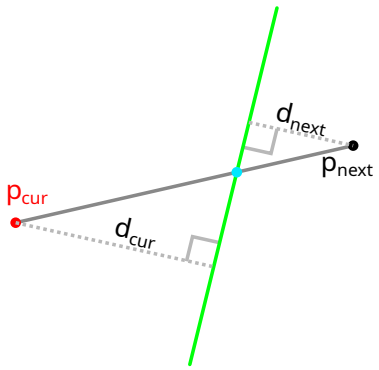


Clipping is an operation between a winding and a plane defined by normal vector $\hat{n}$ and a distance d. Firstly the vertices of the winding need to be classified based on their distance to the plane: if the distance is positive the vertex is classified as "FRONT", if the distance is negative the vertex is classified as "BACK", otherwise the vertex is classified as "ON", meaning it is laying on the surface of the plane. In practical implementation, due to floating point inaccuracy, those comparisons are done with an epsilon $\mathcal{E}$ value (in my code example $\mathcal{E} = 2^{-14}$). In pseudo-code this procedure is as follows:

```
function clipWinding(winding : array, plane : struct plane) : array{
    const SIDE_BACK = 0;
    const SIDE_FRONT = 1;
    const SIDE_ON = 2;

    //sides is an array for classifying the vertices, it's length is
    //set to the length of a winding + 1 to wrap around.
    let sides : array = [winding.length + 1];

    //same with dists holding precomputed vertex distances to the clipping plane
    let dists : array = [winding.length + 1];
```

```
let i = 0;
for(; i < winding.length; ++i){
    let dist = vec3::dotProduct(plane.normal, winding[i]) - plane.distance;
    dists[i] = dist;

    if(dist > EPSILON)
        sides[i] = SIDE_FRONT;
    else if(dist < -EPSILON)
        sides[i] = SIDE_BACK;
    else
        sides[i] = SIDE_ON;
}
//wrap around to avoid modular arithmetic
sides[i] = sides[0];
dists[i] = dists[0];

//...
```

Once the vertices are classified we can proceed to create the clipped winding. Since the sides are constructed only from the front portions of a winding we do not bother creating the back winding. Now we iterate over the vertices once again: if the current vertex is classified as "ON" - it gets added to the front winding and we move on to another vertex (continue). If the current vertex is classified as "FRONT" it gets added to the front winding, but this time we also check the next consecutive vertex: if the next vertex is also classified as "FRONT" or "ON", meaning it is on the same side as the current vertex, then we continue onto the next vertex. If however the next vertex is classified as "BACK", meaning the plane is intersecting an edge between the current vertex and the next one, we need to generate the split point.



To calculate the position of the split point $\vec{p}_{\text{split}}$ firstly a vector $\vec{v}$ from $\vec{p}_{\text{cur}}$ to $\vec{p}_{\text{next}}$ is obtained:

$$\vec{v} = \vec{p}_{\text{next}} - \vec{p}_{\text{cur}}$$

d_{cur} and d_{next} are distances of respective points to the clipping plane. The difference of those two distances is proportional to the length of $\vec{v}$. Let $\vec{v}_2$ be the vector from $\vec{p}_{\text{cur}}$ to $\vec{p}_{\text{split}}$, it's length is proportional to d_{cur}.
We can find the position of $\vec{p}_{\text{split}}$ using linear interpolation between $\vec{p}_{\text{cur}}$ and $\vec{p}_{\text{next}}$ using the fraction t:

$$t = \frac{|\vec{v}_2|}{|\vec{v}|} = \frac{d_{\text{cur}}}{d_{\text{cur}} - d_{\text{next}}}$$

thus:

$$\vec{p}_{\text{split}} = \vec{p}_{\text{cur}} + t\vec{v} = \vec{p}_{\text{cur}} + t(\vec{p}_{\text{next}} - \vec{p}_{\text{cur}})$$

The calculated split point is added to the front winding and we continue onto the next vertex. In pseudo-code this process is as follows (continuation of the code block above):

```
//...

let front : array = [];
for(i = 0; i < winding.length; ++i){
    let p_cur : vec3 = winding[i];

    //current point is on the surface, add it and continue
```

```
        if(sides[i] == SIDE_ON){
            front.push(p_cur);
            continue;
        }

        if(sides[i] == SIDE_FRONT)
            front.push(p_cur);

        //if the next vertex is on the plane or on the same side - continue
        if(sides[i + 1] == SIDE_ON or sides[i] == sides[i + 1])
            continue;

        //WE NEED TO GENERATE THE SPLIT POINT
        //get the next vertex, wrap around if needed
        let p_next : vec3 = winding[(i + 1) mod winding.length];
        let t = dists[i] / (dists[i] - dists[i + 1]);

        //vector from p_cur to p_next
        let v : vec3 = vec3::subtract(p_next, p_cur);
        vec3::scale(v, t);

        let p_split : vec3 = vec3::add(p_cur, v);
        front.push(p_split);
    }

    return front;
}
```

5.4 Rounding

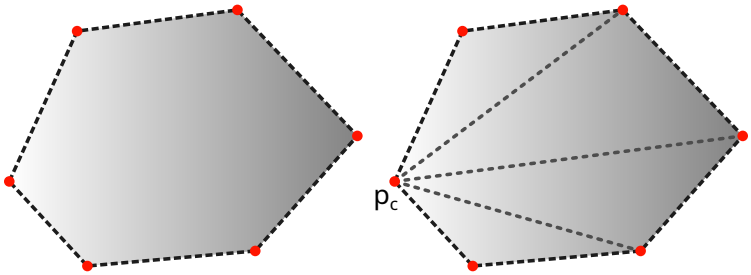
After the clipping step is done and the final winding is obtained, the coordinates of it's vertices should be rounded, or in other words - snapped to the smallest grid. In my code example the coordinates are rounded to $2^{-7} = \frac{1}{128}$ of a unit.

```
function vec3::round(v : vec3) : vec3{
    const ROUNDING = 128;
    const ROUNDING_REC = 1.0 / ROUNDING;

    v[0] = round(v[0] * ROUNDING) * ROUNDING_REC;
    v[1] = round(v[1] * ROUNDING) * ROUNDING_REC;
    v[2] = round(v[2] * ROUNDING) * ROUNDING_REC;

    return v;
}
```

6 Triangulation

As stated in the *Winding* section, a winding can be trivially triangulated. There exist many algorithms to perform that task but the simplest, in my opinion, is the **fan triangulation algorithm**. Given a set of $n \geq 3$ ordered vertices, representing a winding of a polygon, the algorithm generates $n - 2$ triangles. The implementation given in this article is guaranteed to work only on convex polygons but since a concave polygon cannot be constructed in the manner described in *Clipping*, this limitation does not need to be addressed.

The first step of the algorithm is choosing an arbitrary point p_c of a winding as the common vertex of all generated triangles (in computer science "arbitrary" means the "first one"), then the edges are created from p_c to each point not sharing an edge with p_c.



The algorithm in pseudo-code:

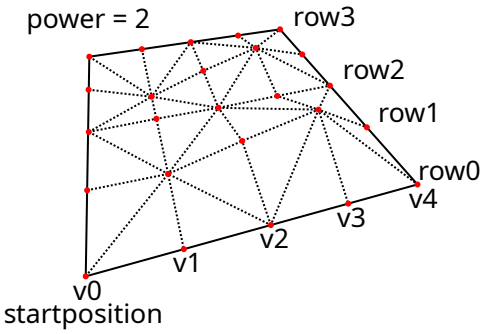
```
function triangulate(winding : array) : array{
    if(winding.length < 3)
        return null;
    let mesh : array = [];
    //set the first vertex as the common vertex
    let p_c : vec3 = winding[0];

    //start from the second vertex
    for(let i = 1; i < winding.length - 1; ++i){
        mesh.push(p_c);
        //ensure anticlockwise order
        mesh.push(winding[i + 1]);
        mesh.push(winding[i])
    }

    return mesh;
}
```

In computer graphics the complex polygons are broken down into triangles due to it's simple geometric nature and consistency - all triangles have exactly 3 vertices, thus given a continuous list of vertex positions the graphics pipeline does not require an information about how many vertices should the shape be. This is very useful, because by converting all the windings into a list of triangles, we can render the entire brush, if not the whole map, using a single command.

Triangulation is the final step in building the mesh of a side, once it is ready to be rendered, and the process applies only to non-displacement sides.

7 Displacements

As stated in *Introduction* section, the displacement is a mechanism allowing mappers to create a smooth, organic looking geometry suitable for representing e.g. the terrain. Displacements transform a flat side of a brush into a subdivided mesh of triangles, of which the vertices can be manipulated individually, additionally, vertices may be assigned a different alpha values used for blending two materials. This feature can be used only on flat side with exactly 4 edges, otherwise the displacement is not valid and should not be created.



Displacements are built "on top" of a side's winding after it has been constructed and clipped, and since the displacements divide the side into a grid of triangles, further triangulation is not needed.

It is worth mentioning that only the displaced sides are rendered if the brush contains any displacements. Other sides exist only for the sake of constructing the brush.

Information about displacements is stored in VMF using a *dispinfo* class, being an optional member of *side* class:

```
side
{
    "id" "306"
    "plane" "(96 -256 0) (-96 -256 0) (-256 256 256)"
    "material" "BRICK/BRICKFLOOR001A"
    "uaxis" "[1 0 0 0] 0.25"
    "vaxis" "[0 -0.894427 -0.447214 0] 0.25"
    "rotation" "0"
    "lightmapscale" "16"
    "smoothing_groups" "0"
    dispinfo
    {
        "power" "2"
        "startposition" "[-96 -256 -0]"
        "flags" "0"
        "elevation" "0"
        "subdiv" "0"
        normals
        {
            //...
        }
        distances
        {
            //...
        }
        offsets
        {
            //...
        }
        offset_normals
        {
            //...
        }
        alphas
        {
            //...
        }
        triangle_tags
        {
            //...
        }
        allowed_verts
        {
            //...
        }
    }
}
```

- *power* is a value representing the level of subdivision - a side is subdivided into 2^n rows of 2^{n+1} triangles, meaning the total number of triangles is 2^{2n+1}, where n is equal to *power*. Number of vertices per row and per column is $2^n + 1$. Only allowed powers are 2, 3 and 4.
- *startposition* defines a position of a bottom left corner of the side, from which the displacement is built. It is represented using 3 coordinates: X, Y and Z.
- *elevation* is an offset uniformly applied to every vertex along it's *offset_normal* vector.

Following data are member classes of the *dispinfo* class.

- *normals* define normal vectors for every point of the displacement:

```
normals
{
    //      X0 Y0 Z0 X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3 X4 Y4 Z4
    "row0" "0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
    "row1" "0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0"
    "row2" "0 0 0 0 -1 0 0.620766 -0.395033 0.677199 0 0 0 0 0 0"
    "row3" "0 0 0 0 0 0 0 0 0 0 0 1 0 0 0"
```

```
    "row4" "0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
}
```

These vectors are used in conjunction with values from *distances* class to position the vertices.
Note: only the vertices that has been moved from their original position have the normals set, others are left as $\langle 0, 0, 0 \rangle$.

- *distances* indicate how much each vertex is moved along it's normal vector:

```
distances
{
    //      v0 v1 v2 v3 v4
    "row0" "0 0 0 0 0"
    "row1" "0 80 0 0 0"
    "row2" "0 20 88.6002 0 0"
    "row3" "0 0 0 45 0"
    "row4" "0 0 0 0 0"
}
```

The direction of movement should be indicated by the normal vector, thus we may assume the distance values are non-negative.

- *offsets* contain vectors for each vertex, used to offset the vertex relative to it's calculated position:

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

- *offset_normals* is a class similar to *normals*. These vectors are used to apply the elevation and they seem to always match the original normal vector of a side (although pointing in the opposite direction - away from the brush's center):

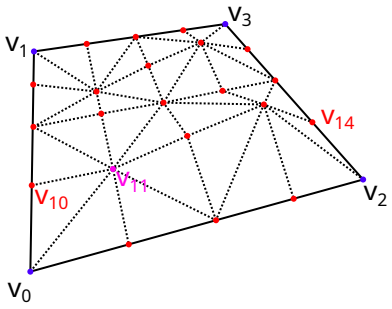
```
offset_normals
{
    //      X0 Y0 Z0 X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3 X4 Y4 Z4
    "row0" "0 -0.447214 0.894427 0 -0.447214 0.894427 0
            -0.447214 0.894427 0 -0.447214 0.894427 0 -0.447214 0.894427"
    "row1" "0 -0.447214 0.894427 0 -0.447214 0.894427 0
            -0.447214 0.894427 0 -0.447214 0.894427 0 -0.447214 0.894427"
    "row2" "0 -0.447214 0.894427 0 -0.447214 0.894427 0
            -0.447214 0.894427 0 -0.447214 0.894427 0 -0.447214 0.894427"
    "row3" "0 -0.447214 0.894427 0 -0.447214 0.894427 0
            -0.447214 0.894427 0 -0.447214 0.894427 0 -0.447214 0.894427"
    "row4" "0 -0.447214 0.894427 0 -0.447214 0.894427 0
            -0.447214 0.894427 0 -0.447214 0.894427 0 -0.447214 0.894427"
}
```

Above structures and values are sufficient to calculate the final position of each vertex according to the formula:

$$\vec{v}_{\mathrm{ij}} = \vec{v}_{\mathrm{ij_{orig}}} + d_{\mathrm{ij}}\hat{n}_{\mathrm{ij}} + d_{\mathrm{elev}}\hat{n}_{\mathrm{ij_{off}}} + \vec{v}_{\mathrm{ij_{off}}}$$

where $\vec{v}_{\mathrm{ij_{orig}}}$ is the original position of the vertex (calculated using bilinear interpolation), d_{ij} is the distance defined in *distances*, $\hat{n}_{\mathrm{ij}}$ is the vertex normal defined in *normals*, d_{elev} is the displacement *elevation*, $\hat{n}_{\mathrm{ij_{off}}}$ is the vertex offset normal defined in *offset_normals* and $\vec{v}_{\mathrm{ij_{off}}}$ is the vertex offset defined in *offsets*.
ij indices denote the rows and columns.

The original vertex positions, laying flat on the side, are calculated using bilinear interpolation between $\vec{v}_0$, being a starting vertex, $\vec{v}_1$ being the top left corner, $\vec{v}_2$ being a bottom right corner and $\vec{v}_3$ - the top right corner.



Firstly, a bottom left vertex $\vec{v}_0$ (starting vertex) of a winding needs to be found by searching for the vertex closest to *startposition*. Since the winding is in anticlockwise order, the top left corner $\vec{v}_1$ is the vertex following the $\vec{v}_0$. The bottom right vertex $\vec{v}_2$ is a vertex preceding the $\vec{v}_0$ and the top right vertex $\vec{v}_3$ is a vertex preceding the $\vec{v}_2$.

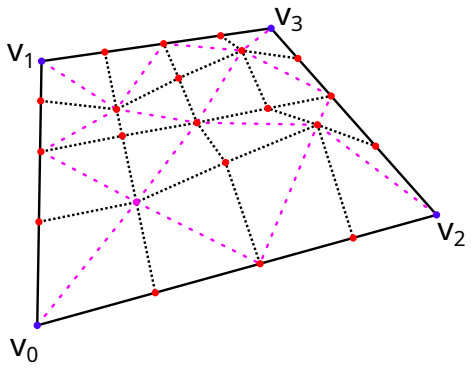
For interpolating between the vertices a fraction $t = \frac{1}{n_{\text{verts}}}$ is needed, where n_{verts} is the number of vertices per row (or column, as these values are equal). For example, to get the position of vertex $\vec{v}_{11}$ first we interpolate t steps from $\vec{v}_0$ to $\vec{v}_1$ getting the position of $\vec{v}_{10}$. Repeating this step for $\vec{v}_2$ and $\vec{v}_3$ we get the position of $\vec{v}_{14}$. Now, interpolating t steps between $\vec{v}_{10}$ and $\vec{v}_{14}$ we get to the $\vec{v}_{11}$ vertex.

Following pseudo-code implements this algorithm, generating the vertex positions:

```
function createDisplacementMesh(winding : array, dispinfo : struct dispinfo)
    : array{
    //displacements must be created on sides with 4 vertices only!
    if(winding.length != 4)
        return null;

    //first we find the starting vertex
    let startIndex = 0;
    let closestDist = MAXVALUE;

    for(let i = 0; i < winding.length; ++i){
        //we can use the squared distance for comparisons
        let dist = vec3::distanceSqr(winding[i], dispinfo.startposition);

        if(dist < closestDist){
            startIndex = i;
            closestDist = dist;
        }
    }

    let v0 : vec3 = winding[startIndex]; //starting vertex
    let v1 : vec3 = winding[(startIndex + 1) mod winding.length];
    let v2 : vec3 = winding[(startIndex + 3) mod winding.length];
    let v3 : vec3 = winding[(startIndex + 2) mod winding.length];

    let vPerRow = 2^dispinfo.power + 1;
    let vertices : matrix = [vPerRow][vPerRow] //2d array, rows, columns
    let t = 1.0 / vPerRow; //fraction t for interpolation

    let elevation = dispinfo.elevation;

    //for every row
    for(let i = 0; i < vPerRow; ++i){
        //calulcate interpolated vertical points
        let vInt0 : vec3 = vec3::lerp(v0, v1, t * i);
        let vInt1 : vec3 = vec3::lerp(v2, v3, t * i);

        //generate row of vertices (horizontal points)
        for(let j = 0; j < vPerRow; ++j){
            let vOrig : vec3 = vec3::lerp(vInt0, vInt1, t * j);

            let distance = dispinfo.distances[i][j];
            let normal : vec3 = dispinfo.normals[i][j];
            let offset_normal : vec3 = dispinfo.offset_normals[i][j];
            let offset : vec3 = dispinfo.offsets[i][j];
```

```
            let vertex : vec3 = vec3::add(vOrig, vec3::scale(normal, distance));
            vertex = vec3::add(vertex, vec3::scale(offset_normal, elevation));
            vertex = vec3::add(vertex, offset);

            vertices[i][j] = vertex;
        }
    }

    //...
```

Having the vertex positions calculated, now it is time to triangulate them to obtain the mesh. Notice, if we remove the diagonal edges (highlighted in purple), the displacement is a grid of quads. Bottom left quad has the diagonal edge going right, the next one going left etc. The orientation of diagonals in the next row is inverted and the pattern continues:




```
    //...
    //quads per row = vertices per row - 1
    let quadsPerRow = vPerRow - 1;
    //total number of triangles = quadsPerRow ^ 2 * 2

    let mesh : array = [];

    //first quad of the first row has the diagonal going right
    let left = false;
    for(let i = 0; i < quadsPerRow; ++i){
        for(let j = 0; j < quadsPerRow; ++j){
            //reuse v0-v3 as vertices of a current quad
            v0 = vertices[i][j]; //bottom left
            v1 = vertices[i + 1][j] //top left
            v2 = vertices[i + 1][j + 1] //top right
            v3 = vertices[i][j + 1] //bottom right

            if(left){
                //diagonal going left, anticlockwise order
                mesh.push(v3);
                mesh.push(v0);
                mesh.push(v1);
                mesh.push(v3);
                mesh.push(v2);
                mesh.push(v1);
            } else {
                //diagonal going right
                mesh.push(v0);
                mesh.push(v2);
                mesh.push(v1);
                mesh.push(v0);
                mesh.push(v3);
                mesh.push(v2);
            }

            left = !left //alternate the diagonal direction
        }

        //start the next row with the same diagonal direction
        //as the last quad of previous row
        left = !left;
    }
```

```
    return mesh;
}
```

The last structure contained in *dispinfo* discussed in this article is *alphas* class. It contains information about alpha value for every vertex, expressed as a number ranging from 0-255, controlling the blending between two materials:

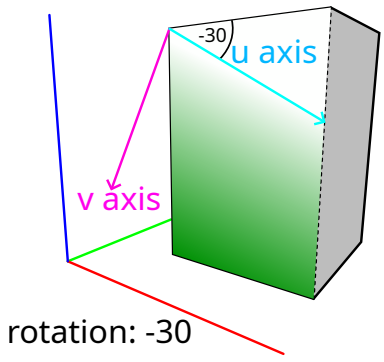
```
alphas
{
    "row0" "255 55 0 0 0"
    "row1" "255 0 0 0 255"
    "row2" "255 255 255 255 255"
    "row3" "0 0 80 180 255"
    "row4" "0 0 0 80 0"
}
```

8 Texture coordinates

My code, on which the article is based, do not feature any texture mapping, but for the sake of completeness I decided to cover the topic.
UV coordinates dictate how the texture is applied to every visible side of a brush, including displacement sides. *side* class contains *uaxis* and *vaxis* defining the respective UV axes onto which the world space coordinates are projected to obtain the final UV coordinates of a vertex. Additionally, a translation along the axes and scaling is provided:

```
side
{
    "id" "306"
    "plane" "(-256 256 256) (256 256 256) (96 -256 0)"
    "material" "NATURE/BLENDDIRTGRASS005A"

    \\      [X Y Z translation] scale
    "uaxis" "[1 0 0 0] 0.25"
    "vaxis" "[0 -0.894427 -0.447214 0] 0.25"
    "rotation" "0"
    "lightmapscale" "16"
    "smoothing_groups" "0"
}
```

rotation serves no purpose for actual UV coordinates, as the texture rotation is already reflected in U and V axes.



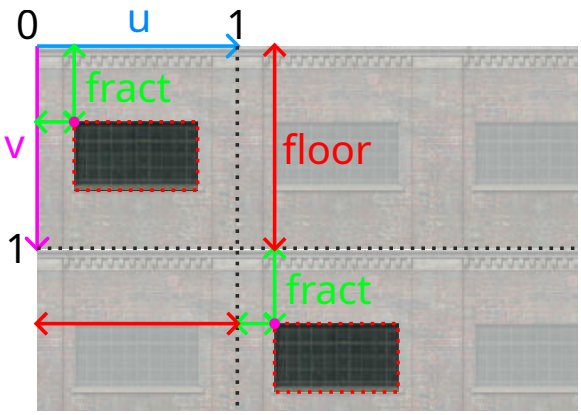
Important aspect to note is, final UV coordinates are calculated based on the size (width and height) of the applied texture, meaning a bigger resolution texture will not look the same as a smaller resolution one, if the texture size is not taken into account.

The formulas for U and V coordinates are:

$$t_u = \frac{\vec{v} \cdot \hat{u}}{ws_u} + \frac{o_u}{w}$$

$$t_v = \frac{\vec{v} \cdot \hat{v}}{h s_v} + \frac{o_v}{h}$$

where $\vec{v}$ are the world space coordinates of a vertex, $\hat{u}$ and $\hat{v}$ are UV axes, s_u and s_v are UV scaling factors, o_u and o_v are UV offsets and w, h are texture width and height.



Generated UV coordinates may be far away from the origin of UV coordinate system. Ideally, UV coordinates should be contained between 0.0 - 1.0 but normalizing them may result in issues with texture scale and tiling, thus we should limit ourselves to translating the texture coordinates, so that they are as close as possible to the origin. Firstly, the smallest t_u and t_v values need to be found ($t_{\mathrm{u_{min}}}$ and $t_{\mathrm{v_{min}}}$). U and V offsets are calculated by flooring the $t_{\mathrm{u_{min}}}$ and $t_{\mathrm{v_{min}}}$:

$$o_u = \lfloor t_{\mathrm{u_{min}}} \rfloor$$

$$o_v = \lfloor t_{\mathrm{v_{min}}} \rfloor$$

lastly, all texture coordinates are offset by subtracting o_u and o_v from respective coordinates.

This procedure in pseudo-code:

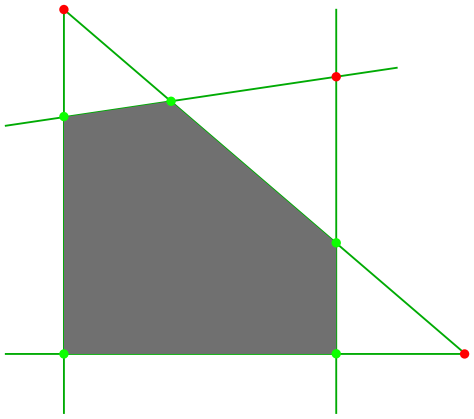
```
function calcUVCoords(winding : array, side : struct side, texW, texH) : array{
    let uvcoords : array = [winding.length];
    let uMin = MAX_VALUE;
    let vMin = MAX_VALUE;

    for(let i = 0; i < winding.length; ++i){
        let uv : vec2;
        uv[0] = vec3::dotProduct(winding[i], side.uaxis.axis)
                / (texW * side.uaxis.scale) + side.uaxis.offset / texW;
        uv[1] = vec3::dotProduct(winding[i], side.vaxis.axis)
                / (texH * side.vaxis.scale) + side.vaxis.offset / texH;

        uvcoords[i] = uv;

        //find the smallest u and v
        if(uv[0] < uMin)
            uMin = uv[0];
        if(uv[1] < vMin)
            vMin = uv[1];
    }

    let uOff = floor(uMin);
    let vOff = floor(vMin);

    for(let i = 0; i < uvcoords.length; ++i){
        uvcoords[i][0] = uvcoords[i][0] - uOff;
        uvcoords[i][1] = uvcoords[i][1] - vOff;
    }

    return uvcoords;
}
```

Texture coordinates are calculated before the triangulation to avoid redundant computations. This also applies to displacements - texture coordinates of each vertex are calculated the same way as original vertex positions - by bilinear interpolation between the side vertices.

9 The intersection algorithm

9.1 Overview

As stated in the *Two main algorithms of creating 3D mesh* section, the intersection algorithm takes a different approach - instead of clipping a big polygon aligned with the plane of a side, an intersection point between the current side plane and each unique pair of other side planes is calculated. After calculating each point a check must be performed to ensure the point is contained in the brush's volume. This process involves checking the distance of the point to every plane. If this distance is negative, or in practical use smaller then the epsilon $\mathcal{E}$, the point is discarded. Such invalid points are often generated if the adjacent sides are not perpendicular.



On the illustration the points highlighted in red are invalid and need to be removed. Only the green points contribute to the final shape of the side.

After the points are generated they are not guaranteed to be in any order, therefor it is necessary to sort them clockwise or anticlockwise. Finally, a winding is obtained and the rest of the process is the same for both the Quake algorithm and the intersection algorithm.

Due to numerical imprecision, the intersection method uses a different epsilon $\mathcal{E}$ value equal to 2^{-8}. This value was chosen based on testing and observations.

9.2 Intersection point formula

The formula for finding an intersection point between three planes, given the planes are not parallel to each other, can be obtained by solving the system of linear equations:

$$\begin{cases} A_1x + B_1y + C_1z = D_1 \\ A_2x + B_2y + C_2z = D_2 \\ A_3x + B_3y + C_3z = D_3 \end{cases}$$

where A_i, B_i, C_i and D_i describe the components of i-th plane normal vector and it's distance. x, y and z are coordinates of a point - these values need to be found. This system of equations can be expressed in matrix form:

$$\mathbb{A} = \begin{bmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{bmatrix}, \vec{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \mathbb{C} = \begin{bmatrix} D_1 \\ D_2 \\ D_3 \end{bmatrix}$$

$$\mathbb{A} \times \vec{p} = \mathbb{C} \implies \vec{p} = \mathbb{A}^{-1} \times \mathbb{C}$$

The inverse of $\mathbb{A}$ is obtained by constructing the adjugate of $\mathbb{A}$ and multiplying it by the inverse of $det\mathbb{A}$:

$$\mathbb{A}^{-1} = (det\mathbb{A})^{-1} \begin{bmatrix} B_2C_3 - B_3C_2 & B_3C_1 - B_1C_3 & B_1C_2 - B_2C_1 \\ A_3C_2 - A_2C_3 & A_1C_3 - A_3C_1 & A_2C_1 - A_1C_2 \\ A_2B_3 - A_3B_2 & A_3B_1 - A_1B_3 & A_1B_2 - A_2B_1 \end{bmatrix}$$

Since the absolute value of determinant is equal to the volume of parallelepiped formed by three column vectors, if all three plane normals are coplanar the determinant is equal to 0, thus the inverse of $\mathbb{A}$ cannot be calculated and intersection point cannot be found.

Notice the columns of the adjugate matrix form vectors equal to, respectively, $\hat{n}_2 \times \hat{n}_3$, $\hat{n}_3 \times \hat{n}_1$ and $\hat{n}_1 \times \hat{n}_2$. It will be important in a moment.
Combining the formula we get:

$$\vec{p} = (det\mathbb{A})^{-1} \begin{bmatrix} B_2C_3 - B_3C_2 & B_3C_1 - B_1C_3 & B_1C_2 - B_2C_1 \\ A_3C_2 - A_2C_3 & A_1C_3 - A_3C_1 & A_2C_1 - A_1C_2 \\ A_2B_3 - A_3B_2 & A_3B_1 - A_1B_3 & A_1B_2 - A_2B_1 \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \\ D_3 \end{bmatrix}$$

Recall we can represent the adjugate matrix as cross products:

$$\vec{p} = \frac{D_1(\hat{n}_2 \times \hat{n}_3) + D_2(\hat{n}_3 \times \hat{n}_1) + D_3(\hat{n}_1 \times \hat{n}_2)}{det\mathbb{A}}$$

where $\hat{n}_i$ are normal vectors of three planes.
The determinant can also be expressed using a dot product and a cross product as follows:

$$det\mathbb{A} = \hat{n}_1 \cdot (\hat{n}_2 \times \hat{n}_3)$$

The final form of the formula is:

$$\vec{p} = \frac{D_1(\hat{n}_2 \times \hat{n}_3) + D_2(\hat{n}_3 \times \hat{n}_1) + D_3(\hat{n}_1 \times \hat{n}_2)}{\hat{n}_1 \cdot (\hat{n}_2 \times \hat{n}_3)}$$

The procedure of finding the intersection point expressed in pseudo-code:

```
function intersectionPoint(pl1 : struct plane, pl2 : struct plane,
    pl3 : struct plane) : vec3 {
    let det = vec3::dotProduct(pl1.normal,
        vec3::crossProduct(pl2.normal, pl3.normal));

    //if determinant is close to 0 - planes do not intersect
    //in a single point
    if(abs(det) < EPSILON)
        return null;

    //(n2 x n3) * d1
    let cross : vec3 = vec3::crossProduct(pl2.normal, pl3.normal);
    let point : vec3 = vec3::scale(cross, pl1.distance);

    //(n3 x n1) * d2
    cross = vec3::scale(vec3::crossProduct(pl3.normal, pl1.normal),
        pl2.distance);
    point = vec3::add(point, cross);

    //(n1 x n2) * d3
    cross = vec3::scale(vec3::crossProduct(pl1.normal, pl2.normal),
    pl3.distance);
    point = vec3::add(point, cross);

    //scale the final coordinates by inverse of determinant
    return vec3::scale(point, 1.0 / det);
}
```

9.3 Removing invalid points

As stated in the *Overview* - invalid points may be generated, therefore each point needs to be checked against all planes of a brush. If the point is outside of any plane - it is discarded.

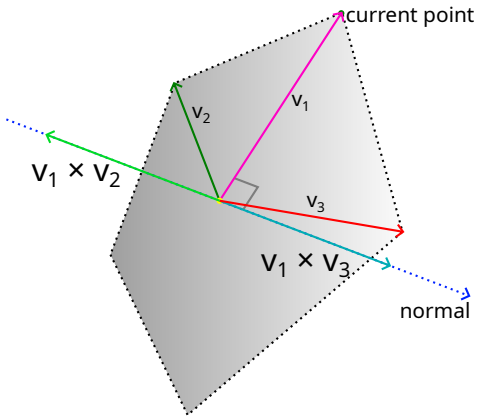
The procedure in pseudo-code:

```
function isPointValid(point : vec3, sides : array) : bool {
    for(int i = 0; i < sides.length; ++i){
        let pl : struct plane = sides[i].plane;
```

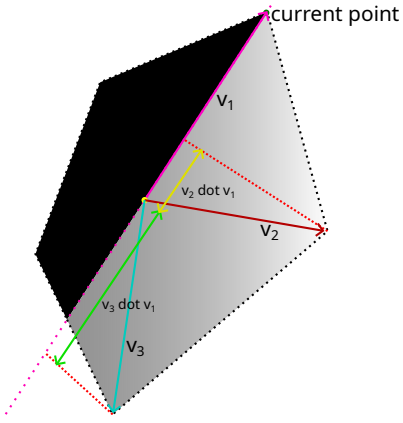
```
            if(vec3::dotProduct(point, pl.normal) - pl.distance < -EPSILON)
                return false;
        }

        return true;
    }
```

9.4 Sorting the points

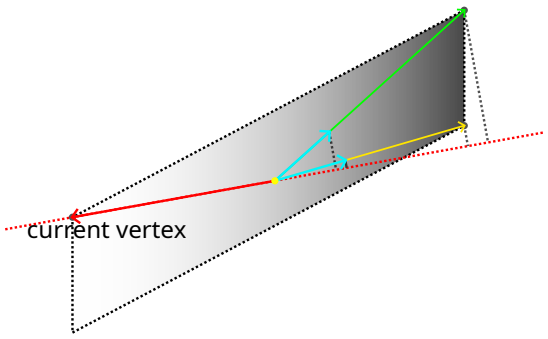
Generated points are in arbitrary order - to transform them into a winding the points need to be sorted clockwise. Vertices of a convex polygon can be sorted easily - observe the center point, being an average of all the vertex coordinates, is inside the shape. For every vertex, a next vertex in order can be found by first obtaining a vector from the center to the current vertex, let's call that vector $\vec{v}_1$. This vector is then compared with $\vec{v}_2$, being a vector from the center to the other vertex, using a cross product. Recall, the direction of a cross product depends on the relative order of the vectors - if a second vector is clockwise to the first one, their cross product will point "down" (in the direction of plane normal). We can use that to discard vertices that are anticlockwise to the current vertex by calculating the cross product $\vec{v}_{\text{cross}}$ of $\vec{v}_1$ and $\vec{v}_2$ and then taking the dot product of $\vec{v}_{\text{cross}}$ and the normal of a plane $\hat{n}$. If the resulting dot product is negative - a vertex is anticlockwise relative to the current vertex and should no longer be considered:



Having a way to consider only the points clockwise to the current vertex, now a vertex closest to the current one needs to be found. This will be done using the dot product of $\vec{v}_1$ and $\vec{v}_2$:



A problem occurs in such cases:



As it is visible on the illustration, based on the dot product, a wrong vertex will be chosen as the next one. That is because the proper next vertex is further away from the center and the dot product is proportional to the length of the vectors. To remedy this issue the $\vec{v}_2$ vector needs to be normalized.

Putting it all together in pseudo-code:

```
function sortWinding(points : array, normal : vec3) : array {
    //sorts the points array in place
    //this function modifies the original points array

    //calculate the coordinates of center point by averaging
    //the coordinates of all points
    let center : vec3 = points[0];
    for(let i = 1; i < points.length; ++i){
        center = vec3::add(center, points[i]);
    }
    center = vec3::scale(center, 1.0 / points.length);

    for(let i = 0; i < points.length - 1; ++i){
        let v1 : vec3 = vec3::subtract(points[i], center);

        let dotMax = -MAX_VALUE;
        let nextIndex = i + 1;

        for(let j = nextIndex; j < points.length; ++j){
            let v2 : vec3 = vec3::subtract(points[j], center);
            let cross : vec3 = vec3::crossProduct(v1, v2);

            //if the next point is anticlockwise - skip it
            if(vec3::dotProduct(cross, normal) < EPSILON)
                continue;

            //normalize the vec2!
            let dot = vec3::dotProduct(v1, vec3::normalize(v2));

            if(dot > dotMax){
                dotMax = dot;
                nextIndex = j;
            }
        }

        //swap the next point with the found one
        let temp : vec3 = points[i + 1];
        points[i + 1] = points[nextIndex];
        points[nextIndex] = temp;
    }

    //the points array is sorted now!
    return points;
}
```

9.5 Optimizations

Finding an intersection point between three planes is an $O(n^4)$ algorithm, because for every plane (n) we need to check every unique pair of planes (n^2). Once the point is generated it needs to be

determined whether it is valid or not and, as discussed earlier, it involves testing it against all planes of a brush (n). Of course the $O(n^4)$ complexity is just a worst-case estimation, anyhow, the algorithm would benefit greatly from a way of optimizing it.

One method is to cache the intersection of three planes using data structures like the Hash Map and a Hash Set. The Hash Map would store information about the point of intersection, and whether the point is valid or not, for every index of intersection, meaning a tuple consisting of three indices of planes which formed the intersection - (i, j, k). Because the planes are tested in an arbitrary order, we might get intersections of (i, j, k), (i, k, j), (k, i, j) etc. and they all evaluate to the same point even though the order is different. To get the index of intersection the three indices are sorted and combined into a single number using bitwise operations:

```
function indexOfIntersection(i, j, k) : number {
    let temp = i;
    if(i > j){ temp = i; i = j; j = temp;} //sort i, j, k
    if(j > k){ temp = j; j = k; k = temp;}
    if(i > j){ temp = i; i = j; j = temp;}

    //assume i, j, k < 256, thus 8 bits per index.
    return i | j << 8 | k << 16;
}
```

Such intersection index can be used as a key in Hash Map, therefore we can cache the intersections for an entire brush.

The intersection index is also used in conjunction with Hash Set to avoid duplicate points per side, which speeds up the later sorting process.

10 Putting it all together

The *generateMesh* functions take an array of sides of a single brush as an argument and return an array of triangulated vertices.

10.1 The Quake algorithm

```
function generateMesh(sides : array) : array{
    let triangles : array = [];

    for(let i = 0; i < sides.length; ++i){
        let curSide : struct side = sides[i];

        //if the current side has a NODRAW texture
        //or is not a displacement and there exist
        //other displacement side in brush
        if(!curSide.draw)
            continue;

        //generate the base winding
        let winding : array = baseWinding(curSide.plane, MAXMAPSIZE / 2);

        for(let j = 0; j < sides.length; ++j){
            let plane2 : struct plane = sides[j].plane;

            //if plane2 and current side plane are parallel - continue
            if(abs(vec3::dot(curSide.plane.normal, plane2.normal)) > 1-EPSILON)
                continue;

            winding = clipWinding(winding, plane2);
        }

        //something went wrong! less than 3 vertices!!!
        if(winding.length < 3)
            continue;

        //round the vertices
        for(let j = 0; j < winding.length; ++j){
            winding[i] = vec3::round(winding[i]);
```

```
        }

        if(curSide.dispinfo){
            let dispTris : array = createDisplacementMesh(
                winding, curSide.dispinfo);

            //add the triangles from dispTris to triangles
            //check if dispTris is null
            if(dispTris)
                triangles.concat(dispTris);
        } else {
            let tris : array = triangulate(winding);
            triangles.concat(tris);
        }
    }

    return triangles;
}
```

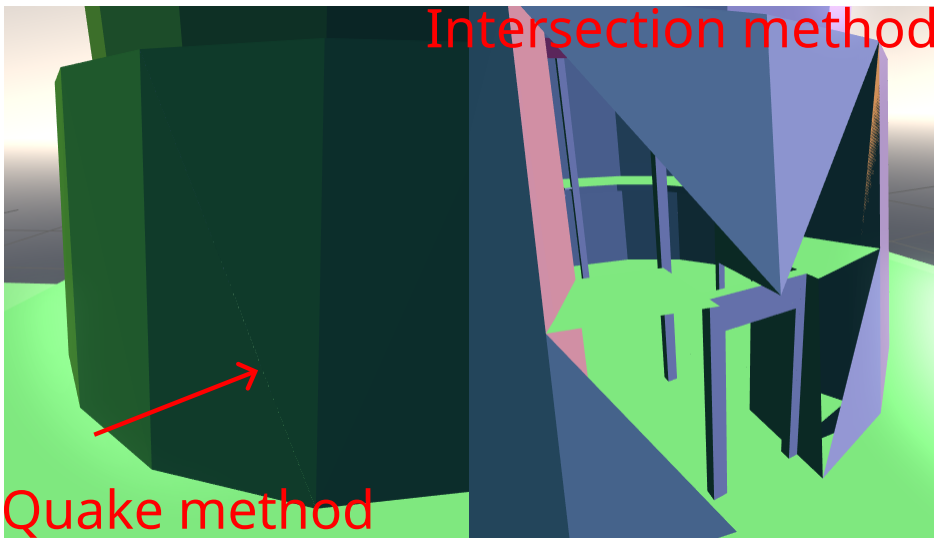
10.2 The Intersection algorithm

```
struct intinfo{
    point : vec3,
    valid : bool
}

function generateMesh(sides : array) : array{
    let triangles : array = [];
    let intMap : hashmap = hashmap();

    for(let i = 0; i < sides.length; ++i){
        let curSide : struct side = sides[i];

        if(!curSide.draw)
            continue;

        let intSet : hashset = hashset();
        let winding : array = [];
        let pl1 : struct plane = curSide.plane;

        for(let j = 0; j < sides.length; ++j){
            let pl2 : struct plane = sides[j].plane;

            //if pl1 and pl2 are parallel - continue
            if(abs(vec3::dot(pl1.normal, pl2.normal)) > 1-EPSILON)
                continue;

            for(let k = 0; k < sides.length; ++k){
                let pl3 : struct plane = sides[k].plane;

                //if pl3 is parallel either to p1 or p2 - continue
                if(abs(vec3::dot(pl1.normal, pl3.normal) > 1-EPSILON)
                    continue;
                if(abs(vec3::dot(pl2.normal, pl3.normal) > 1-EPSILON)
                    continue;

                let intIndex = indexOfIntersection(i, j, k);

                //look up if the current intersection point was already
                //calculated for the side i
                if(intSet.contains(intIndex))
                    continue;

                intSet.add(intIndex);

                let p : vec3;
                let valid : bool;

                //look up if the current intersection point was already
                //calculated for the brush
                let intInfo : struct intinfo = intMap.get(intIndex);
```

```
                //not calculated, calculate it and add to hashmap
                if(!intInfo){
                    p = intersectionPoint(pl1, pl2, pl3);

                    if(!p)
                        continue;
                    valid = isPointValid(p, sides);

                    p = vec3::round(p);

                    intMap.set(intIndex, struct intinfo {point: p, valid: valid});
                } else if(intInfo.valid){
                    winding.push(intInfo.point);
                }
            }
        }

        if(winding.length < 3)
            continue;

        //winding ready, now needs to be sorted
        sortWinding(winding, pl1.normal);

        if(curSide.dispinfo){
            let dispTris : array = createDisplacementMesh(
                winding, curSide.dispinfo);

            if(dispTris)
                triangles.concat(dispTris);
        } else {
            let tris : array = triangulate(winding);
            triangles.concat(tris);
        }
    }
}
```

11 Summary

The Quake method of obtaining a mesh for a brush is faster, and in my opinion much more intuitive, than intersection method. During the tests I have observed the intersection method is much less numerically precise, thus requiring a much higher epsilon value. In my code the *glMatrix* library is used for vector calculations, therefore the calculations are based on single precision floating point numbers. I assume using double precision floating point numbers might have given a much better result - even with bigger epsilon the intersection method still occasionally produce incorrect geometry. In case of Quake method the inaccuracies are subtle.



The final points of a winding, after the clipping or intersection calculations are done, can be safely converted into single precision floating point numbers to conserve memory.

12 Links

- VMF WebGL Preview demo: https://jakgor471.github.io/vmf-files_webgl/
- Source code: https://github.com/jakgor471/vmf-files_webgl

13 Bibliography

- ".MAP files, file format description, algorithms, and code", Stefan Hajnoczi (https://github.com/stefanha/map-files)
- Quake 2 Tools, ID Software (https://github.com/id-Software/Quake-2-Tools/)
- VMF format specification, Valve Developer Wiki (https://developer.valvesoftware.com/wiki/VMF_(Valve_Map_Format))
- "OpenGL Game Programming", Kevin Hawkins, Dave Astle