Development Chaos Theory
An on-going conversation about stuff I find interesting

3D Clipping in Homogeneous Coordinates.

Posted on **May 22, 2016**

Recently I got an Arduboy, and I thought it'd be an interesting exercise to blow the dust off my computer graphics knowledge and put a simple 3D vector drawing system together. After all, while by today's standards a 16 MHz doesn't sound like a lot, back when I was learning Computer Graphics at Caltech in the 1980's in Jim Blinn's computer graphics class, that was quite a bit of computational power.

A limiting factor on the Arduboy is available RAM; after you factor in the off-screen bitmap for drawing, you only have perhaps 1250 bytes left–not a lot for a polygonal renderer. But when a 4×4 transformation matrix fits into 64 bytes, that's plenty of space for a vector graphics pipeline.

First step in building a vector graphics pipeline, of course, is building a 3D line clipping system; you want to be able to clip lines not just to the left and right, top and bottom, but also to the near and far clipping planes. And you do want to clip before you draw the lines; on the Arduboy, line drawing coordinates are represented as 16-bit signed integers, and with a 3D rendering system it would be very easy to blow those limits.

Sadly, however, I was unable to find a concise description of the homogeneous coordinate system clipping algorithm that I used when I was at Caltech. It'd been 30 years, and I had a vague memory of how we did it–how we combined the Cohen-Sutherland algorithm with Clipping using homogeneous coordinates and some general observations made during class to build an efficient 3D line clipper. And an Internet search yielded very little.

So I thought I'd write an article here, in case in another 20 years I need to dip back into the archives to build yet another line clipping system.

If you want to skip the math and get to the algorithm, scroll down to the section marked "Algorithm."

Some preliminaries, before we get started.

In Computer Graphics, it is advantageous to represent 3D points using homogeneous coordinates. That is, for any point in 3D space (x,y,z), you add an additional term w, giving (x,y,z,w).

To convert from a homogeneous coordinate (x,y,z,w), you use the following relationship:

Typically when we convert a point from 3D space to a homogeneous coordinates we simply tack a 1 at the end. There are special cases where you may want to do something simple (such as representing stars plotted infinitely far away, but typically you simply tack a 1 on:

The purpose of this extra parameter is to simplify coordinate transformation through transformation matrices. This extra 1 term, for example, allows us to represent translation (that is, moving a point from one location to another) as a simple matrix multiplication. So if we are moving the points in our system over by (5,3,2), we can create a 4×4 translation matrix:

and a point (x,y,z) would be translated through multiplication:

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 3 & 2 & 1 \end{bmatrix} \rightarrow \begin{pmatrix} x+5 & y+3 & z+2 & 1 \end{pmatrix}$$

One nice thing about using homogeneous coordinates is that we can represent perspective through a matrix multiplication. I'll save you the diagrams you see on other countless web sites by noting that to represent point perspective, you divide the location (x,y) by the distance z; objects far away appear smaller and closer objects appear larger.

If we use a perspective matrix which adjusts the size (x,y) to fit our screen (so objects to the far left are at x = -1, objects to the far top are at y = -1, right and bottom are at +1 respectively), then a point at (x,y,z) would map in our perspective transform:

(Side note: we use a right hand rule for coordinates. This means if up is +y and right is +x, then distance is -z. Which brings me to a rule of thumb with computer graphics: if the image looks wrong, look for an incorrect minus sign. While writing this article and testing the code below, I encountered at least 4 places where I had the wrong minus sign somewhere.)

Now if we convert our coordinates using the homogeneous coordinate to 3D point transformation above:

$$(ax\ by\ -1\ -nz) \rightarrow \left(\frac{ax}{-nz} \quad \frac{by}{-nz} \quad \frac{-1}{-nz} \right)$$

That is, our 3D point (in what we call “screen space”) has the point perspective transformation baked in, and points in 3D space at (x,y,z) show up with perspective on our 2D screen: we just drop the third term and plot points on our screen at the location ($^{ax}/_{-nz}$,$^{by}/_{-nz}$).

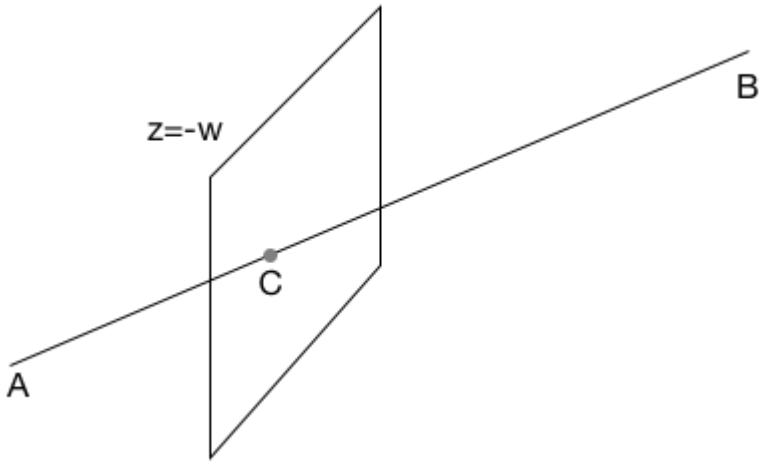
Notice that we’ve set our boundaries for screen space with the screen space rectangle (-1,-1)-(1,1). It is advantageous for us to clip our drawings at those boundaries, and (for reasons I’ll skip here), it helps to also clip Z coordinates at a near clipping plane (generally z=-1), and a far clipping plane (z=-∞).

The rest of this article will describe in concrete steps how to do this clipping in homogeneous coordinates, using the two articles cited at the start of this article.

First, let us note that to clip in the screen space x/w >= -1, in homogeneous coordinates we really are clipping at x >= -w.

Second, let us define what we mean by clipping. When we say we are clipping a line, what we’re doing is finding the point where a line intersects our clipping plane x >= -w. We then shorten the line at that new point, drawing only the segment that is visible.

So, given two points A and B, we want to find the point C where C intersects the plane x = -w, or rather, where x + w = 0:



We can do this by observing that the dot product of a point and a vector is the distance that point is along the vector, and that our clipping plane x + w = 0 can also be defined as the set of points P where

$$P\ \langle 1\ 0\ 0\ 1\rangle \rightarrow 0$$

So to find our point C, we calculate the distance along the vector (1,0,0,1) (that is, x = -w) for the points A and B, and linearly interpolate to find the location C.

$$A\ \langle 1\ 0\ 0\ 1\rangle \rightarrow A_\alpha$$
$$B\ \langle 1\ 0\ 0\ 1\rangle \rightarrow B_\alpha$$
$$\frac{A_\alpha}{A_\alpha - B_\alpha} \rightarrow C_\alpha$$
$$A(1-C_\alpha) + B(C_\alpha) \rightarrow C$$

There are some other useful observations we can make about the set of relationships above.

First, if x is greater than -w — that is, if we are on the inside of the clipping plane x/w >= -1, then the dot product A(1,0,0,1) will be greater than or equal to 0.

Second, if both A(1,0,0,1) and B(1,0,0,1) are both negative, then trivially our line is not visible, because both endpoints of the line are not visible.

And we can repeat this operation for all the relationships: for x >= -w, x <= w, y >= -w, y <= w, and z >= -w.

The far clipping plane can also be taken care of, by observing that our far clipping plane at infinity is reached as z/w approaches infinity; that is, as w approaches zero. So our far clipping plane would be the relationship z <= 0.

This gives us the following normal vectors representing our 6 clipping planes:

$$X_{left} = \langle 1\ 0\ 0\ 1 \rangle$$
$$X_{right} = \langle -1\ 0\ 0\ 1 \rangle$$
$$Y_{top} = \langle 0\ 1\ 0\ 1 \rangle$$
$$Y_{bottom} = \langle 0\ -1\ 0\ 1 \rangle$$
$$Z_{near} = \langle 0\ 0\ 1\ 1 \rangle$$
$$Z_{far} = \langle 0\ 0\ -1\ 0 \rangle$$

Note that the vectors are set up so that when we are outside of the clipping plane, the value of the dot product is negative: that is, if the relationship x <= w holds, then 0 < w – x, and our clipping plane for that relationship is (-1 0 0 1).

The algorithm

This allows us to start sketching our solution.

Note our definition of a 3D vector is a homogeneous vector:

```
struct Vector3D
{
    double x,y,z,w;
}
```

First, we need a method to calculate all our clipping dot products. Note that because we've transformed our process into clipping against a set of vectors, the routine below could be generalized to add additional clipping planes. (For example, you may want to create a renderer that only renders the stuff in front of a wall in your game.)

```
static double Dot(int clipPlane, const Vector3D &v)
{
    switch (clipPlane) {
        case 0:
            return v.x + v.w;        /* v * (1 0 0 1) */
        case 1:
            return - v.x + v.w;      /* v * (-1 0 0 1) */
        case 2:
            return v.y + v.w;        /* v * (0 1 0 1) */
        case 3:
            return - v.y + v.w;      /* v * (0 -1 0 1) */
        case 4:
            return v.z + v.w;        /* v * (0 0 1 1) */
        case 5:
```

```
            return - v.z;            /* v * (0 0 -1 0) */
        }
        return 0; /* Huh? */
    }
```

Now we borrow from the Cohen-Sutherland algorithm, and create a method which generates an "outcode"; this is a bitmap with 6 bits, each bit is set if a point is "outside" our clipping plane–that is, if the dot product for the given coordinate is negative.

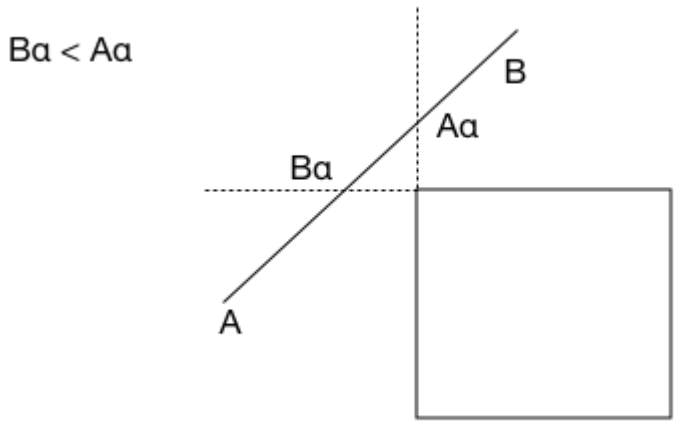
```
static uint8_t OutCode(const Vector3D &v)
{
    uint8_t outcode = 0;
    for (int i = 0; i < 6; ++i) {
        if (Dot(i,v) < 0) outcode |= (1 << i);
    }
    return outcode;
}
```

(Note that we could easily optimize this method by using direct compares rather than by calling into our Dot method. We can also generalize it if the dot product method above is generalized.)

A nice feature about our OutCodes is that if we have the outcodes for points A and B, the following relationships are true:

- **if ((a_outcode & b_outcode) != 0)** then both points are on the wrong side of at least one edge, and we can trivially reject this line.
- **if ((a_outcode | b_outcode) == 0)** then both points are inside our viewing area, and we can trivially accept this line.

Now if it turns out both relationships are true, then we must clip our line A – B. And we can do that by calculating the alpha coordinates along A and B–if A is outside, we track the alpha on the A side of the line, and if B is outside, we track an alpha along B. And if the two flip–then we have a case like the one shown below:



This allows us to set up our final clipping algorithm.

First, we track the last point that was added, and the out code for that point:

```
static Vector3D oldPos;      // The last point we moved or draw to
static uint8_t oldOutCode;   // The last outcode for the old point.
```

We also rely on a drawing routine which actually draws in screen space. This should take the 3D homogeneous point provided, transform (using the transformation to 3D space noted above) to a 3D point, drop the Z component, convert the (x,y) components from the rectangle (-1,-1) – (1,1) to screen pixels, and draw the line as appropriate:

```
void MoveDrawScreen(bool draw, const Vector3D &v);
```

One last method we need is a utility method that, given a start point and an end point, and an alpha, calculates the linear interpolation between the start and end points.

```
static void Lerp(const Vector3D &a, const Vector3D &b, float alpha, Vect
{
    float a1 = 1.0f - alpha;
    out.x = a1 * a.x + alpha * b.x;
    out.y = a1 * a.y + alpha * b.y;
    out.z = a1 * a.z + alpha * b.z;
    out.w = a1 * a.w + alpha * b.w;
}
```

Now our method declaration takes a new point and a flag that is true if we're drawing a line, or false if we are moving to the point:

```
void MoveDrawClipping(bool draw, const Vector3D &v)
{
    Vector3D lerp;
    float aold;
    float anew;
    float alpha;
    float olda,newa;
    uint8_t m;
    uint8_t i;
    uint8_t newOutCode;
    uint8_t mask;

    /*
     *    Calculate the new outcode
     */
```

```
newOutCode = OutCode(v);
```

In the above we calculate the outcode for our new point *v*. This allows us to quickly determine if the point is inside or outside our visible area. This is useful when we want to just move to a point rather than draw to a line:

```
/*
 *  If we are not drawing, and the point we're moving to is visible,
 *  pass the location upwards.
 */

if (!draw) {
    if (newOutCode == 0) {
        MoveDrawScreen(draw,v);
    }
} else {
```

If *draw* is true, then we're drawing a line, and so we get to the meat of our clipping algorithm. Note that our old out code and old points (stored in the globals above) have already been initialized. (Or rather, we assume they have been with an initial move call.)

So we can borrow from the Cohen-Sutherland algorithm and quickly determine if our line is entirely clipped or entirely visible:

```
    /*
     *  Fast accept/reject, from the Cohen-Sutherland algorithm.
     */

    if (0 == (newOutCode & oldOutCode)) {
        /*
         *  The AND product is zero, which means the line may be
         *  visible. Calculate the OR product for fast accept.
         *  We also use the mask to quickly ignore those clipping
         *  planes which do not intersect our line
         */

        mask = newOutCode | oldOutCode;
        if (0 == mask) {
            /*
             *  Fast accept. Just draw the line. Note that our
             *  code is set up so that the previous visible line has
             *  already been moved to
             */

            MoveDrawScreen(true,v);
        } else {
```

If we get to the code below, this means we have a value for *mask* which indicates the clipping planes that we intersect with. So now we want to interate through all the clipping planes, calculating the dot product for only those planes we intersect with:

```
/*
 *  At this point mask contains a 1 bit for every clippi
 *  plane we're clipping against. Calculate C alpha to f
 *  C, and adjust the endpoints A alpha (aold) and B alp
 *  (anew)
 */

aold = 0;            /* (1 - alpha) * old + alpha * new
anew = 0;            /* so alpha = 0 == old, alpha = 1 =
m = 1;
for (i = 0; i < 6; ++i) {
    if (mask & m) {

        /* Calculate alpha; the intersection along the l
        /* vector intersecting the specified edge */

        olda = Dot(i,oldPos);
        newa = Dot(i,v);
        alpha = olda/(olda-newa);
```

And next we need to bring in the side of the line which we know has clipped against the clipping plane. Meaning if A is outside our visible area and B is inside, we want to adjust A_α to the new value calculated in *alpha*; but if A is inside and B outside, we want to adjust B_α instead.

```
        /* Now adjust the aold/anew endpoints according
        /* which side (old or v) is outside. */
        if (oldOutCode & m) {
            if (aold < alpha) aold = alpha;
        } else {
            if (anew > alpha) anew = alpha;
        }
```

And finally if somehow the edges cross, we have the circumstance above where the line is not visible but crosses two clipping planes. So we quickly reject that case:

```
        if (aold > anew) {
            /*
             *  Our line is not visible, so reject.
             */
```

```
                    break;
                }
```

The rest simply closes our loop:

```
            }
            m <<= 1;
        }
```

At this point if we have iterated through all six planes, we have a line which is visible. If our starting point in *oldPos* was not visible, then we need to move to the place where our starting point intersects with the clipping boundaries of our view space:

```
        if (i >= 6) {
            /* Ran all clipping edges. */
            if (oldOutCode) {
                /* Old line coming into view */
                Lerp(oldPos,v,aold,lerp);
                MoveDrawScreen(false,lerp);
            }
```

And if our destination point is visible, we draw to it; otherwise, we draw to the point on the line we're drawing to which intersects our clipping boundaries:

```
            /* Draw to the new point */
            if (newOutCode) {
                Lerp(oldPos,v,anew,lerp);
                mdl2(true,lerp);
            } else {
                mdl2(true,v);
            }
```

Our clipping is done; all that is left is to close up the loops and conditionals, and store away in our globals the outcode for the point we just moved to, as well as the location of that point:

```
            }
        }
    }
}

/*
 *    Now update the old point to what we just moved to
 */

oldOutCode = newOutCode;
```

```
        oldPos = v;
    }
```

There are a number of optimizations that can take place with the clipping code above. For example, we can also track an array of dot products so we're not repeatedly calculating the same dot product over and over again. We could replace our outcode routine with simple compare operations. We could also extend the above routine to handle additional clipping planes.

But this gives us a vector line drawing clipping system that clips in homogeneous coordinates, complete with source code that can be compiled and executed.

What is old is new again; what started as a powerful desktop computer has now appeared as an embedded processor. So it's good to remember the old techniques as we see more and more microcontrollers show up in the products we use.

This entry was posted in **Algorithms**, **Things To Remember** by **William Woody**. Bookmark the **permalink [https://chaosinmotion.blog/2016/05/22/3d-clipping-in-homogeneous-coordinates/]** .