Rotation and backface culling for simple demo-like objects

For simplicity of illustrations this document will describe the 2D case, not the 3D case. However my goal is to give you the needed insight to picture the problem and visualize the solution. This will make it trivial to extend into the 3D domain.

The rotation matrix

First of all I'd like you to really see what's going on in the rotation matrix and not just treat it mathematically, but instead really picture it.

The general rotation matrix in 2D is:

$$\begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix}$$

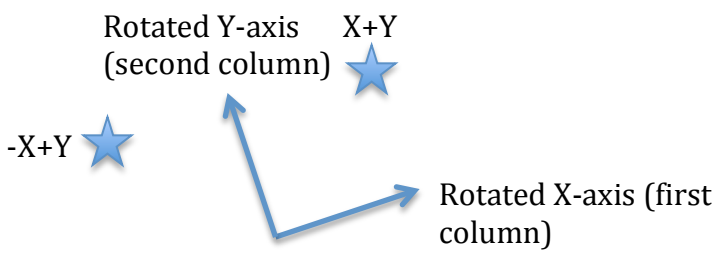
What this matrix tells you are the coordinates of the rotated world axes expressed in the current coordinate system before rotation. Each column express the coordinates of an axis, first the rotated X-axis then the rotated Y-axis. To clarify, imagine the normal X-axis [1,0]. When rotating (always counter-clockwise), the X-axis will start in its normal [1,0] position and after 90 degrees reach [0,1] then continue to [-1,0] after 180 degress and finally after 270 degrees reach [0,-1]. Now, take a look at the x-value in this progression; first 1, then 0, then -1 and finally 0. Sounds a lot like cos(a) to me… Now take a look at the y-value of the X-axis: first 0, then 1, then 0 and finally -1, looks like sin(a).

Let's do the same with the Y-axis. It first starts in [0,1], then rotates to [-1,0], then to [0, -1] and finally [1,0]. The x-component of the Y-axis goes from 0, -1, 0, 1 => -sin(a). The y-component of the Y-axis goes from 1, 0, -1, 0 => cos(a).

So, the coordinates of the X-axis are [cos(a), sin(a)] and the coordinates of the Y-axis are [-sin(a), cos(a)]. As you can see we've now quite simply deduced each row of the rotation matrix above. The same simple principles hold for the 3D case. but there you'll have a bunch of 0 and 1 as well, which is quite natural. Imagine rotating the Z-axis [0,0,1] around Z, it will stay static, hence that column in the Z-rotation-matrix will stay [0,0,1] etc.

Quick object generation

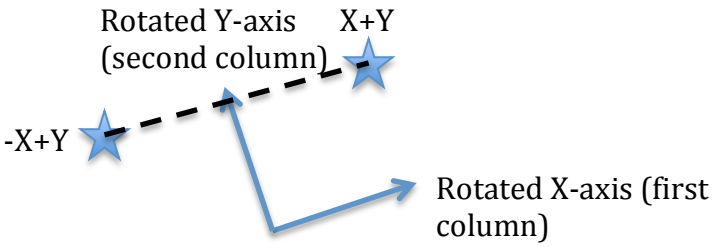
Now that we have the rotation matrix and that we really can picture each row of the matrix as coordinates of one of the world vectors we can add those vectors in various ways to reach different points in rotated space:



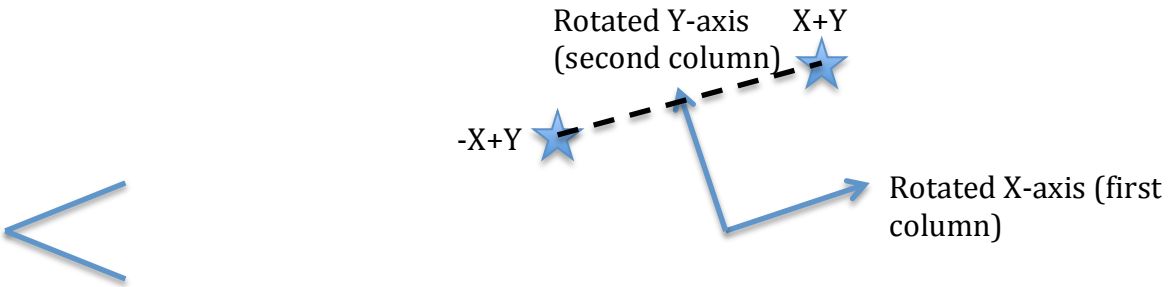
Backface culling

How to determine if a side (or a face in 3D) is visible or not? One method is to figure out if the eye or camera is in front of a face or side or not. If the camera is on the backside of a face then that face must be invisible, or culled.

Let's first define a side between the two calculated points:



We must also define the location of the eye, but let's keep the object in origo and move the camera a bit away from the object instead:



Camera at a fixed [-cx, 0] position. (in 3D it's fixed at [0,0,-cz]

To determine which side of a line or face a point is we have to resort to the equation of a line in general form: $a*x + b*y + c = 0$

[a,b] is the normal to the line. [x,y] is a point on the line. c is the closes distance from the line to origo in terms of normal lengths. If the formula equals 0 [x,y,z] lies on the line. If it's >0 then the point lies on the side of the line the normal is pointing at.

So, what are [a,b] and c for our line above. Well, the normal is simple, it's just the rotate Y-axis! And c is also simple, the line lies 1 normal from origo, i.e. 1.

The equation for our line above would then be:

$Y_x*x + Y_y*y + 1 = 0$

Remember, if the point [x,y] lies on the normal side of the line the formula would be >0.

The point to check is actually our camera which lies in [-cx, 0] which gives:

Y_x*-cx + Y_y*0 + 1 > 0 ⇔

1 - Y_x*cx > 0 ⇔

Y_x < 1/cx

So, since 1/cx is constant it's sufficient to simply compare Y_x to 1/cx to check if the line is visible or not!

So the general idea is to calculate c/cx for each side (in 3d, d/cz) and store that. Then also store how to calculate the normal for each side/face in terms of world axis additions, just like the vertices.

Then when displaying the object just add the world axes in various ways to get the vertices AND normals and then compare each face-normal to each face-constant to determine visibility.

And this is exactly how it was done in the hidden line vector part in the 1991 demo by Booze Design. In that code I created a simple byte code that described each object. The byte code would after interpretation generate a chunk of unrolled code that calculated all the vertices and normal.

Also remember that you don't have to do full additions for the normal since you only need the z-component!

/JackAsser