1 Introduction

Given the prevalence of three-dimensional polygonal graphics in modern gaming and computer graphics, a technique for rotating two-dimensional bitmaps may seem too primitive or basic to bother learning. Nevertheless, studying such a technique has its merits. Two-dimensional graphics still thrive on handheld game systems, where simple rotations are frequently employed to produce a variety of visual effects (see, for example, any handheld Castlevania game). It can also serve as an intermediate step before moving on to learning more advanced graphics techniques. After all, rotating a bitmap is equivalent to rotating a polygon, which is a basic transformation in 3D graphics engines.

Though there is rarely a reason to manually implement rotations thanks to most modern graphics hardware providing functions to do so, understanding what is actually going on when using these functions can help design engines that exploit the features more efficiently. Below, a well-known algorithm for rotating a bitmap is explained, along with some of the requisite mathematics for understanding how it works.

2 Mathematical Background

There are many ways of representing rotations mathematically, but the one most relevant to rotating a bitmap is the linear transform

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

where x and y are the coordinates of the point to be rotated and θ is the angle of rotation around the origin.

One way of justifying this formula is by relating it to multiplication of complex numbers. Using polar representation, we can write two complex numbers as $z_1 = r_1 e^{i\theta_1}$ and $z_2 = r_2 e^{i\theta_2}$, where r is the magnitude of the vector formed from by the real and imaginary components of z in the complex plane, θ is the angle of the same vector with respect to the real axis, and the subscripts indicate which number the values are associated with. The formula for multiplying two complex numbers in this form is

$$z_1 z_2 = r_1 r_2 e^{\theta_1 + \theta_2}$$

We can see here that multiplication increases the angle of z_1 by the angle of z_2 (or vice-versa) and scales the magnitude of z_1 by the magnitude of z_2 (or vice-versa). If we restrict z_2 to the set of complex numbers with magnitude 1 (i.e., complex numbers on the unit circle), then the formula becomes

$$z_1 z_2 = r_1 e^{\theta_1 + \theta_2}$$

Here only the angle of z_1 changes, while the magnitude stays the same. Since rotation of a vector around the origin is the same as keeping the magnitude constant while rotating around the origin, we can say multiplication of a complex number by another complex number with magnitude 1 and angle θ is equivalent to rotation around the origin by the angle θ.

So what does this have to do with the matrix formula above? First we find the coordinates of the point, denoted (x, y), after it has been rotated by multiplying the vector by the matrix, denoted (x', y'). This gives

$$x' = x\cos\theta - \sin\theta$$

and

$$y' = x\sin\theta + y\cos\theta$$

Now we rewrite the above complex multiplication. Convert z_1 to Cartesian form using Euler's formula, we get

$$z_1 = r_1\cos\theta_1 + ir_1\sin\theta_1$$

and we let x equal the real part and y equal the imaginary part, giving

$$x = r_1\cos\theta_1$$

$$y = r_1\sin\theta_1$$

This represents our starting point, before rotation, as we can set the x and y coordinates to any point we want and they will match the vector in the above matrix multiplcation. Next we convert z_2, the unity magnitude complex number to Cartesian form in the same way

$$z_2 = \cos\theta_2 + i\sin\theta_2$$

Multiplying z_1 and z_2 gives

$$\begin{aligned} z_1 z_2 &= (x + iy)(\cos\theta_2 + i\sin\theta_2) \\ &= x\cos\theta_2 - y\sin\theta_2 + i(x\sin\theta_2 + y\cos\theta_2) \end{aligned}$$

Equating the real part of this result with x' and the imaginary part with y' gives

$$x' = x\cos\theta_2 - \sin\theta_2$$

$$y' = x\sin\theta_2 + y\cos\theta_2$$

Comparing this to the result of the matrix multiplication, we can see these are the same.

You can also get a more intuitive geometric feel for why the formula is justified by drawing diagrams showing the projections onto the x and y axes of the standard basis vectors $(1, 0)$ and $(0, 1)$. Understanding why those work will give an idea of why the formula in general works, since it is a linear transform.

3 Implementation

With the above background, an implementation can now be developed. By treating each pixel in a bitmap as a point in the x-y plane, the matrix above can be almost directly applied. There are a couple of important practical factors to consider, however. First, the conventional notation for the layout of pixels on a

screen has the y-axis increasing in the downward direction, which is the opposite of conventional mathematic notation for the Cartesian plane. This has the effect of reversing the direction of the rotation - i.e., a positive angle for rotation rotates in the clockwise direction, rather than the counter-clockwise rotation. Second, computing the rotation directly on the source bitmap will leave gaps in the final rotation for some angles due to quantization errors resulting from the representation of points in the Cartesian plane as discrete pixels on the computer. A simple way to avoid this problem is to compute the rotation in the other direction. The following steps explain the method.

1) Rotate the four points at the corners of the source bitmap to determine the minimum and maximum points of the rotated bitmap to be computed.

2) Using these dimensions, iterate line-by-line and pixel-by-pixel, computing the rotation in the opposite direction (basically, rotating from the destination bitmap back to the original). If the computed pixel is within the dimensions of the original bitmap, copy its color to the corresponding pixel in the rotated bitmap. Otherwise, copy black, transparency, or whatever background color is in use to the current pixel in the rotated bitmap.

That's about it for a basic bitmap rotation. This is not the only way to do it, as features like anti-aliasing are not implemented, but just a basic method meant to demonstrate the concepts. Example source code implementing these ideas which uses SDL for graphics output is available at http://ensomnya.net/writings/001_bitmap_rotation/code.tar.gz