Integer-Based Rotations of Images

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/
Copyright © 1998-2008. All Rights Reserved.

Created: February 9, 2006
Last Modified: March 2, 2008

Contents

1 Rotation 2

1.1 Real-Valued Rotation . . . 2

1.2 Integer-Based Rotation . . . 4

1 Rotation

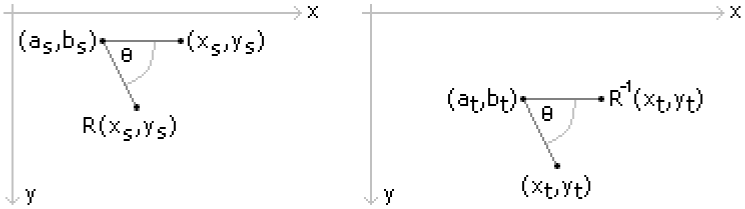
1.1 Real-Valued Rotation

Rotation of bitmaps by angles not a multiple of 90 degrees is a challenge to make fast and accurate when the algorithm must avoid floating point computations. The mathematics of rotation is relatively simple. We need a center point of rotation for the source, (a_s, b_s), an angle of rotation θ, and a center point for the target, (a_t, b_t). The source point (x_s, y_t) is transformed to the target point (x_t, y_t) by

$$\begin{aligned} x_t &= a_t + (x_s - a_s)\cos\theta + (y_s - b_s)\sin\theta \\ y_t &= a_t - (x_s - b_s)\sin\theta + (y_s - b_s)\cos\theta \end{aligned}$$

Figure 1.1 shows a typical rotation. A positive angle θ is associated with a counterclockwise rotation in the (left-handed) screen coordinates.

Figure 1.1 Left: The original and rotated points in the source block. Right: The rotated point in the target block.



The R in the left image denotes the rotation of (x_s, y_s) about the center (a_s, b_s) to obtain the point $R(x_s, y_s)$. In the right image, the point (x_t, y_t) is to be assigned the bitmap color that was associated with (x_s, y_s).

The natural inclination in implementing the rotation is to iterate over the source pixels, transform each pixel to a target location, and assign the source color to that target pixel. The target location is not necessarily a point with integer coordinates, so rounding must be performed to obtain the nearest pixel. As it turns out, this algorithm does not always work since not all target pixels will be "filled in". The correct implementation is to iterate over the target pixels, transforming each pixel to a source location, and calculating a color for the source location and assigning that color to the target pixel. In this way all target pixels are filled in. The right image of Figure 1.1 shows the "inverse rotation" of (x_t, y_t) to a point $R^{-1}(x_t, y_t)$. The inverse mapping is

$$\begin{aligned} x_s &= a_s + (x_t - a_t)\cos\theta - (y_t - b_t)\sin\theta \\ y_s &= b_s + (x_t - b_t)\sin\theta + (y_t - b_t)\cos\theta \end{aligned}$$

As indicated, x_s and y_s are in most cases not integer-valued.

The pseudocode for the nearest neighbor interpolation is

```
angle:  the angle of rotation
```

```
as, bs:  the source center of rotation
at, bt:  the target center of rotation
srcX1, srcX2, srcY1, srcY2: source rectangle
trgX1, trgX2, trgY1, trgY2: target rectangle
srcColor(x,y): return the source color at pixel (x,y)
trgColor(x,y): return the target color at pixel (x,y)

cs = cos(angle);  sn = sin(angle);
for (yt = trgY1; yt < trgY2; yt++)
{
    dy = yt - bt;
    for (xt = trgX1; xt < trgX2; xt++)
    {
        dx = xt - at;
        xs = as + dx * cs - dy * sn;
        ys = bs + dx * sn + dy * cs;
        rxs = (int)(xs + 0.5);
        rys = (int)(ys + 0.5);
        if (srcX1 <= rxs && rxs < srcX2 && srcY1 <= rys && rys < srcY2)
        {
            trgColor(xt, yt) = srcColor(rxs, rys);
        }
    }
}
```

The pseudocode for the bilinear interpolation is

```
angle:  the angle of rotation
as, bs:  the source center of rotation
at, bt:  the target center of rotation
srcX1, srcX2, srcY1, srcY2: source rectangle
trgX1, trgX2, trgY1, trgY2: target rectangle
srcColor(x,y): return the source color at pixel (x,y)
trgColor(x,y): return the target color at pixel (x,y)

cs = cos(angle);  sn = sin(angle);
for (yt = trgY1; yt < trgY2; yt++)
{
    dy = yt - bt;
    for (xt = trgX1; xt < trgX2; xt++)
    {
        dx = xt - at;
        xs = as + dx * cs - dy * sn;
        ys = bs + dx * sn + dy * cs;
        xb = (int)xs;
        yb = (int)ys;
        if (srcX1 <= xb && xb < srcX2 && srcY1 <= yb && yb < srcY2)
        {
            if (xb < srcX2-1)
            {
                deltaX = xs - xb;
                omdeltaX = 1.0 - deltaX;
            }
            else
            {
                xb = srcX2-2;
                deltaX = 1.0;
                omdeltaX = 0.0;
            }
            if (yb < srcY2-1)
            {
                deltaY = ys - yb;
                omdeltaY = 1.0 - deltaY;
            }
            else
            {
                yb = srcY2 - 2;
                deltaY = 1.0;
                omdeltaY = 0.0;
            }
            w00 = omdeltaX * omdeltaY;
            w01 = omdeltaX * deltaY;
            w10 = deltaX * omdeltaY;
            w11 = deltaX * deltaY;
            bilerpColor = (int)(
                w00 * srcColor(xb, yb) +
                w01 * srcColor(xb, yb + 1) +
                w10 * srcColor(xb + 1, yb) +
                w11 * srcColor(xb + 1, yb + 1)
                + 0.5);
        }
    }
}
```

The interpolation methods involve quite a bit of floating point calculations, including sine and cosine evaluation, the rotation transformation, the weight calculations, and the final weighted sum. We wish to avoid these for those platforms that must do floating point in software.

1.2 Integer-Based Rotation

The first issue we resolve is the evaluation of sine and cosine. We approximate sine and cosine for a set of angles by rational numbers. It is convenient to choose the same integer-valued denominator for the approximations, call it d. The integer-valued numerators are stored in an array for easy lookup. In our implementation we choose an array of 360 entries, the entry i corresponding to i degrees. Let $n_s(i)$ and $n_c(i)$ denote the numerators for the sine and cosine approximations, respectively. The approximation to sine of i

degrees is $n_s(i)/d$ and the approximation to cosine of i degrees is $n_c(i)/d$. The rotation of target to source is approximated by the equations

$$\begin{aligned} x_s &= a_s + (x_t - a_t)\tfrac{n_c(i)}{d} - (y_t - b_t)\tfrac{n_s(i)}{d} = \tfrac{da_s + (x_t - a_t)n_c(i) - (y_t - b_t)n_s(i)}{d} \\ y_s &= b_s + (x_t - a_t)\tfrac{n_s(i)}{d} + (y_t - b_t)\tfrac{n_c(i)}{d} = \tfrac{db_s + (x_t - a_t)n_s(i) + (y_t - b_t)n_c(i)}{d} \end{aligned}$$

As long as all coordinates for the points are integer-valued, the right-hand sides of these equations are rational numbers. The accuracy of the values depends on how accurate the rational numbers approximations are for sine and cosine.

We have now reduced the problem to integer calculations, but we do have two integer divisions, one per equation. Integer division can be as slow as floating point division on hardware, but typically is much faster than a software floating point division. But we can do even better. If the denominator d is chosen to be a power of 2, we can replace the division by right shifts. The important issue here, though, is making sure that the denominator is not chosen too large so that the numerators of the right-hand sides become large enough to cause 32-bit integer overflow. In our implementation, we chose $d = 1024$ so that the division is replaced by a right shift of 10, which is the log base 2 of 1024. This constant and the tables are exposed to the engine via this code placed in a header file:

```
const SInt32 kTrigTableSize = 360;
const SInt32 kTrigLogDenom = 10;  // denominator = 1024
extern const SInt32[] kTrigCosTable[];
extern const SInt32[] kTrigSinTable[];
```

The tables themselves are stored in a corresponding source file. They are generated by the code:

```
#include <cmath>
#include <fstream>
using namespace std;

void MakeTable (int size, int logDenom, const char* filename)
{
    ofstream ostr(filename);
    double denom = (double)(1 << logDenom);
    double twoPi = 8.0*atan(1.0), angle;
    int i, numer;

    // copy these lines from the output file to the header file
    ostr << "const SInt32 kTrigTableSize = " << size << ';' << endl;
    ostr << "const SInt32 kTrigLogDenom = " << logDenom << ';' << endl;
    ostr << "extern const SInt32 kTrigCosTable[];" << endl;
    ostr << "extern const SInt32 kTrigSinTable[];" << endl;

    // copy these lines from the output file to the source file
    ostr << "const SInt32 kTrigCosTable[kTrigTableSize] =" << endl;
    ostr << '{' << endl;
    for (i = 0; i < size; i++)
    {
        angle = twoPi*i/size;
        numer = (int)(denom*cos(angle)+0.5);
        ostr << '\t' << numer;
        if ( i+1 < size ) ostr << ',';
        ostr << endl;
    }
    ostr << "};" << endl << endl;
    ostr << "const SInt32 kTrigSinTable[kTrigTableSize] =" << endl;
    ostr << '{' << endl;
    for (i = 0; i < size; i++)
    {
        angle = twoPi*i/size;
        numer = (int)(denom*sin(angle)+0.5);
        ostr << '\t' << numer;
        if ( i+1 < size ) ostr << ',';
        ostr << endl;
    }
    ostr << "};" << endl;
    ostr.close();
}

int main ()
{
    const int size = 360;
    const int logDenom = 10;  // denominator = 2^{10} = 1024
    MakeTable(size,logDenom,"trigdata.txt");
    return 0;
}
```