Sines and Cosines for Animating Computer Graphics

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

You Know about Sines and Cosines from Math, but They are Very Useful for Animating Computer Graphics

First, We Need to Understand Something about Angles:

θ
X
Y

One of the things we notice is that each angle θ has a unique X and Y that goes with it. These are different for each θ.

Fortunately, centuries ago, people developed tables of those X and Y values as functions of θ. They called the X values cosines and the Y values sines. These are abbreviated cos and sin.

cos θ = X
sin θ = Y

How People used to Lookup Sines and Cosines – Yuch! Fortunately We Now Have Calculators and Computers

cos θ = X
sin θ = Y

If a circle has a radius of 1.0, then we can march around it by simply changing the angle that we call θ.

First, We Need to Understand Something about Angles:

R
θ
X
Y

If we were to double the radius of the circle, all of the X's and Y's would also double. So, really the cos and sin are ratios of X and Y to the circle Radius.
First, We Need to Understand Something about Angles

So, if we know the circle radius, and we march through a bunch of \( \theta \) angles, we can determine all of the X's and Y's that we need to draw a circle.

\[
\cos \theta = \frac{X}{R} \\
\sin \theta = \frac{Y}{R}
\]

Thus, We Could Create Our Very Own Circle-Drawing Function

```c
void Circle(float xc, float yc, float r, int numsegs)
{
    float dang = 2.f * F_PI / (float)numsegs;
    float ang = 0.;
    glBegin( GL_TRIANGLE_FAN );
    glVertex3f( xc, yc, 0. );
    for( int i = 0; i <= numsegs; i++ )
    {
        float x = xc + r * cosf(ang);
        float y = yc + r * sinf(ang);
        glVertex3f( x, y, 0. );
        ang += dang;
    }
    glEnd();
}
```

Circles and Pentagons and Octagons, Oh My!

```c
void Ellipse(float xc, float yc, float rx, float ry, int numsegs)
{
    float dang = 2.f * F_PI / (float)numsegs;
    float ang = 0.;
    glBegin( GL_TRIANGLE_FAN );
    glVertex3f( xc, yc, 0. );
    for( int i = 0; i <= numsegs; i++ )
    {
        float x = xc + rx * cosf(ang);
        float y = yc + ry * sinf(ang);
        glVertex3f( x, y, 0. );
        ang += dang;
    }
    glEnd();
}
```

And, there is no reason the X and Y radii need to be the same...
There is also no reason we can't gradually change the radius ...

```c
void Spiral(float xc, float yc, float r0, float r1, int numsegs, int numturns)
{
  float dang = (float)numturns * 2.f * F_PI / (float)numsegs;
  float ang = 0.;
  glBegin(GL_LINE_STRIP);
  for( int i = 0; i <= numsegs; i++ )
  {
    float t = (float)i / (float)numsegs; // 0.-1.
    float newrad = (1.-t)*r0 + t*r1; // linearly interpolate from r0 to r1
    float x = xc + newrad * cosf(ang);
    float y = yc + newrad * sinf(ang);
    glVertex3f( x, y, 0. );
    ang += dang;
  }
  glEnd();
}
```

### Parametric Linear Interpolation (Blending)

What's this code all about?

In computer graphics, we do a lot of linear interpolation between two input values. Here is a good way to do that:

1. Setup a float variable, \( t \), such that it ranges from 0. to 1. The line
   ```c
   float t = (float)i / (float)numsegs;
   ```
   does this.

2. Step through as many \( t \) values as you want interpolation steps. The line
   ```c
   for( int i = 0; i <= numsegs; i++ )
   ```
   does this.

3. For each \( t \), multiply one input value by \( (1.-t) \) and multiply the other input value by \( t \) and add them together. The line
   ```c
   float newrad = (1.-t)*r0 + t*r1;
   ```
   does this.

We Can Also Use This Same Idea to Arrange Things in a Circle and Linearly Blend Their Colors

```c
int numObjects = 9;
float radius = 2.f;
float xc = 3.f;
float yc = 3.f;
int numSegs = 20;
float r = 50.f;
float dang = 2.f*F_PI / (float) ( numObjects - 1 );
float ang = 0.;
for( int i = 0; i < numObjects; i++ )
{
  float x = xc + radius * cosf(ang);
  float y = yc + radius * sinf(ang);
  float t = (float)i / (float)(numObjects-1); // 0.-1.
  float red   = t; // ramp up
  float blue = 1.f - t; // ramp down
  glColor3f(red, 0., blue);
  Circle(x, y, r, numSegs);
  ang += dang;
}
```

### By Understanding what the Sine Function Looks Like, We Can Also Use it to Control Animations Based on Time

In your sample.cpp file, we have some code that looks like this:

```c
float Time; // global variable intended to lie between [0.,1.)

const int MS_PER_CYCLE = 10000; // 10000 milliseconds = 10 seconds

// in Animate( ):
int ms = glutGet(GLUT_ELAPSED_TIME);
ms %= MS_PER_CYCLE; // makes the value of ms between 0 and MS_PER_CYCLE-1
Time = (float)ms / (float)MS_PER_CYCLE; // makes the value of Time between 0. and slightly less than 1.
```

The sine function goes from \(-1\) to \(+1\), and does it very smoothly

\[ y = \sin(2 \cdot \pi \cdot \text{Time}) \]
Increasing the Amplitude, Increasing the Frequency

\[ \sin(2 \pi \text{Time}) \]
\[ 2 \sin(2 \pi \text{Time}) \]
\[ \sin(2 \cdot 2 \pi \text{Time}) \]

Increasing the Amplitude, Increasing the Frequency

\[ A \sin(F \cdot (2 \pi \text{Time})) \]

Changing this number changes the Amplitude

Changing this number changes the Frequency

Oscillating Motion

Let's say you want a block to oscillate back and forth in *x*:

This code would cause it to do that:

```c
if (in Display() )
    float x = X*sin(F*(2.*pi*Time) )
    ...
    glTranslatef( x, 0., 0. );
    glCallList( BlockList );
```

Rocking Motion

Let's say you want a block to rock back and forth:

This code would cause it to do that:

```c
if (in Display() )
    float theta = 45.f*sin(F*(2.*pi*Time) )
    ...
    glRotatef( theta, 0., 0., 1. );
    glCallList( BlockList );
```