Avoiding some floating point pitfalls

Sam Buss

October 18, 2004; revised January 28 2017

For the programming project #3 assignment, you will need to create a surface of rotation. This will require writing loops that draw all the vertices; indeed, you will have an outer loop that loops over all quad strips and an inner loop that outputs all the vertices of the quad strip. Likewise, you will need a loop that generates the vertices of the triangle fan. (Possibly, you will use triangle strips instead of the quad strips and triangle fan, but the same issues will arise.)

The purpose of this note is to discuss some potential pitfalls due to floating point roundoff. The first moral is: **Do not use floating point variables to control the stopping condition of a loop!** The second moral is **Do not compute the same floating point number in two different ways!**

Suppose you need to generate equally spaced real values that split the internal $[0, 2\pi]$ into equal length intervals. To begin with, here is a very bad piece of code:

1. Very bad code:

```
for ( float x = 0.0; x!=2.0*PI; x += 2.0*PI/MeshCount ) {
   // Use the value of x here.
}
```

For the above code, `PI` is a floating point value defined elsewhere. Also, `MeshCount` is a small positive integer value defined elsewhere.

The problem with the above code is that, due to floating point roundoff, the value `2.0*PI/MeshCount` will almost never be exactly correct. Thus, `x` may never *exactly* equal `2.0*PI`, and the loop will never exit!

A slightly better piece of code is next, but this is still bad:

2. Bad code:

```
for ( float x = 0.0; x<2.0*PI; x += 2.0*PI/MeshCount ) {
   // Use the value of x here.
}
```

This uses "`<`" instead of "`==`" to stop the loop.. This second sample code will at least always terminate. However, due to roundoff errors, after `MeshCount`+1 iterations, `x` will sometimes be slightly less than `2.0*PI` and sometimes slightly more than `2.0*PI`. In the former case, the loop will do one more iteration than expected.

A better method is to use an integer variable to control the loop parameter. For example:

3. Good code:

```
float x = 0.0;
for ( int i = 0; i<MeshCount; i++ ) {
   x += 2.0*PI/MeshCount;
   // Use the value of x here.
}
```

For the loops in the class programming assignments, the loop will probably include the last value 2π and will actually use the value $\sin(x)$ and $\cos(x)$. The first thing you might try is:

4. Fair code:

```
float x = 0.0;
for ( int i = 0; i<=MeshCount; i++ ) {
   x = i*2.0*PI/MeshCount;
   float sinX = sin(x);
   float cosX = cos(x);
   // Use the values of sinX and cosX here.
}
```

We called this "fair" code since it might appear to work correctly if you test it lightly, but it will still have problems in some cases. The problem is that, due to roundoff errors, `x`, will not end up exactly equal to 2π when `i` is equal to `MeshCount`; thus, for the last iteration of the loop, `sinX` and `cosX` may not have the desired values of 0 and 1. If these wrong values are used to draw a vertex with `glVertex`, then the vertex may end up on a wrong pixel, and leave a visible gap in the surface of revolution.

A better way to to write this code is:

5. Good code:

```
float x = 0.0;
for ( int i = 0; i<=MeshCount; i++ ) {
   x = (i%MeshCount)*2.0*PI/MeshCount;
   float sinX = sin(x);
   float cosX = cos(x);
   // Use the values of sinX and cosX here.
}
```

This code uses `i%MeshCount` so that `x` will be exactly zero when `i` equals `MeshCount`.

A when you need to use two angles `x1` and `x1` corresponding to i and $i = 1$ in the same loop, the best way to write the code is as:

5. Good code:

```
float x = 0.0;
for ( int i = 0; i<=MeshCount; i++ ) {
   x1 = (i%MeshCount)*2.0*PI/MeshCount;
   x2 = ((i+1)%MeshCount)*2.0*PI/MeshCount;
   float sinX1 = sin(x1);
   float cosX1 = cos(x1);
   float sinX2 = sin(x2);
   float cosX2 = cos(x2);
   // Use the values of sinX1, cosX1, sinX2, and cosX2 here.
}
```

Note that `x2` is **not** computed as `x2 = x1 + 2.0*PI/MeshCount;` This would seem to be mathematically the same, but may be slightly different in practice due to roundoff errors. This will occasionally result in unsightly missing pixels in your surfaces.