PART 6: “Floats”

Introduction

In this tutorial, we're going to look at how you can get the computer to calculate with decimal values (numbers like 3.14, called *floats*), instead of only integral values (numbers like 3, called *integers*).

Getting the stuff you need

As usual, extract the package to a fresh directory (say, `c:\my_projects\floats`) and load up the .sln file. Remove the ‘hello world’ code in the `Tick` function.

Balls Again

Before we start working with floats, let's setup things so we can show something interesting. Let's load a Sprite again!

Add a *Sprite* variable above the tick function like this:

```
Sprite theSprite( new Surface("assets/ball.png"), 1 );

void Game::Tick( float deltaTime )
{
...
```

You may have noticed that some images are in .tga files, while others are in .png files. These are image file formats. Other formats are: .gif, .jpg, .bmp, and many others exist. Each format has advantages and disadvantages: .gif is small but can store only 256 unique colors; .jpg is often even smaller but is ‘lossy’, which means that it is only an approximation of the original image, .tga has a really simple internal layout.

The template loads images using a library, in this case FreeImage. A library is a collection of useful code that we can include in our project. Have a look at Surface::LoadImage (in surface.cpp, line 50) to see how FreeImage is used to load image files to a surface.

Now make your Tick function like this:

```
int spriteY = 0, speed = 1;
void Game::Tick( float deltaTime )
{
    screen->Clear( 0 );
    spriteY += speed++;
    if (spriteY > (512 - 50))
    {
        spriteY = 512 - 50;
```

```
        speed = (-speed * 40) / 50;
    }
    theSprite.Draw( screen, 20, spriteY );
}
```

Some interesting things here: the lines to add ‘speed’ to ‘spriteY’ have been collapsed into a single line by using the increment operation (speed++). This translates to: use speed here, and increment it right after obtaining its value. Alternatively, we can write: ++speed, which *first* increments the value, and then uses the already incremented value.

The bouncing also contains an interesting construct: the inverted speed is multiplied by 40, then divided by 50. Why? To dampen it: the ball will now loose energy. Alternatively, we could have written:

```
speed = -speed * 0.8;
```

But that is not the same thing. Try that line, and instead of hitting F5 to run the application, hit F7 to just build it. The compiler will give you a warning:

```
1>game.cpp(32): warning C4244: '=': conversion from 'double' to 'int', possible
loss of data
```

Variable `speed` is an integer, which means that it can store whole numbers only. So when we perform a calculation that involves a float (or a double, as in this case), the processor has to truncate the result to the nearest whole number, which is a potential loss of precision; hence the warning.

Switching to Floats

Let’s convert our program so it uses floats only. Here is the code:

```
float spriteY = 0, speed = 1;
void Game::Tick( float deltaTime )
{
    screen->Clear( 0 );
    spriteY += speed;
    speed += 1.0f;
    if (spriteY > (512 - 50))
    {
        spriteY = 512 - 50;
        speed = -speed * 0.8f;
    }
    theSprite.Draw( screen, 20, spriteY );
}
```

Not a lot changed: incrementing speed with ‘++’ doesn’t work with floats, so it was put back on its own line, and the damping now done using 0.8f. The code also still does the same thing. So what’s the big deal?

Well, in the previous version, speed could only be 1, 2, 3, … ; making the ball bounce slower would be a bit of a problem. With floats, we totally have that option. We can for instance have a low gravity bounce by adding 0.1f to speed. Or, we can adapt to frame rate.

One more thing changed: we now get a warning on the line that draws the sprite. Why? Sprite::Draw expects integer coordinates for the sprite. Sending it floats again forces the processor to use truncated values. To prevent the warning, we can do the truncation ourselves, so that it is clear that the loss of precision is intentional. We do this by adding (int) in front of spriteY:

```
theSprite.Draw( screen, 20, (int)spriteY );
```

Likewise, we can *cast* an integer to a float value:

```
int a = 300;
float b = (float)a;
```

This seems useless: the compiler would not warn us of such a conversion (since no precision is lost). However, you need to be aware that this conversion is happening: conversion takes time, and can be a performance issue once we start executing complex applications.

FRAPS

The bouncing ball application probably runs at 60fps on your machine. You can verify this using a tool called FRAPS. Get it here: http://www.fraps.com. When you start the application, it looks like this:

Click on the '99 FPS' icon. Now start your application again. FRAPS adds a framerate counter, which should confirm that you are indeed running at 60 frames per second. This is not a limit of the template; typically the driver of your video card enforces this. Let's remove this restriction. For that, we need to modify the template somewhat. This is going to get ugly.

Open template.cpp and locate the main function. This is where a template application starts. Right after the opening bracket of this function we are going to add a line:

```
SDL_SetHintWithPriority( SDL_HINT_RENDER_VSYNC, "0", SDL_HINT_OVERRIDE );
```

This line tells SDL2 (another library the template uses) to disable the *VSync*. What's a VSync? Well, old-skool monitors produced images by aiming an electron beam at a

glass plate (and then your eyes). This beam zig-zagged over the pixels to light them up. At the bottom of the screen, this beam stops, and returns to the top-left corner. This is called the 'vertical retrace'. This is the perfect moment to start displaying a new image; only during the retrace swapping can be done without tearing. So synchronizing frames to this retrace is useful. Modern monitors simulate this retrace, typically at 60Hz. Synchronizing to that limits your framerate to 60.

Now that we have lifted the limitation we can reach much higher framerates: on my system, I get ~450fps. But there's more. ☺

In the file `precomp.h` you will find a line that reads:

```
// #define ADVANCEDGL
```

The start of this line indicates that it is a comment. This is a popular way to leave functionality in a source file without enabling it. We can apply the setting by removing the slashes:

```
#define ADVANCEDGL
```

Now run the application again. This setting enables a faster internal code path of the template. Sadly, this code path is not compatible with all graphics adapters, which is why it is disabled by default. On my machine it works, and now I get almost 700fps.

Too Fast to be Fun

By now, the ball sprite is bouncing frantically on your screen. We could easily reduce its speed, but by how much? One system runs at 450fps, another at 700, and perhaps your ancient Asus EEE barely achives 60fps…

We can solve that.

Add the following line at the top of your `Tick` function:

```
    printf( "%f\n", deltaTime );
```

This prints the mysterious deltaTime variable that we had available since the start of this course. The numbers that you see in the text window are the frame durations, in milliseconds. In other words: we *know* how long it takes to render one frame.

And that is very useful if you want something to move over the screen at a constant speed, regardless of framerate. Which takes us directly to the…

Assignment

Convert the bouncing ball application to floating point. Make sure that the ball moves at a constant speed, regardless of framerate. Make sure your code compiles without compiler warnings.

END OF PART 6

Next part: "Debugging"