PART 5: "Conditions"

Introduction

This tutorial is about the condition commands C/C++ has which allow you to make automated decisions in your program. You already used conditions when you did loops (perhaps without realizing it), but in this episode we will look at more explicit conditions.

Getting the stuff you need

As in the previous episodes, we'll use the template again. But you will need to know how to use your sprites from episode 3 so if you have not done that one please go back and do so now. Extract a fresh version of the template, remove the hello world code, and then we're ready to begin.

At this point you probably have several projects that are all called 'tmpl_2017-01', which can get annoying, especially if you have several projects open at once. You can quite easily rename your project:

Step 1 is to close Visual Studio. Then, run `clean.bat`. This removes all files that will be regenerated by Visual Studio when you recompile the project. It reduces the size of the folder and gets rid of clutter.

Next, rename the project files: `tmpl_2017-01.sln` becomes e.g. `myproject.sln`; `tmpl_2017-01.vcxproj` becomes `myproject.vcxproj` and `tmpl_2017-01.vcxproj.filters` becomes `myproject.vcxproj.filters`. Make sure you use the same name each time (in this case, `myproject`).

Now, right-click `myproject.sln`, select 'Open With', and use Notepad to edit the file. You will see that `tmpl_2017-01.vcxproj` is mentioned in there; replace this by `myproject.vcxproj`.

Optionally, replace `"Template"` by something else, e.g. `"Episode 5"`. If you do this, you also need to open `myproject.vcxproj` in Notepad. Look for a line that reads `<ProjectName>Episode 5</ProjectName>` and replace Template by 'Episode 5' here as well.

Now open the .sln file with Visual Studio 2015 again. Remember, Visual Studio is just a tool, and we just gained a bit more control over it.

To Bounce or not to Bounce

In your `Game::Tick` loop from the last lesson you put a sprite on screen. Let's do that again.

Add a *Sprite* variable <u>above</u> the `Tick` function:

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

Make your `Tick` function look like this:

```
void Game::Tick( float deltaTime )
{
    screen->Clear( 0 );
    theSprite.Draw( screen, 0, 0 );
}
```

Try and run this and make sure you get an image on screen at the top left corner.

Ok?

Good let's continue
Lets move our sprite down a little bit and then move it around using your variable skills, Above the tick function and below the Sprite definition add a couple of global variables.

```
int spriteX = 0;
int spriteY = 100;
int speed = 1;
```

Hopefully these variable names will explain their function. Notice also that each variable is set to a value.

Now let's alter our game tick a little:

```
void Game::Tick( float deltaTime )
{
    screen->Clear( 0 );
    theSprite.Draw( screen, spriteX, spriteY );
}
```

Run this and you should see your sprite is now drawn a little down the screen.

Now let's make the sprite move. Add two lines to the `Tick` code:

```
    spriteY += speed;
    speed += 1;
```

Now the sprite quickly leaves the screen. It would be nice obviously if it would bounce back at the bottom… That's where conditions come in. 'If it reached the bottom' is a condition, and we can actually use something very close to that in our code:

```
if (spriteY > (512 – 50)) { spriteY = 512 – 50; speed = -speed; }
```

That line looks simple enough, but there's a lot of detail in there.

First, '512': That's the bottom of the screen alright, but sprites are positioned with their upper-left corner. The sprite we use has 50 rows of pixels, so we make it bounce on line 512 – 50.

Next, the code that is executed if the condition is true: this is a single-line code block surrounded by curly brackets. You can of course put it on separate lines, but this is also valid. Making the ball bounce is a matter of inverting its vertical speed. Then there's one

thing left to do: we may already have left the screen, so we make sure that the ball is positioned exactly at the bottom in that case.

You may notice there are parentheses around 512-50. *Why?* Well, we want to compare against 512-50, instead of subtracting 50 from the result of comparing against 512 (whatever that would mean). I wasn't sure about what takes precedence ('<' or '-') so to be sure I used the parentheses to make it completely clear.

Read more about operator precedence here: http://en.cppreference.com/w/cpp/language/operator_precedence; this will tell you that in fact it was safe to skip the parentheses.

Or else

To add a bit of drama to the bouncing, we can flash the background when the ball hits the bottom of the screen. Try this:

```
void Game::Tick( float deltaTime )
{
    spriteY += speed;
    speed++;
    if (spriteY > (512 - 50))
    {
        spriteY = 512 - 50;
        speed = -speed;
        screen->Clear( 0xff0000 );
        theSprite.Draw( screen, spriteX, spriteY );
    }
    else
    {
        screen->Clear( 0 );
        theSprite.Draw( screen, spriteX, spriteY );
    }
}
```

This time, we have *two* code blocks related to the condition. The second block is executed when the condition is false.

Booleans

In many programming languages, 'true' and 'false' are values. We can store these in a boolean, e.g.:

```
bool hitBottom = spriteY > 512 – 50;
if (hitBottom) { … }
```

Interesting question is of course how the computer stores these values in memory. Let's try to gain some insight. Modify your Tick so that it uses the `hitBottom` boolean. Then, print the boolean to the text window:

```
printf( "%i\n", hitBottom );
```

Run the program, and bring the text window into focus. Things move fast, but if you watch carefully, you will see that ‘false’ apparently equals ‘0’, and ‘true’ equals ‘1’. In C/C++, this works both ways: ‘0’ is ‘false’, and anything that is not zero is ‘true’.

So:

```
int a = 1000, b = -1000;
if (a && b)
{
    printf( “both true.\n” );
}
```

…will indeed claim that *a* and *b* are both ‘true’. Not directly useful, but now you know. ☺

Assignment

Let’s combine some things from the last couple of episodes. Add a variable ‘speedx’ to make the ball move horizontally as well. Use this to make the ball bounce from the left to the right side of the screen. When it reaches the right side, make it deflect on the right wall so it starts moving back to the left. After, say, 50 bounces, stop testing the screen bottom collision, so that the ball falls through and disappears.

END OF PART 5

Next part: “Floats”