PART 9: “Colors”

Introduction

In this article we will return to the numbers that represent colors, as demonstrated in the second installment of this series. This can only be done through the wonderful world of bit magic, which is by the way a very nice place to be, so we will explore it thoroughly. As an introduction, try the following Tick function in a fresh template:

```
void Game::Tick( float deltaTime )
{
    screen->Clear( 100 );
}
```

What you get is a somewhat dark-bluish backdrop. Change ‘100’ to ‘255’, and it will be brighter blue. Increase it one more, and it will be black. Well, not really, actually it’s very dark green, as we will see in a minute.

Ingredients of an integer

Let me introduce you to some new C things. Try the following Tick function:

```
void Game::Tick( float deltaTime )
{
    union
    {
        int i;
        unsigned char b;
    };
    i = 255;
    i++;
}
```

Set a breakpoint at the `i++` line and start the application. Use the debugger to view the value of variable `i` and `b`. At the breakpoint position, `i` and `b` both contain 255, but when you proceed one line, `i` is 256 as expected, but `b` is now 0.

A word about this weird ‘twinning’ of variables: `i` and `b` are in a *union*. You can use a union to have two (or more) variables that share the same memory location. The result of that is that changing one of them will change the other as well.

In this case, `i` is an integer, but `b` is a char. Both can contain numbers, but there is a difference: an integer is 32 bit, and a char is 8 bit. Since a bit can contain ‘0’ or ‘1’, it can contain 2 unique values; 8 bits can therefore contain 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 256 unique values (0...255). After that, it’s back to zero... If we look at the separate bits (binary numbers), we get this situation:

```
00000000000000000000000 11111111   =   255
00000000000000000000001 00000000   =   256
```

What does this have to do with colors? Hang in there, we'll get to that in a minute! There's another thing about those binary numbers. Have a look at the following table:

```
00001  =  1
00010  =  2
00100  =  4
01000  =  8
10000  = 16
```

So, if we take a number, and multiply it by 2, we effectively shift the binary number to the left. Or, when we swap that around, shifting the binary number to the left multiplies it by 2; shifting it 'a bit' to the right divides it by 2.

Now let's return to colors. To store a color in an integer, we in fact store three values: red, green and blue. Each of them needs 8 bits. Blue goes in the lowest 8 bits, and that is why any value between 0 and 255 will get you a shade of blue. Green goes in the next 8 bits. To get there, we need to shift 8 bits to the left, which we do by multiplying by 256. So, to get shades of green, we take a number between 0 and 255, and multiply it by 256. And to get to red, we multiply by 256 twice. So:

```
screen->Clear( 100 );                    // yields dark blue
screen->Clear( 100 * 256 );              // yields dark green
screen->Clear( 100 * 256 * 256 ); // yields dark red
```

And thus, the brightest red that you can get is 255 * 256 * 256. You can also mix red, green and blue, by adding them together:

```
screen->Clear( (200 * 256 * 256) + (200 * 256) ); // yellow
```

So, to blend some colors, you will be doing plenty of multiplications. There is a slightly easier way: using *bitshifting*. Try this:

```
int i = 1;
i = i << 1;
```

Using the debugger, you can see that '<< 1' multiplies a value by 2. Likewise, '<< 2' will multiply it by 4. For colors, we can use '<< 8' and '<< 16':

```
screen->Clear( (200 << 16) + (200 << 8) ); // yellow
```

Now it is a bit more clear what's happening: We are taking a value of 200 here for red, and we shift it in the right position (which is 16 bits to the left). Likewise, we put 200 for green in the right position by shifting it 8 bits to the left. And, consider this to be a little teaser, you can also store some stuff in the invisible alpha channel, by shifting it to the left by 24 bits...

Bitmasks

Now that we can get red, green and blue exactly where we want them to be, we can start answering the question how to access them separately.

Suppose you have a yellow pixel. It's stored in a 32-bit variable, which looks like this, in binary:

```
11010111 11111111 11111111 00000000
```

So: Garbage in the leftmost eight bits (bits 31-24, alpha), then 255 for red in bits 23-16, then 255 for green in bits 15-8, and finally zeroes in bits 7..0. The garbage might not actually be there, it could be zeroes, or ones, but that doesn't matter. Now suppose we want to know what green is for this color. Of course you can see it right away, but we need to do this in a program, so let's see how we can make C++ extract that color.

The answer is: using *bitmasking*. We will be using the '&-operator' (pronounced as 'And') in this case. In code, it looks like this. Note that I used a different value for green (yielding a bright orange) to illustrate the concept.

```
int color = (255 << 16) + (237 << 8); // orange
int mask = 255 << 8; // mask for green
int green = color & mask;
```

In binary, the following happens:

```
11010111 11111111 11101101 00000000    // orange, garbage in alpha
00000000 00000000 11111111 00000000    // mask for green
----------------------------------- &
00000000 00000000 11101101 00000000
```

What '&' does is this: Every individual bit that was '1' in the first value, and also '1' in the second value, will be '1' in the result. All other bits will be '0' in the result. The bottom line is that we extracted the value for green. Well, almost: it's still shifted to the left by 8 bits, so to get the correct value for green, we need to move it back:

```
int green = (color & mask) >> 8;
```

Red, blue and alpha can be extracted in the same manner.

There's a lot more to explore when it comes to bit magic, and trust me, you will learn to love it. But now, let's make things practical, in the assignment for this week.

Assignments are on the next page!

Assignment

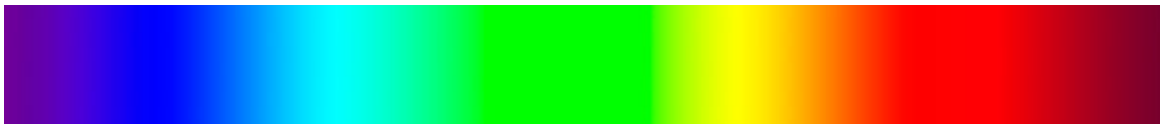
Here are your tasks for today:

1. Load an image, display it, and fade it (slowly) to black.

Note: there are two ways to do this. The easy way is using floating point operations. The slightly harder approach uses integers only. Try both, and measure the speed difference.

2. Draw a 200x200 checkerboard pattern with black and white pixels. Next to it, draw a 200x200 pixel solid grey bar. Adjust the brightness of the grey color so that it matches the brightness of the checkerboard pattern. Is the result what you expected?

3. **EXTRA / HARD:** draw the colors of the rainbow in vertical lines:



   This will require some research; what you are looking for is a conversion from wavelength to red/green/blue.

Once you have completed these, you may continue with the next part.

END OF PART 9

Next part: "Arrays"