PART 14: “Fixed Point”

Introduction

In article 9, you where introduced to the noble art of bitmagic. Here’s a quick refresher: multiplying an integer value by (a power of) 2 can be done by shifting its (binary) bits to the left. Division is similar, except this time you shift to the right. So, `a*6` is: `(a<<2)+(a<<1)`. Being aware of the bits in a number is especially important if more than one number is stored in a 32 bit integer. This happens for colors: red, green, and blue take 8 bits each, with ‘alpha’ completing the 32 bits used for a single pixel.

Scaling

To introduce some new concepts, I will start with a brief snippet that slowly zooms in on an image:

```
Surface* img = new Surface( "assets/aagun.tga" );
float dx = 2.0f;
void Game::Tick( float deltaTime )
{
    Pixel* dst = screen->GetBuffer(), *src = img->GetBuffer();
    for ( int x = 0; x < SCRWIDTH; x++ ) for ( int y = 0; y < img->GetHeight(); y++ )
    {
        int readxpos = (int)(dx * x);
        Pixel sample = src[readxpos + y * img->GetWidth()];
        dst[x + y * SCRWIDTH] = sample;
    }
    dx *= 0.999f;
}
```

The code works like this:
First, we copy pixels from `img` to `screen` (by accessing the pixels in the Surface, via `GetBuffer()`). Variable `dx` is set to 2, so when `x` is 10, `dx * x` is 20: we read pixel 20 from the original image.
For the next frame however, `dx` is no longer 2.0: it slowly gets smaller. After a few frames, `dx` is much smaller than 2, and so we read far less pixels from img, but we still plot the same amount of pixels. This causes the stretching effect.

Some other remarks about this code:

- If you do not add ‘f’ to 2.0, you will get a warning. This is because C++ assumes that 2.0 is a double precision floating point number, and you are storing it in a single precision float variable. Don’t worry about it, just add the `f` to prevent the warning.
- Pointers are used as arrays in this code. This works, because an array *is* a pointer in C++. And the other way round: `Pixel* screen` is an array, and therefore you can access the tenth pixel as `screen[10]`.

There’s something else going on in this code, something that is not immediately obvious. And that’s the bit that I want to talk about.

Conversions

Have a look at this line again:

```
int readxpos = (int)(dx * x);
```

We need an integer to read from the image pixel array. But, `dx * x` is not an integer. Actually, x *is* an integer, but as soon as you multiply an integer by a float, the result is a float too. Or, to be precise: the integer is converted to a floating point value, and *then* the multiplication is done. So, there are actually two conversions on that line:

```
int readxpos = (int)(dx * (float)x);
```

Is that a problem? Yes it is. Conversions take time, and even if it's not a lot, in this case it can easily be prevented.

Before you proceed, get yourself a larger image, preferably one that is 512 pixels high (don't exceed 512, it will crash the application). I got this one from the internet:

When you try the larger image in the application, you'll see that the speed went down (quite) a bit. Getting rid of the conversions will improve that.

Fixed point math

So how do you get rid of conversions, if you need integers for the array, and a float to store 0.999f? The answer is: *fixed point math*. The concept is pretty simple. Suppose you 'forget' the dot in 0.999f. Instead, you write: 999, and you remember that this number is a thousand times too large. Next, you do all your calculations as usual. So: `readxpos = dx * x`. Once you're done calculating, you get the correct answer by dividing by 1000. What you just did is exchanging an int-to-float conversion and a float-to-int conversion for a division. This in itself is not a big improvement (if at all), but it does become a big improvement when you realize that '1000' is just an arbitray factor. For a computer, a power of 2 is much better: 1024 would do just fine.

There's just one thing you need to be aware of. When you multiply two numbers that are both 1000 times too large, the result is 1000000 times too large. You can see how this is countered in the modified code snippet:

```
Surface* img = new Surface( "assets/aagun.tga" );
int dx = (int)(2.0f * 1024.0f);
void Game::Tick( float deltaTime )
{
    Pixel* dst = screen->GetBuffer(), *src = img->GetBuffer();
    for ( int x = 0; x < SCRWIDTH; x++ ) for ( int y = 0; y < img->GetHeight(); y++ )
    {
        int readxpos = (dx * x) >> 10;
        Pixel sample = src[readxpos + y * img->GetWidth()];
        dst[x + y * SCRWIDTH] = sample;
    }
    dx = (dx * 1023) >> 10; // 1023 is the new 999
}
```

In other words: when you multiply two numbers that are both 1024 times too large, the result is 1024 * 1024 too large. You can fix this by dividing by 1024. In the code above, this is done by shifting 10 bits to the right.

The code is now a lot faster. We now have a *bitshift* instead of the two conversions. Apparently, this is a lot faster in C++.

Range versus Precision

In the above example, we got roughly three digits of decimal precision by multiplying our floats by 1024. This means that there is a now a limit to precision: the smallest number in an integer is still 1, which represents 1/1024 in our fixed point notation. What if we want things to be more precise?

Well, simple: we scale by a bigger power of 2. We used 1024, which is 2^{10}; let's go for 2^{16} instead, which is 65536. If you try this in the fixed point code, you will notice that things break… Why?

Well: there is this line that says 'dx = (dx * 1023) >> 10'. What kind of numbers are we processing here? Variable dx starts at 2 * 1024 = 2048; this fits in 12 bits (almost 11, but 2047 is the largest number that still fits in 11 bits). Then we have '1023', which takes 10 bits. Multiply them together, and we get results that require 21 bits. No problem. But what happens if dx takes 17 bits, and 1023 becomes 65535, which takes 16 bits? The result (before shifting back by 16) takes 33 bits, and that doesn't fit in an integer! Things do work however if we use 2^{15} instead, which is 32768.

With fixed point arithmetic you will need to carefully balance the *range* of your numbers, and their *precision*.

Assignment is on the next page!

Assignment

Your tasks for today:

BASIC

1. Extend the code so that it zooms simultaneously in x and y.
2. Modify the code so that it can zoom out, instead of just zooming in. Use a modulo (%) to make sure the source image gets repeated if it's too small.

Once the modulos are in, you'll notice that your code gets quite a bit slower again. This is because modulo is performing a division (which is slow). If you make your source image size a power of 2, you can get rid of the modulo: `% 256` can be calculated more rapidly using `& 255`, and likewise `% 512` can be calculated using `& 511`. But that's bit magic again. ☺

INTERMEDIATE:

3. Modify the code so that the zooming image appears to bounce against your screen.

ADVANCED:

4. Add rotation to the zooming image. Keep using fixed point arithmetic in the inner loop.

END OF PART 14

Next part: "Mathematics"