Mandelbrot set rendering

Introduction

With the arrival of DirectX9 level hardware a whole new world of possibilities has opened up in graphics. One such possibility that the new floating point pixel shaders have opened is the ability to evaluate advanced mathematical operations without significant precision loss or with limited range. The kind of applications one tend to think of first where the capabilities of DirectX9 shaders can be beneficial is often various lighting scenarios, atmospheric effects, animation etc. These are all very interesting topics to dive deep into, but there's another dimension that these shaders open up that may not have crossed our minds the first time we learned about the new shaders. For the first time we can utilize pixel shaders to visualize the wonderful world of fractals.

The Mandelbrot fractal

The probably most famous and most well-known fractal is the Mandelbrot set. The Mandelbrot set is basically a set consisting of the complex number that after an infinite number of iterations of a simple formula still is within close range from the origin. While the higher math behind all this and all it's implications is something that interests only a selected few the graphical art you can produce with such series is something that can amaze just about everyone.

How does one visualize the Mandelbrot set? Easy, you simply take a complex number, evaluate a function on this number and get a new number. Then repeat this in a sufficient number of times. The classical iteration looks like this.

$$Z_{i+1} = Z_i^{\,2} + C$$

Here C is the original number and Z_i is the number we are working with. We begin by setting Z_0 to C. Expanding this formula into real and imaginary parts of the complex number we get these two formulas.

$$X_{i+1} = X_i^{\,2} - Y_i^{\,2} + C_x$$
$$Y_{i+1} = 2X_iY_i + C_y$$

The X is the real part and Y is the imaginary part. Now we only need to do this math in a pixel shader. The first thing we need to do is pass the constant C to the pixel shader. Where do we get C from, what is it really? As we want to visualize the Mandelbrot set we want to view every point in the XY plane that belongs to the Mandelbrot set such that these are clearly different from the pixel that doesn't belong to the set. This means that we are interested in those points in the XY plane that after an infinite amount of iterations of the formulas above still close to the origin. So what we pass as C is basically the position of a point in the XY plane. We will define a subset of the plane, for instance the rectangle (-2, -2) - (2, 2), and draw this range as a quad covering the whole viewport. C is basically the position, and thus we will pass it as a texture coordinate which will be interpolated over the surface. The Mandelbrot set definition declares that we need to loop the equations above an infinite number of times in order to decide whether a point is within the set or not. Obviously this is impossible to do, so usually one just loop it a sufficient number of times and then decide if we are still close to the origin. If we are, then we assume we are part of the Mandelbrot set, a fairly reasonable assumption.

An iteration of the formulas above can be done in three instructions. C is passed in texture coordinates t0, r0 contains our Z in its x and y components and r2 is a temporary register. The implementation will look like this.

```
mad        r2.xy, r0.x, r0,   t0
mad        r1.x, -r0.y, r0.y, r2.x
mad        r1.y,  r0.x, r0.y, r2.y
```

As you can see the result ends up in another register, r1. This is because both the x and y components of the previous value is needed in the evaluation of both the new value, so we can't overwrite any of them. So we write the results to r1 instead. In the next iteration we can do the same thing again, but with r0 and r1 reverse such that the next result ends up in r0 again. Then we only need to take these two iterations and cut'n'paste them until we reach the limit of the hardware. A Radeon 9700 for instance accepts pixel shaders of at most 64 ALU

instructions. This means we can get at most 21 iterations, but in reality probably fewer because we will probably prefer to do something cool with the end result before we write it to the framebuffer. In the code for this article we will end up with 19 iterations.

Visualizing it

Alright, so we have done our 19 iterations, now what? Well, we need to transform it into something meaningful for the eye. There are zillions of ways to do this, and which one we choose is arbitrary and can be based just on our subjective preference. For all this to be meaningful though we need to make the pixels that end up in the Mandelbrot set to be visually different from those that didn't. Traditionally, when one render Mandelbrot sets on the CPU, people have used the number of loops it took until we ended up at a distance larger than 2 from the origin. This is then used to look up a color from a palette. Unfortunately, this kind of information is not available for us. As of this writing there's no support for data based branching in the pixel shaders in any hardware available on the market, so we can't count loops. All the information we have is the final position after all our iterations. This is sufficient however, and we'll map this distance to a color. A large distance means it's not in the Mandelbrot set while a small distance means it most likely is. To get some nice coloration we will use the distance as a texture coordinate and lookup the color from a texture with a dependent texture read. This texture is one-dimensional and contains a color spectrum not too different looking from a rainbow, except that it's softly fades to black to the right. The distance can be anywhere from zero to very high, so instead of just mapping it directly we will use a similar formula as when one map high dynamic range images into the 0…1 interval with exposure control. The formula we will use is this.

$$R = 1 - 2^{-cd}$$

R is our resulting color and d is our distance. Instead of bothering with taking the square root in order to find the distance we will just use the squared distance, mind you, this final step is no exact science, it is better classified as art. The constant c is just an arbitrary constant that says how far from the origin a point can be without mapping to our black edge of the texture. We select it purely on subjective grounds and I have found that something around 8 will suit us well. The final implementation is pretty straightforward.

```
def       c0, 0.0, 1.0, 8.0, 0.0
...

mov       r1.z, c0.x
dp3_sat   r0, r1, r1
mul       r0.x, r0.x, c0.z

exp       r0.x, -r0.x
sub       r0, c0.y, r0.x

texld     r0, r0, s0

mov       oC0, r0
```

First we fill r1.z with a zero so we can use the dot product instruction without reading uninitialized components. You may wonder why we use a dp3_sat, shouldn't we use dp3. Well, we should. Unfortunately, in practice though some implementations seams to have problems raising numbers to high negative numbers; this can create some noisy artifacts. However, as 2^{-8} is already a very small number there will be no visual difference if we clamp it. We should now have a nice colored Mandelbrot before our eyes.