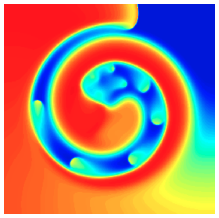
Abubu.js

The WebGL Computational Library



Iterations: drawing the Mandelbrot set

The Mandelbrot set is one of the most famous fractals in mathematics. It also is a classical example where parallel computing can be employed to speed up the computations.

The Mandelbrot sequence is constructed as:

$$z_n = z_{n-1}^2 + z_0$$

where z_0 is the coordinate of the point on the complex plane and z_n is the complex number after n iterations starting from z_0. A point z_0 is a member of the Mandelbrot set if and only if the above sequence remains bound.

It can be proven that if $|z| > 2$, the sequence will diverge. So, it's enough to check if the sequence modulus remains below 2.

Now, we want the canvas to be our solution domain where each pixel represents a point on the complex plane such that

$$-3 < \Re(z_0) < 1$$

$$-2 < \Im(z_0) < 2$$

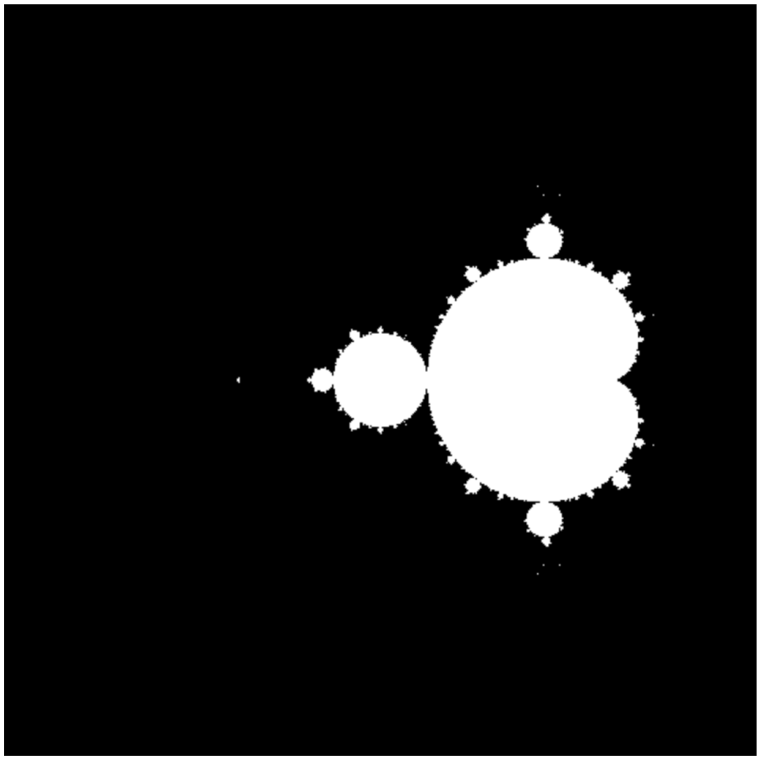
where $\Re$, $\Im$ are the real and imaginary parts, respectively. Since the pixel positions in horizontal and vertical directions are between 0, and 1 it is really easy to construct the initial values of z_0 based on pixel position assuming the real axis is along the horizontal direction of the canvas (along `cc.x`) while the imaginary axis is along the vertical direction of the canvas (along `cc.y`).

We can easily construct the program by just modifying fragment shader from the previous example:

```
#version 300 es
precision highp float ;
precision highp int ;

out vec4 outcolor ; /*  output of the shader
                        pixel color             */
in vec2 cc ;        /* input from vertex shader */

// Main body of the shader
void main() {
    vec2 z0 = cc*4. - vec2(3.,2.) ; /* Initial coordinate based
                            on pixel position */
    vec2 z = z0 ;   /* initialize sequence with initial coordinate */

    /* Iteration loop to march the iterative map for a 1000 times */
    for(int i=0; i<1000; i++){

        /* Mandelbrot map */
        z = vec2(z.x*z.x-z.y*z.y,2.*z.x*z.y) + z0 ;

        if (length(z)>2.){  /* if the point is not part of the set
                            color it black and end fragment shader */
            outcolor=vec4(0.,0.,0.,1.) ;
```

```
            return ;
        }
    }

    /* if we managed to finish the loop the point is part of the set
       so we can color it white */
    outcolor = vec4(1.) ;
    return ;
}
```

Running this program will produce the picture below.



[The Mandelbrot set program]

Why are the calculations so fast?

It is noticeable that rendering all the pixels of the picture is almost instantaneous. It might seem even faster than downloading the picture from the internet. The reason is the automatic parallelization that takes place here. As you might have noticed by now, in the fragment shaders, there is no looping over the pixels. Only a recipe is provided for coloring all the pixels.

The graphics card usually have several computational cores that can be utilized. Modern GPUs can have thousands of cores. The *WebGL 2.0* engine here automatically utilizes as many cores as possible to concurrently color the pixels. This means that several pixels are

processed at the same time which provides a high level of parallelization and hence the speed of the code.

Let's have a look at the overall HTML page code

```
<!DOCTYPE html>
<html>
<!-- Head -->
<head>
<script src='http://abubujs.org/libs/Abubu.latest.js'
        type='text/javascript'></script>
</head>
<!-- body of the html page -->
<body>
    <canvas id="canvas_1"
            width=512 height=512
            style="border:1px solid #000000;" >
        <!-- This message is displayed if canvas is not available -->
        Your browser does not support the HTML5 canvas tag.
    </canvas>
</body>

<!--&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&-->
<!-- fragment shader                                                  -->
<!--&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&-->
<script id='fshader' type='shader'>#version 300 es
precision highp float ;
precision highp int ;

out vec4 outcolor ; /*  output of the shader
                        pixel color             */
in vec2 cc ;        /* input from vertex shader */

// Main body of the shader
void main() {
    vec2 z0 = cc*4. - vec2(3.,2.) ; /* Initial coordinate based
                            on pixel position */
    vec2 z = z0 ;   /* initialize sequence with initial coordinate */

    /* Iteration loop to march the iterative map for a 1000 times */
    for(int i=0; i<1000; i++){

        /* Mandelbrot map */
        z = vec2(z.x*z.x-z.y*z.y,2.*z.x*z.y) + z0 ;

        if (length(z)>2.){  /* if the point is not part of the set
                            color it black and end fragment shader */
            outcolor=vec4(0.,0.,0.,1.) ;
            return ;
        }
    }

    /* if we managed to finish the loop the point is part of the set
       so we can color it white */
```

```
    outcolor = vec4(1.) ;
    return ;
}
</script>

<!--&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&-->
<!-- Main script -->
<!--&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&-->
<script>
// get the shader source by its id ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function source(id){
    return document.getElementById(id).text ;
}

// Get the canvas ------------------------------------------------------
var canvas_1 = document.getElementById('canvas_1') ;

// Setup a solver ------------------------------------------------------
var renderer = new Abubu.Solver( {
    fragmentShader  : source('fshader'),
    canvas          : canvas_1,
} ) ;

// rendering (running) the solver
renderer.render() ;
</script>
</html>
```

Notice how incredibly concise this code is. We only have a canvas, a shader, and a handful of lines to define a solver and run it. That's the major beauty of the **Abubu.js** library which enables you to concentrate on the numerical part, and almost forget about the graphical pipeline, and all the details involved with the *WebGL* directives and syntax.

[Download the source code for all tutorials]