TP02
Blurred fractal

Robin Faury

01/11/2021

Abstract

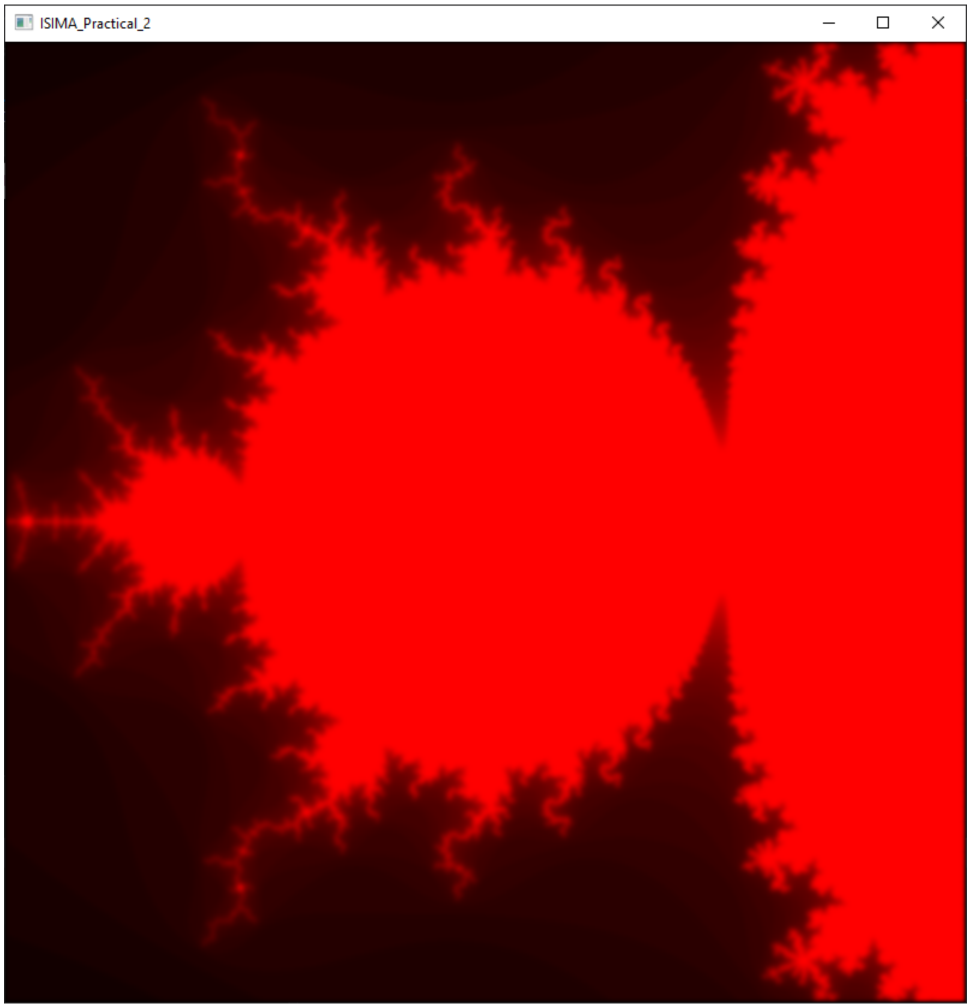

Figure 1: The blurred Mandelbrot set

1 Introduction

1.1 Cloning the repository

You can find the source of the practical work by cloning this git repository:

```
https://github.com/robinfaurypro/GPGPU_ISIMA_2020-2021.git
```

The CMakeLists file is stored into the Practical_2 folder. Open a terminal on this folder and use thus commands:

```
git submodule update --init
mkdir build
cd build
cmake -G "Visual Studio 15 Win64" ..
```

Of course if you are on UNIX system feel free to use the generator you want. If everything is going well, you can compile and run the executable and get a colored gradient window.

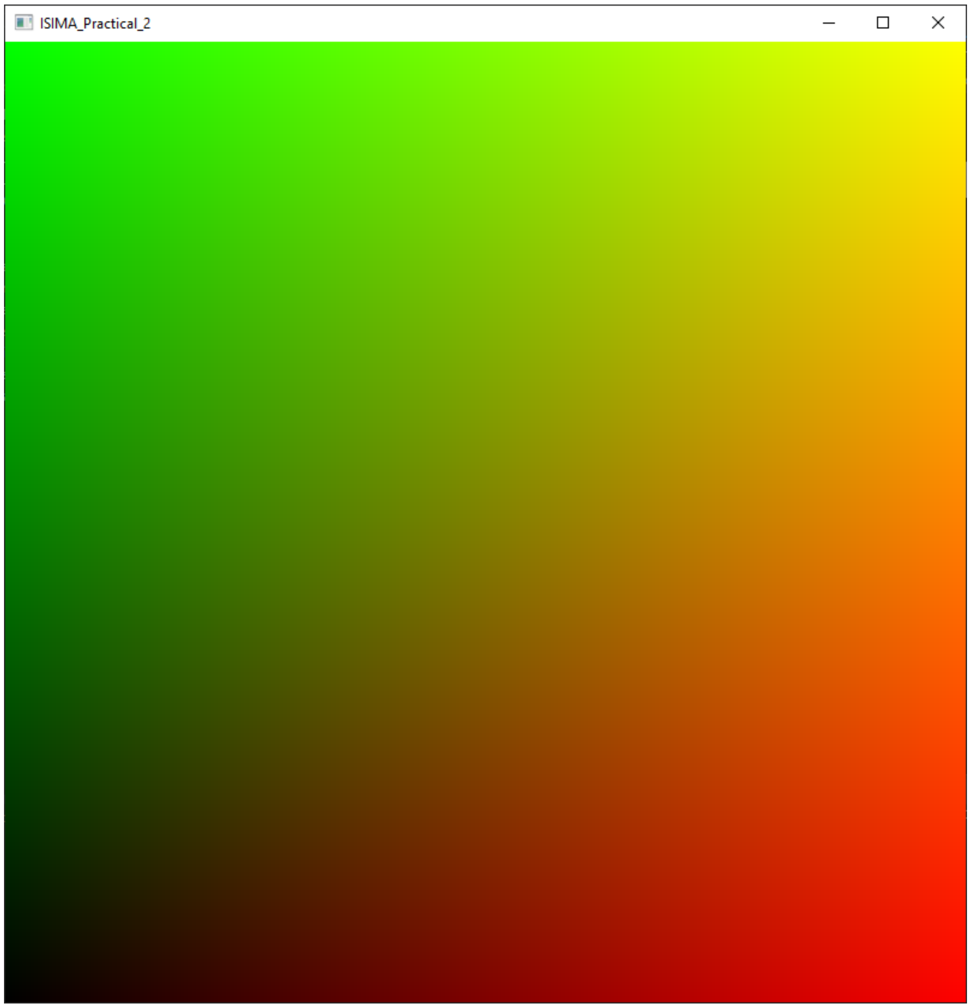

Figure 2: The UV of a image

The UV of an image is simply the coordinate of an image. An image have always UV between 0 and 1. As an example, an image with 1024 pixel on x and 2048 pixels on y get this U and V values: if $U = 0.25$, $X = 256$ and if $V = 0.25$, $Y = 512$.

2 GPU texture creation

On the last practical we use a buffer to store some particles and show them on screen. This time our buffer will store each pixel on the image. On the Content struct you can see that the texture_ item has replace the buffer_ one. For the moment this handle is set to 0. On the kFragmentSource you can see that we draw the uv coordinate if the texture isn't valid.
On the Initialization function we can create the memory space on the GPU memory. First we need to create the handle and bind the texture to work on it:

```
glGenTextures(1, &content.texture_);
glBindTexture(GL_TEXTURE_2D, content.texture_);
```

Then we need to specify some parameters:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

all parameters can be seen on this well documented tutorial (https://learnopengl.com/Getting-started/Textures).
Finally, we can now allocate the memory using this code line and unbind the texture:

```
    glTexImage2D(
            GL_TEXTURE_2D, 0, GL_RGBA32F, 1024, 1024, 0, GL_RGBA, GL_FLOAT, nullptr);
glBindTexture(GL_TEXTURE_2D, 0);
```

Parameters of the function are:

- GL_TEXTURE_2D: the target image. You can also manipulate 1D or 3D images.
- 0: the level of the texture. In computer graphic, texture can be store along there down scaled version. Here we only use the raw texture.
- GL_RGBA32F: The texture use 32 bits float to store the R, the G, the B and the alpha value.
- 1024: The size on the width.
- 1024: The size on the height.
- 0: This value must be 0 (https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml).
- GL_RGBA: The format of the pixel. Can also be GL_BGRA.
- GL_FLOAT: Pixel components are float.
- nullptr: a pointer on a CPU data buffer.

Because we specify nullptr on the last parameter, the texture will be initialize with null values. If you run you program, you can see the black texture.

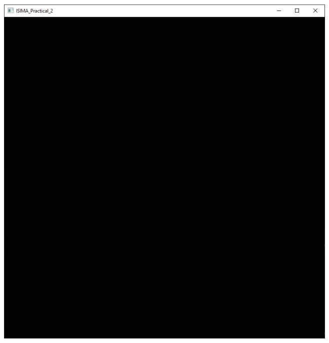

Figure 3: A black texture

3 The fractal compute shader

In the same way you did in the practical 1. Create a new compute shader (fractal_compute_shader) for the fractal generation and it's source named kComputeFractalSource. You need to create the program kernel_compute_fractal_ in the content struct, attach the shader and link the program. The binding of the texture can be done like this:

```
layout(rgba32f, binding=0) uniform image2D texture_fractal;
```

We can use the feature of a 2D layout for the organization of our threads:

```
layout(local_size_x = 8, local_size_y = 8, local_size_z = 1) in;
```

The value 8 was chosen to keep all shader invocations of a bloc inside a warp (8x8=64 threads). Finally, read and write on the texture can be done using imageLoad and imageStore. We can use the main function to show all the local invocation shader:

```
void main() {
        ivec2 coord = ivec2(gl_GlobalInvocationID.xy);
        imageStore(
                texture_fractal,
                coord,
                vec4(vec2(gl_LocalInvocationID.xy)/32.0, 0.0, 1.0));
}
```

On the CPU side we can use the program and the texture like on the first practical. The dispatch function will set 128 on Y too:

```
glUseProgram(content.kernel_compute_fractal_);
glBindTexture(GL_TEXTURE_2D, content.texture_);
glBindImageTexture(
  0, content.texture_, 0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RGBA32F);
glDispatchCompute(128, 128, 1);
glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
glBindImageTexture (0, 0, 0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RGBA32F);
glBindTexture(GL_TEXTURE_2D, 0);
glUseProgram(0);
```

4 The Mandelbrot set

The Mandelbrot set is the set of complex numbers c for which the function

$$f_c(z) = z^2 + c \tag{1}$$

does not diverge when iterated from $z = 0 + 0i$.
As a reminder complex number multiplication is:

$$z_1 * z_2 = (z_1.real * z_2.real - z_1.imag * z_2.imag) + (z_1.real * z_2.imag + z_1.imag * z_2.real) * i \tag{2}$$

The addition is:

$$z_1 + z_2 = (z_1.real + z_2.real) + (z_1.imag + z_2.imag) * i \tag{3}$$

And the modulus of a complex is his length or magnitude.
We can run the equation (1) while the result don't diverge. We consider a complex number is outside of the [-2; 2] window is a divergent value. *iterationMax* is the number max of "jump".

```
        float iteration_max = 42.0;
        float iteration = 0.0;
        while (length(z) < 2 && iteration <= iteration_max) {
                z = ComplexeMul(z, z) + c;
                ++iteration;
        }
```

Figure 4: Print local UV on a image using a compute shader

If iteration reach iteration_max that means the complex number chosen is part of the Mandelbrot set. We can use the iteration value as a color to show the fractal.

```
vec4 color = vec4(iteration/iteration_max, 0.0, 0.0, 1.0)
```

For the generation of the Mandelbrot set we will use $c = -1.5 - 0.5i$.

```
vec2 c = vec2(float(coord.x)/1024.0-1.5, float(coord.y)/1024.0-0.5);
```

Implement this algorithm to generate a fractal into the texture and show it:

5 The blur compute shader

On the last shader we write values into the texture. On this section we will read and write pixels. This operation can be done easily but the blur algorithm need neighborhood pixels. So sometime you will read pixel and sometime you will read modified pixel. To prevent that we can use the barrier key word.
Create another compute shader named kComputeBlurSource that compute the mean of the 9 neighborhood pixels by reading the input texture (imageLoad). Use the barrier(); function before write in the texture (imageStore). By applying this process 10 time you can get this result:

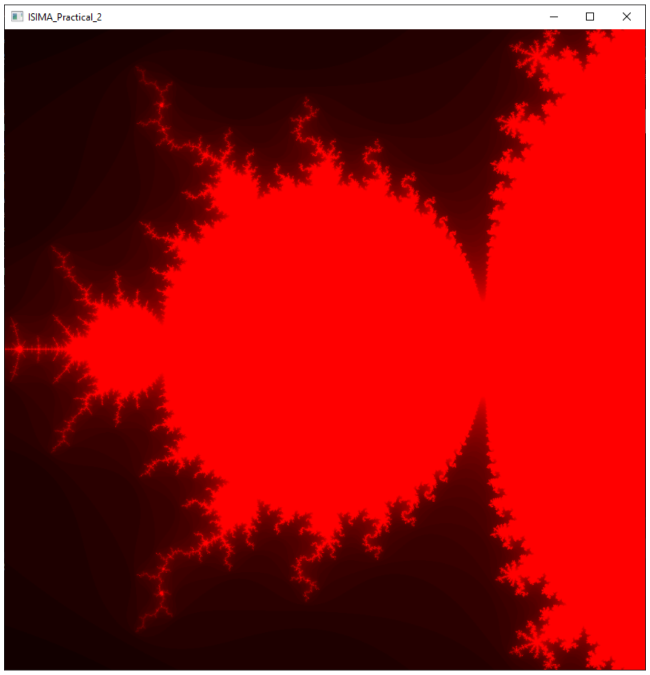

Figure 5: Mandelbrot set

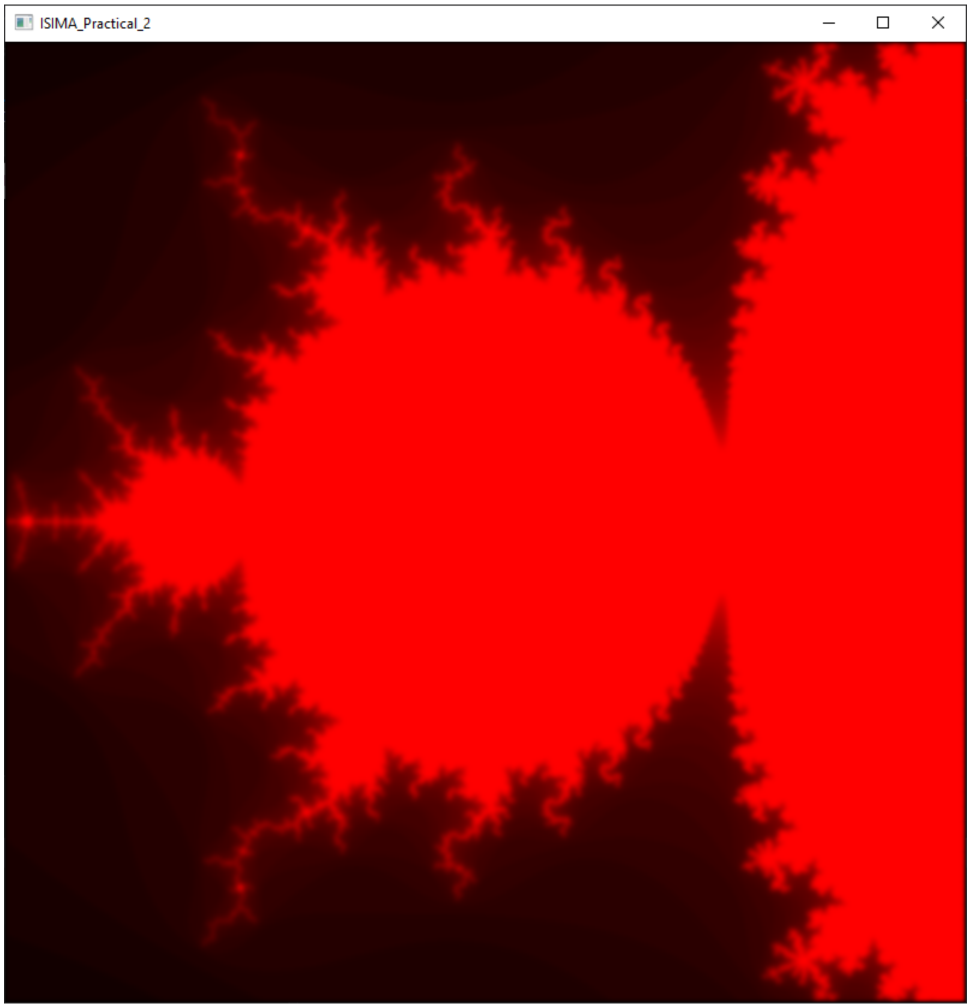

Figure 6: The blurred Mandelbrot set

6 Bonus

Implement the Julia Set using the same algorithm but with this parameter at start:

- z = x/width + (y/height)*i
- c = 0.292 + 0.015*i
- iteration_max = 400