

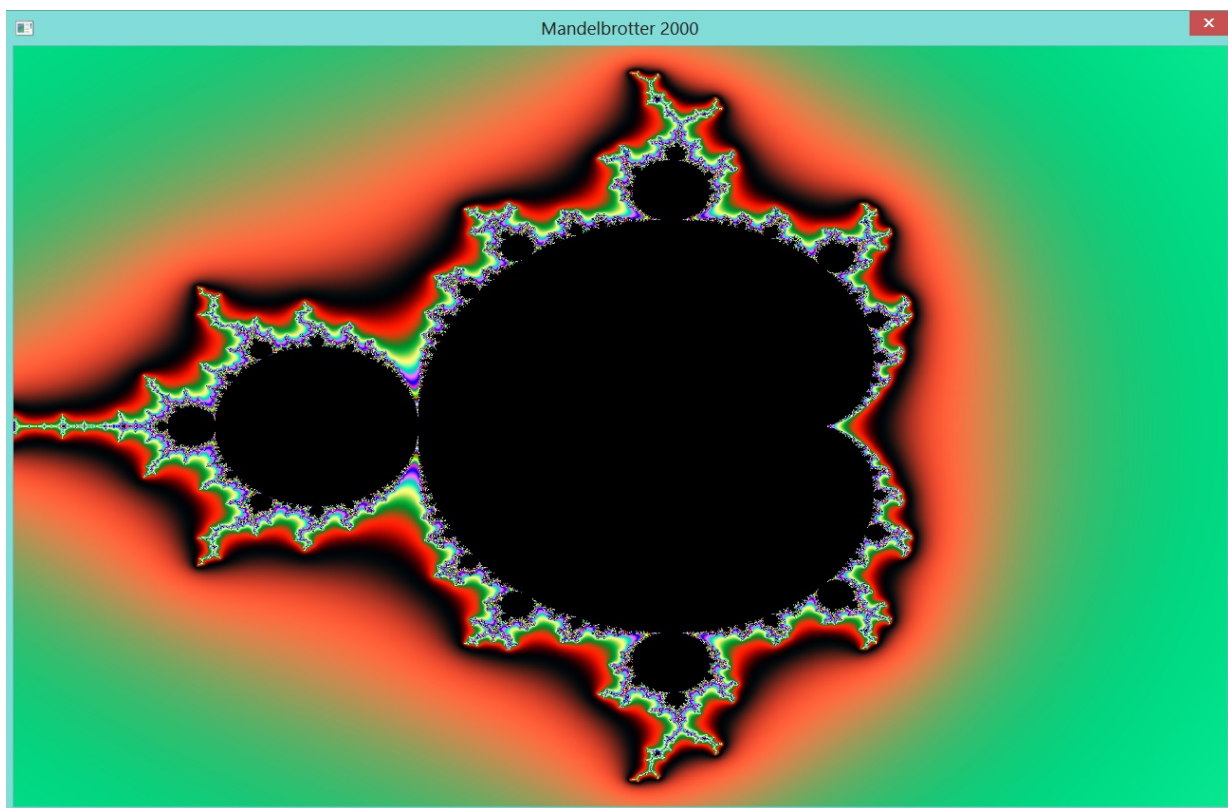
# Mandelbrot Practical Assignment

Advanced Graphics course

Student: Joeri van der Velden

Solid ID: 3928853

Rev. 2



## References used for this assignment:

- Microsoft MSDN knowledge base
- <http://www.directxtutorial.com>
- <http://www.rastertek.com/dx11tut04.html>
- All web sources cited in the assignment description

## Starting deficiencies

This assignment was my first time working with C++, DirectX and the Windows API. I knew some of the underlying concepts but I had not put this knowledge to use before.

## Application instructions / specs

- Use spacebar to toggle between Mandelbrot and Julia set drawing
- Use scrollwheel to zoom in
- Press and hold left mouse button to move the view
- Hold right mouse button and use scrollwheel to adjust time speed
- Use number keys (1 to 5) to change shader mode
- Press R to reset position and zoom

Instructions are also listed in the console window which should show upon launching the application. The Julia set constants move between -1 and 1 using a sinus function over time. By adjusting the time speed (right mouse + scrollwheel) you can view adjust or stop the time progression.

The sets are rendered up to 100 iterations. This is constant and thus I made a zoom limit that stops the user from further zooming in at a certain point.

The shaders are compiled with Shader Model 4.

## Additionally implemented features

The shader modes are the main attraction. These can be set using the number keys 1 through 5. They mainly play with the colours and other parameters in the pixel shader based on time, which produces some interesting results. They're not too spectacular from a technical viewpoint, but I personally found it more fun to explore. The Julia set is also implemented, you can toggle between Mandelbrot and Julia set drawing using the spacebar.

Shader mode 1: standard view. Colours are smoothed out and based upon time, with different offsets and factors per colour channel.

Shader mode 2: changes the amount of counted iterations once the loop terminates for colour picking, resulting in clearly visible borders over which the colour flows.

Shader mode 3: changes amount of counted iterations before colour picking too. Looks rather ominous.

Shader mode 4: changes the red and green colour channels based on position and time, giving the variety of hues a more soothing look.

Shader mode 5: the real and imaginary values are scaled every loop iteration, which is very visible in the Mandelbrot set. Change time speed to examine details at a specific moment.

## **Code structure description**

The MandelBrotWin32.cpp is the entry point of the application, at the WinMain function. Here it creates instances of the Window and Renderer classes, and then goes into the program loop until termination.

The Window instance creates the window using the Windows API and holds the window handle. This handle is passed to the Renderer instance, which initializes the Direct3D by creating the swap chain, various buffers, render target and viewport.

Next the Renderer initializes the pipeline by loading the compiled VertexShader and PixelShader shaders from the .cso files, while creating the layout and data buffers to pass information to the shaders.

Finally, the Renderer creates a screen-filling quad by passing two triangles to the vertex buffer.

In the main program loop the current time is updated, shader data is updated and a frame is rendered. It also executes the Step() function every 1/60 of a second, to update movement and zoom transitions.

## **Performance bottleneck notes**

Everything runs smoothly on my laptop (using an Intel HD4000 integrated graphics card, supports DX11), but I'm sure some parts of my code could be made more efficient.

Firstly I haven't fully wrapped my head around the inner workings of HLSL yet. My shader code could possibly be made more efficient for the GPU to handle, or some of the computational tasks in the pixel shader could be moved to the vertex shader.

Secondly, I update the constant buffers of both the VertexShader and PixelShader every frame, since time and the transitioning zoom level require constant updating. I wonder if this has any noticeable performance impact or if it can be made more efficient.

Lastly, the pixel shader calculates 100 iterations of the mandelbrot loop for every pixel in it. This could be made more efficient by storing results from the previous frame and iteratively calculating the fractal. This would probably mess with my current application features though.

## **Other code comments / future work / stuff I didn't quite have time for**

Currently my input handling is done in the main file, which produces some clutter. It could possibly be moved to a class dedicated to handling input.

Same goes for movement handling. A camera / view class could be made that handles a transformation matrix to handle zooming and view translation.