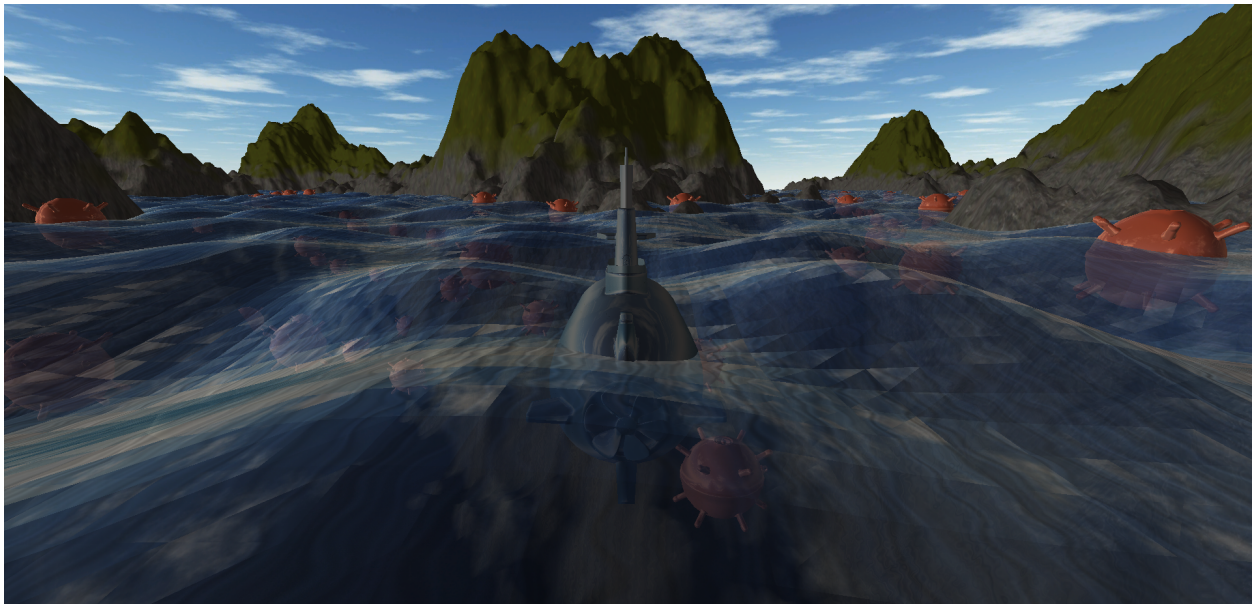

Coursework Report

CMP505 - Advanced Procedural Methods

Alessandro Arsenio - 2003816

Summary of the Application

The application developed for the module CMP505 - Advanced Procedural Methods, is a game set in a seascape where the player has to drive a submarine and destroy with its missiles all the watermines placed underwater, being careful to not touch them.

The game was developed starting from what was built in the module CMP502. The old scene was deleted, but some features were kept, such as:

- The skybox which surrounded the scene
- The reflection shader used to create the skybox reflection on the objects
- The input system used to move the camera
- Some previous textures and models such as the submarine

Several features were then added to create the new scene:

- A procedural terrain which use Fractal Brownian Motion on CPU to recreate the model of the mountains and the seafloor present in the scene
- A water mesh which use Simplex Noise, this time on GPU, to recreates the waves movement
- A post process effect which use Gaussian Blur and Simplex Noise to recreate the effect of the camera being underwater
- An improved hierarchy of classes to handle more efficiently the different objects in the code
- Collision detection between the player, the missiles and the watermines

Controls

The submarine can be controlled by using **A,S,D,W** to move it on the XZ plane, and **Q,E** to move it up or down on the Y axis. The **mouse** can be used to rotate the submarine horizontally, and its **left button** can be used to fire.

Procedural Terrain

A terrain class was already given in the labs at the beginning of the module. This class could create a mesh divided in a chosen amount of triangles, to reproduce a flat surface. The class could also let the user modify the height of each vertex inside the mesh, and to calculate every vertex normal after changing the shape.

To transform the terrain from a flat surface to a mountainous landscape, it was necessary to change its height map using **Simplex Noise** and **Fractal Brownian Motion**.

Simplex Noise

Simplex Noise is an improved version of Perlin Noise, an algorithm developed by Ken Perlin, which is capable of generating pseudo random values in single or multidimensional space, which variate one another in a smooth and natural way.

Values from a permutation table are assigned to every vertex in the square grid in the case of a 2D space, or in a cube grid in the case of 3D space. The perlin Noise is then capable, by having some space coordinates as input, to return a value which is a gradient measured among all the values from the nearest vertices in the grid.

Simplex Noise, developed by Ken Perlin himself, uses instead a simplicial grid, made of triangles if we are in two dimensions, or pyramids if instead there are three dimensions.

Simplex noise was developed to overcome some limitations given by Perlin noise, and offer several computational advantages, making this technique a more valid solution (an in depth explanation and implementation in Java of Perlin and Simplex Noise is given by Stefan Gustavson in his paper which can be found at the following link: https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf).

Implementation

A C++ implementation of the Gustavson Simplex Noise was made by Sebastien Rombauts and can be found at the following link: https://github.com/SRombauts/SimplexNoise. The implementation also includes Fractal Brownian Motion (fbm), a technique which scales and sums recursively octaves made with Simplex Noise, to produce a more complex structure.

After adapting this implementation in the project, it was created a function which loops through every vertex in the terrain, calculates its fbm value using the vertex coordinates as parameters, and assigns the result to the vertex's y component.

The noise coordinates were scaled down properly to adapt the noise to the size of the terrain, and the returned value was multiplied to a scale factor to obtain higher mountain peaks (figure 1).

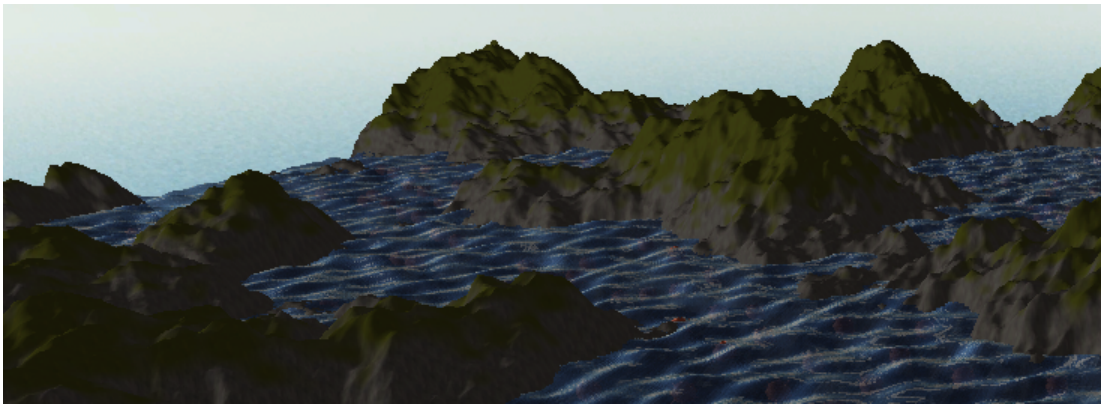

Figure 1. Mountains produced with fbm

The Water

The second goal was to reproduce the effect of the water moving and creating waves.

Since the peaks created on a 2D surface using Simplex Noise are smooth and sinuous if the returned value is multiplied for a small scale factor, this technique was considered good enough to replicate the shape of a wave. Furthermore, when using Simplex Noise in the third dimension, it is possible to change the x, y or z value to obtain different layers of noise, which are however connected one another as fotograms of a moving liquid surface.

Therefore, the same noise function used for the terrain was also used to change the height map of a second flat terrain, this time on every Update call. The coordinates used for the Noise were also incremented over time using a delta time value, to obtain every time a different result. The terrain was consequently modifying its shape over time, creating realistic waves which were moving into a specific direction.

However, the game's framerate had a dramatic decrease, since the application was calculating every frame the noise value of every vertex of the mesh. It was therefore understood that the noise computation should have been moved on the GPU, where instead such operations can be done in parallel.

The Water Vertex Shader

Moving the noise computation on a shader, it could have been possible to apply the operation described in the previous paragraph to every vertex of the mesh simultaneously.

A GLSL implementation of Simplex Noise was made by Ian McEwan and can be found at the following link: https://github.com/ashima/webgl-noise.

As the author of the code says, this version the Simplex Noise does not use a permutation table stored in a texture and then passed as input to the shader. Doing so, this technique is "*more scalable to massive parallelism and much more convenient to use*". On the other hand it is "*not quite as fast as texture-based implementations*"

The Noise function was then converted from **GLSL** to **HLSL**, and integrated into a vertex shader used to render the water mesh. The noise was calculated passing the x y and z coordinate of the vertex and the result was applied to its y component. However, before this operation, the y and z components of the vertex were multiplied for a "time" value passed as input into the shader. This value represents the time passed from start of the game. Doing so, it was possible to obtain a moving coordinate inside the 3D noise, and therefore to modify the surface of the mesh creating waves, this time without impacting the performance of the game (the code can be found in the file *water_vs.hlsl*).

However, a problem encountered at this stage was that even if the shape of the mesh was changing accordingly, the normals of its vertices were remaining the same. It was therefore impossible to see the waves shape, since the water color was identical in every point of the surface.

When rendering the terrain on the CPU, after modifying the height map, its class has to loop through the entire surface, calculate each face normal, and then modify every vertex normal by calculating the average among the face normals that vertex touches.

However, on the vertex shader, the only information available for the water mesh is about a single vertex. Therefore, after modifying the vertex's y position with the Simplex Noise, there were no solutions to recalculate its normal on the vertex shader itself.

The Geometry Shader

A solution to the normals problem was found in the **geometry shader**. This shader, unlike the vertex shader, can have as input vertices of an entire primitive. This primitive can be a single **point**, a **line**, a **line with its adjacent vertices**, a **triangle** or a **triangle with its adjacent vertices**, all belonging to the original mesh that has to be rendered. (more information on the geometry shader can be found in the DirectX's official website at the link: https://docs.microsoft.com/en-us/windows/win32/direct3d11/geometry-shader-stage)

Before passing any mesh to the geometry shader, it is necessary to specify what primitive topology has to be used among the ones listed above, and to appropriately create the list of vertices and indexes for the vertex and index buffer, which are used to draw the mesh (as can be seen in figure 2)

```
// Set the type of primitive that should be rendered from this vertex buffer, in this case triangles.
deviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

Figure 2. How to define the primitive used to draw the mesh (Terrain.cpp)

It was therefore understood how using a specific primitive, such as the **triangle strip with adjacency**, which interprets the vertex data as a list of connected triangles with their neighboring vertices, it could have been possible to recalculate the normals of every vertex modified in the vertex shader.

A possible method could have been to calculate the normals of the surrounding faces of a triangle using the adjacent vertices and then to calculate the normal of the vertex positioned in between those faces. Even if in a single call of the geometry shader there are not enough information to see all the contributions of every face who touches the vertices of the triangle, it could have been possible to complete the computation and sum the missing face normals in the next invocations of the shader, where the same vertices are represented, this time in a different position (as visible in figure 3)

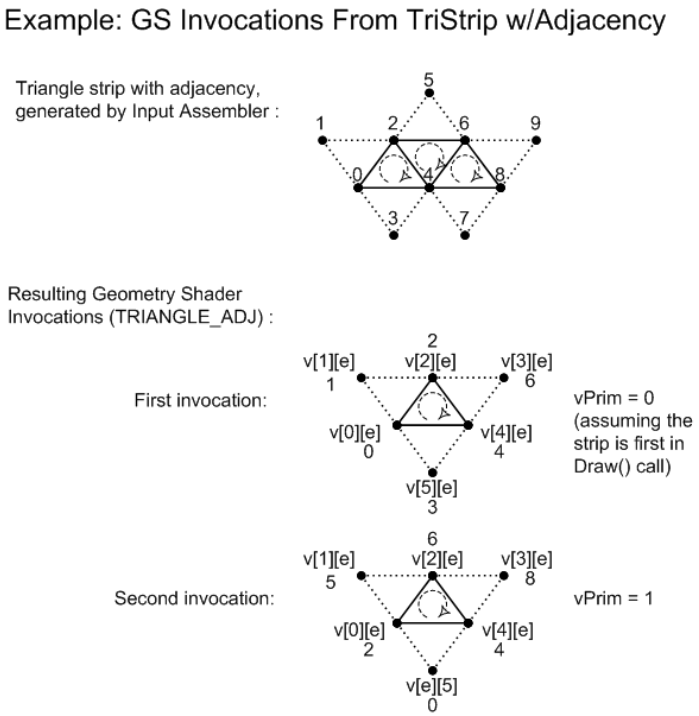

Figure 3. Geometry shader iInvocation of the primitive: triangle strip with adjacency (from: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-geometry-shader)

However, with the time left for the project development, it was not possible to understand how to modify the vertex and index buffer used in the Terrain class and to let the Geometry shader interpret the primitives not as single triangles, but as triangles with adjacency.

A discussion regarding the same problem was found at the following link:
https://www.gamedev.net/forums/topic/499275-good-news-read-please-how-to-generate-adjacency/?page=1

Also, a library which could hypothetically calculate the adiancey of a given mesh can be found at the following link:
https://github.com/microsoft/DirectXMesh/blob/master/DirectXMesh/DirectXMeshAdjacency.cpp

Considering the lack of time, it was decided to let the geometry shader use only the information given by the primitive "triangle list" which was already in use in Terrain class, and to calculate the normal of a single triangle face to then apply the result to all of its vertices.

This is an incomplete result, which however started to show the difference in brightness given by the inclination of the water in a specific point in relation with the position of the light source (figure 4)

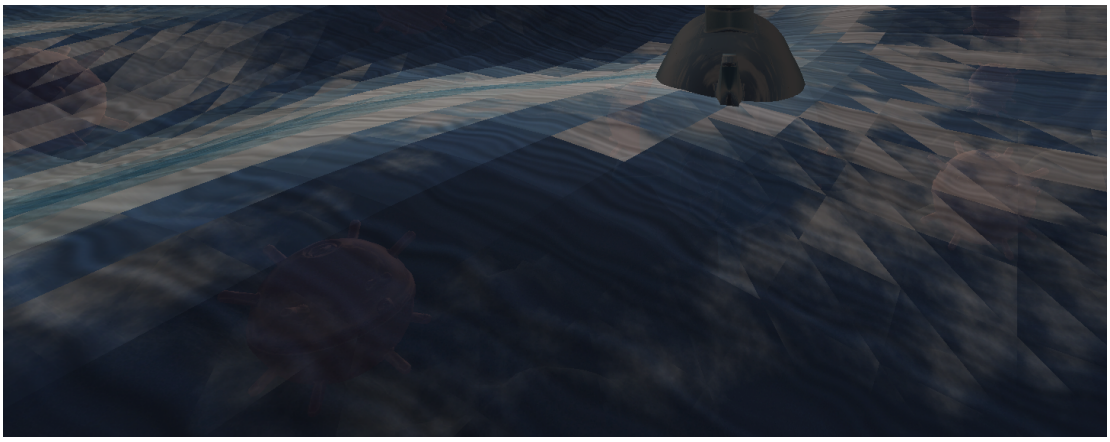

Figure 4. Waves with different brightness

It was also possible to apply the reflection technique developed in the previous DirectX module, to reflect the skybox on the surface of the water.

Simplex Noise on Pixel Shader

The HLSL version of the Simplex Noise was also applied on the pixel shader used for the water and the terrain mesh, to change the brightness of each pixel and reproduce two effects:

- On the water, to recreate little sea ridges moving in the same direction of the waves (figure 4)
- On the terrain, to reproduce the shadow created from the sea ridges on the seafloor (figure 5)

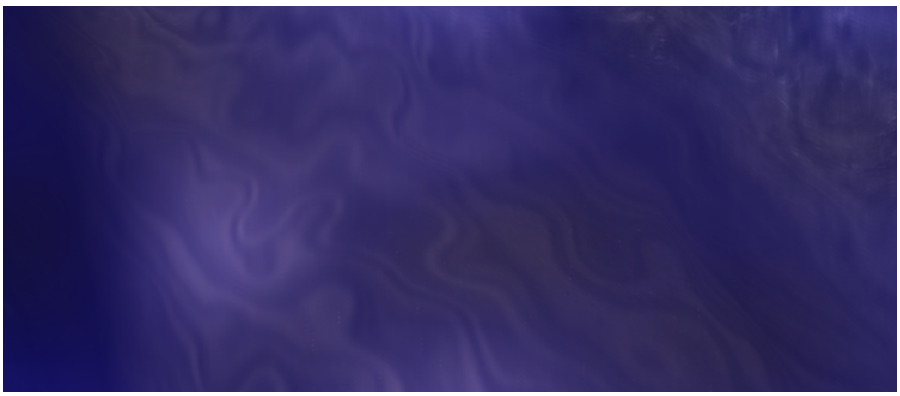

Figure 5. Sea ridges shadow on the seafloor

Post Processing

A post processing effect was added to the game to recreate the effect of the camera being underwater. The implementation followed part of the steps given in the DirectX official tutorial on implementing a post processing effect, which can be found at the link: https://github.com/microsoft/DirectXTK/wiki/Writing-custom-shaders.

Using this guide, it was created a pixel shader which takes as input a texture, and adds a Gaussian Blur effect to it. The input texture is created by rendering the scene on a different RenderTargetView, using the RenderTexture class given in the engine. The same class is used to render the first call of the post processing effect, which adds an horizontal Gaussian Blur effect. For the vertical Gaussian Blur effect, the shader takes as input the texture produced by the horizontal Gaussian Blur while the render target is set back to the original back buffer, so that the effect is drawn on the screen (the *PostProcessing()* function can be found in the file *Game.cpp*).

Together with the Gaussian Blur effect, the Gaussian Pixel Shader adds a blu color to the scene and uses the HLSL Simplex Noise to add some noise to the final result (figure 6).

Figure 6. Underwater post processing effect (GaussianBlur.hlsl)

Terrain Texturing

To texture the terrain, it was used a set of three different textures (sand, rock, grass) and a Terrain Pixel Shader which, using two defined boundaries, blend between the three textures to color the terrain differently depending on the height of the pixel. To then randomize the boundaries, so that the change between two textures wouldn't have been defined by a straight line, the HLSL Simplex Noise was also applied to the formula defining both of the limits. Each texture is also faded gradually while going close to the boundaries position to obtain a smooth transition between textures. (the implementation can be seen in the file *terrain_ps.hlsl*)

Code Structure

The application's code was restructured to organize more efficiently the different objects in the scene, to reduce redundant lines of code and to facilitate the rendering of the environment.

A **GameObject** class was created, which stores all the common information used by every object in the scene. Some of them are:

- **Position**: can be global or local, depending if the player has a parent Object assigned
- **Orientation**: the orientation also define the forward vector used to move an object on its relative coordinates
- **Tag**: used to recognise the object's type

- **Mesh**: mesh assigned to the object
- **Shader:** the shader used to render the mesh
- **Texture**: used by the assigned shader
- **Collider**: a sphere collider to detect collisions
- **Parent object**: define if the player is binded to another object. If so, its position is relative to the position of the parent

The GameObject class have also several methods, such as the ones necessary to set and get the above variables, plus the:

- **Update** method: used to update the object's coordinates
- **Render** method: used to render the object in the scene

It was then created a specific subclass of the GameObject class for every object in the scene which necessite to use its properties (figure 7). These subclasses then override some of the GameObject Class's methods to change their behaviour. These classes are:

- **PlayerObject Class**: overrides the Update method, adding input detection to move the player around the scene
- **CameraObject Class**: was initially used to override the Update method and move it around the scene. It was then decided to attach the camera to the player and create a third person movement system.
- **TerrainObject Class**: overrides the Render method to render its mesh differently, using the water shader if set as water, or a terrain shader if set as terrain. It also stores multiple textures to texture the terrain.
- **Missile Class**: overrides the Update function, to move on its forward vector once it is initialised in the scene. It also has a life component, which determines how long the object can remain alive in the scene before being deleted.
- **Watermine Class**: overrides the Update function, to add a simple movement up and down and a rotation around its Y axis.

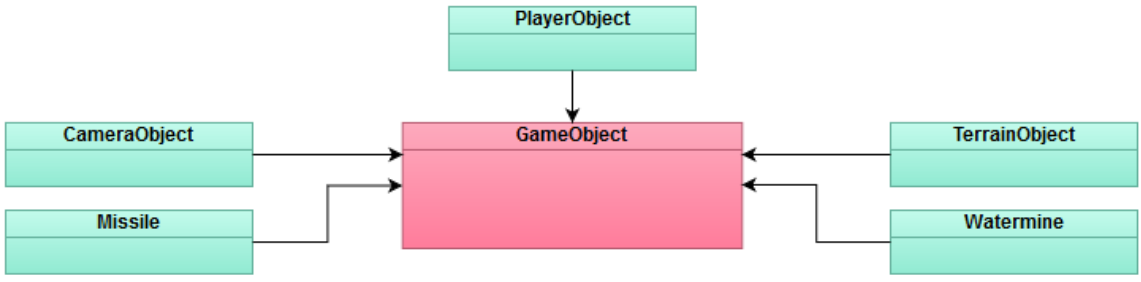

Figure 7. Hierarchy of the game objects in the scene

A *std::vector<GameObject>* variable was then defined in the Game.h file, and every object created for the game was stored inside it. This way, it was possible to update and render each object in the scene looping through the vector instead of defining several matrices and rendering each object individually.

The application's code is therefore more organise, concise, and also easy to read. However, a better structure should have been created for the different shader classes used to render the game objects. The code in these classes can be a little redundant and not easy to understand. An hypothetical solution could have been to create a hierarchy also for the different shader classes, creating a default class which can then be inherited by other subclasses who need to define their own properties.

Conclusion

After working on this project, it is clear how important procedural methods are in Computer Graphics. It is satisfying to see how from mathematical functions it is possible to generate natural landscapes and how after understanding the concepts behind, these techniques can be a robust and easy alternative to modelling or animating an object manually. After this project, it is also more clear the strength of shader programming, the performance improvements which can be achieved when computing on a GPU and the vastity of effects which can be generated using shaders.

References

Gustavson, S. (2005). Simplex noise demystified. [online] . Available at: https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.

Rombauts, S. (2021). SRombauts/SimplexNoise. [online] GitHub. Available at: https://github.com/SRombauts/SimplexNoise.

GitHub. (2021). ashima/webgl-noise. [online] Available at: https://github.com/ashima/webgl-noise.

Steve Whims (n.d.). Geometry Shader Stage - Win32 apps. [online] docs.microsoft.com. Available at: https://docs.microsoft.com/en-us/windows/win32/direct3d11/geometry-shader-stage.

Steve Whims (n.d.). Geometry-Shader Object - Win32 apps. [online] docs.microsoft.com. Available at: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-geometry-shader.

GameDev.net. (n.d.). [GOOD NEWS! READ PLEASE] How to generate adjacency? [online] Available at: https://www.gamedev.net/forums/topic/499275-good-news-read-please-how-to-generate-adjacency/?page=1.

GitHub. (n.d.). microsoft/DirectXMesh. [online] Available at: https://github.com/microsoft/DirectXMesh/blob/master/DirectXMesh/DirectXMeshAdjacency.cpp.

GitHub. (n.d.). microsoft/DirectXTK. [online] Available at: https://github.com/microsoft/DirectXTK/wiki/Writing-custom-shaders.