Procedural waves and geometry

TNM084 - Project report

Rasmus Hogslätt, rasho692

January 4, 2023

Contents

1 Introduction

Procedural generation of geometry is a useful tool for generating complex scenes in computer graphics, without having to explicitly create the geometry in a modeling tools such as *Blender* or *Maya*. These methods can thus save time and resources when creating geometry assets for various applications. This report explores how procedural methods can be used to generate water waves and simple geometry floating atop those waves utilizing graphics cards. More explicitly, the essential theory to implement such generations of geometry will be presented and discussed.

1.1 Problem description

The project implemented and its constraints are described by the following bullet points:

1. Generate procedural waves
   - Flat grid is created on CPU
   - Wave displacement on GPU
2. Generate boxes floating on top of waves
   - Single vertices created on CPU
   - Vertex positions used by GPU to generate boxes
3. Allow interactive modification of wave properties
   - Using *ImGui*, a GUI library for C++
4. Render shaded geometry

2 Background

2.1 Description of waves

Approximation of waves is commonly done by utilizing the nature of trigonometric functions. A common sine wave taking time as an input yields a rather boring and uniform wave pattern as seen in Figure 1.

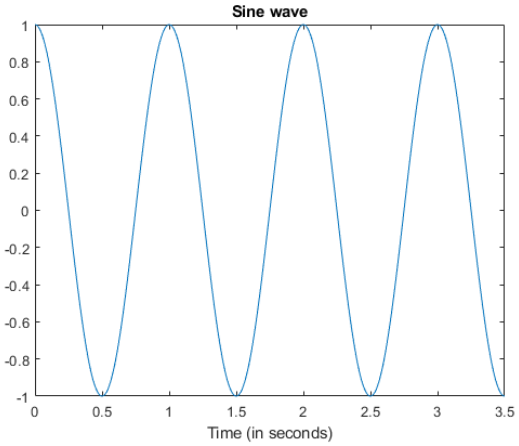

Figure 1: Sine wave over time.

Real waves however, exhibit more complex motions, and thus, Gerstner, a German engineer, proposed a more accurate approximation, called *Gerstner waves* or *Trochoidal waves*. It makes use of trochoidal patterns, which can is illustrated in Figure 2

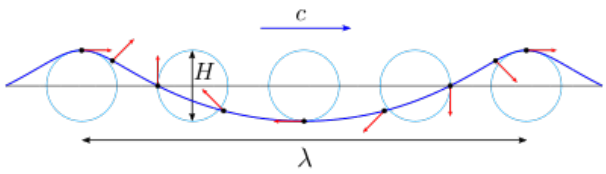

Figure 2: Wave surface is traced along trochoidal pattern.

Simple Gerstner waves can be computed by converting a sine wave into a circle to offset the starting particle's position as in equation 1, 2 and 3

$$x_{new} = x_{old} + a * cos(f) \tag{1}$$

$$y_{new} = a * sin(f) \tag{2}$$

$$z_{new} = z_{old} + a * cos(f) \tag{3}$$

where a is the amplitude of the wave and f is its frequency. Furthermore, several of these waves with varying properties can be computed and summed together to create more complex waves.

2.2 Fractal Brownian motion

Waves, such as in Figure 1, have a set of properties. These can be derived to an amplitude and frequency. Changing the amplitude adjusts the height of the wave, whereas the frequency adjusts how often the wave is repeating. Knowing this, more complex waves can be achieved by letting the height of the final wave depend on several subwaves. This introduces the concept of octaves, which determines how many subwaves will make up the final wave. For each octave, the amplitude is scaled by some noise depending on a set of coordinates. In this case, the noise was a 2D Perlin noise function implemented by Stefan Gustavson[1]. These coordinates increase by a factor of two for each octave and the amplitude is halved. Thus, higher octaves increase computational cost, but can achieve higher levels of detail than lower octaves. This generates a fractal Brownian motion[3] which can be further applied to Gerstner waves to add more detail, as seen in Figure 3

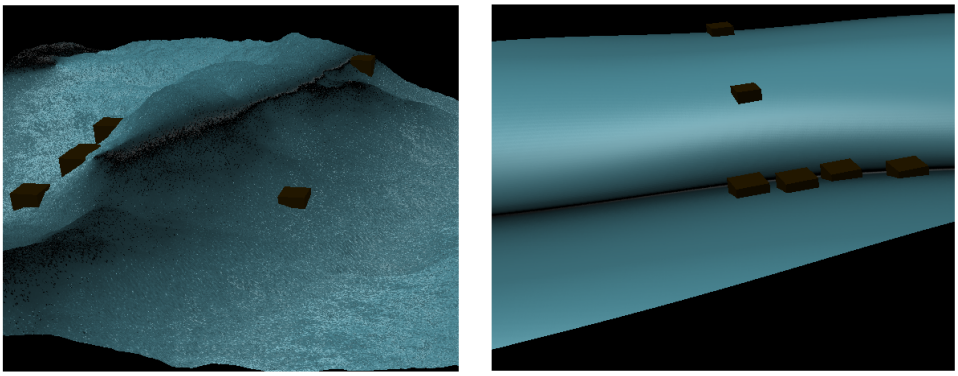

Figure 3: Same scene with Fractal Brownian motion with 12 octaves on left and without on right.

3 Implementation

3.1 Waves

Initially, a flat mesh of triangles were generated on the CPU and passed to the GPU using OpenGL's API. At this point, only indices and vertex positions were passed as normals were to be recalculated on the GPU anyway. The entire generation of waves were handled in one vertex shader and fragment shader pass respectively. All of the relevant uniform variables mentioned ahead were being sent from the CPU side and adjusted using ImGui in real time.

3.1.1 Vertex shader - Waves

The vertex shader handled most of the wave generation. To make things simpler, one function called *displace* was implemented and its purpose was to take a single vertex position and time in order to calculate its new position. The *displace* function calculated this by first applying a *fractal Brownian motion* function and then calling a *Gerstner* function four times to simulate four different wave motions. In other words, a noise was applied initially and further wave offsets were accumulated. To make the various waves more interesting,the user could adjust each wave's direction, steepness and wavelength from the GUI. Thus, different wave motions could be explored interactively.
In order to calculate normals, three vertices were sought after. Thus, the incoming vertex position were first offset three times in the xz-plane in different directions which were scaled by a very small *epsilon* value. Each of the new vertices were displaced again using *displace* and vectors were spanned along two of their edges. The cross product was then used to calculate the normal vector for the central position. This might not be the truly correct normal for the vertex at the center, but given a small enough *epsilon* value, the approximation was good enough. Further, the actual vertex was also displaced and passed on through OpenGL's rendering pipeline along with the normal.

3.1.2 Fragment shader - Waves

The fragment shader implemented the traditional Phong illumination model using the normal passed from the vertex shader. Positioning and colors were handled using ImGui. The user can assign two uniform colors, which were blended depending on the height of the wave.

3.2 Boxes

On the CPU the boxes were constructed as single individual vertices, randomly placed within the bounds of the flat mesh that had already been generated. Thus, at this point, they could only be rendered as singular points. The process of turning them into boxes and positioning them on top of the waves was done by using a vertex-, geometry and fragment shader.

3.2.1 Vertex shader - Boxes

The exact same functions with the same parameters as for the waves were applied to the incoming vertex. In this case, it would not be possible to calculate any normals, so *displce* was only called once in this instance. The box did not yet exist, thus no normals could be calculated yet. The new vertex's position was passed to the geometry shader.

3.2.2 Geometry shader - Boxes

The task of the geometry shader was to construct a box as a triangle strip and send normals for each face of the box to the fragment shader. This meant that a single point was received from the vertex shader and a triangle strip of 24 vertices were being passed out from the shader.
Initially, eight corner points of the box were calculated around the vertex being received from the vertex shader. This meant that a face of the cube could be represented using four of these corner points to construct a triangle strip. As a box has six faces, six primitives and normals had to be calculated. This was done by constructing one triangle strip at a time with four corners, using correct winding order, and then ending that strip using GLSL's function *EndPrimitive*. Before ending each primitive a normal vector declared as an *out* variable in GLSL, was first calculated. This meant that whatever the normal's value was at the time of ending the primitive, that normal was sent to the fragment shader. The normal calculation was done, once again, using the triangulation method with three of the face's points.

3.2.3 Fragment shader - Boxes

The fragment shader for the boxes was now implemented the same way the one for the waves was, with the only difference being that some properties now affected e.g. shininess of the box instead of the waves.

4 Results

The results shown in Figures 4, 5, 6, 7, 8, 9 and 10 were all captured running on Windows 11 with an *Intel i7 12700* processor and a *NVIDIA RTX 3060 TI 8GB* graphics card. In all cases, the maximum monitor refresh rate, 165 Hz, was achieved. Increasing the fractal Brownian motion octaves to around 50 caused framerate to drop to around 30 frames per second.

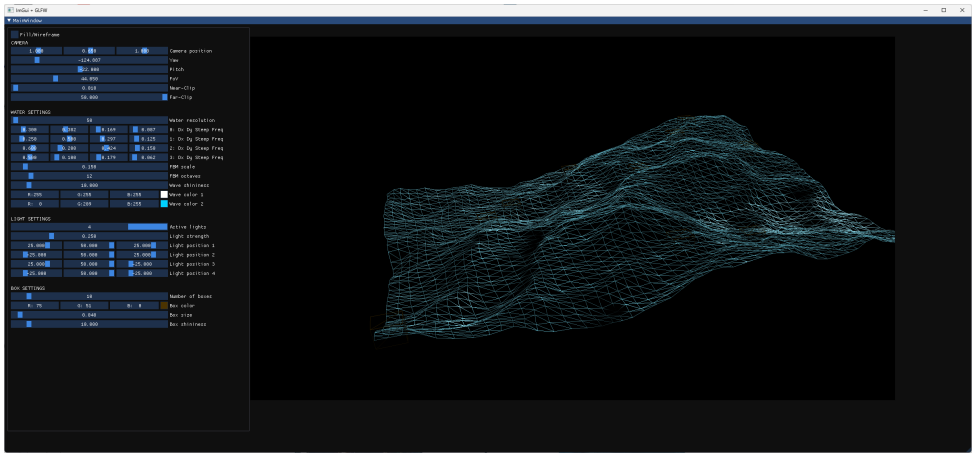

Figure 4: Wire frame render of 50x50 mesh with 12 FBM octaves.

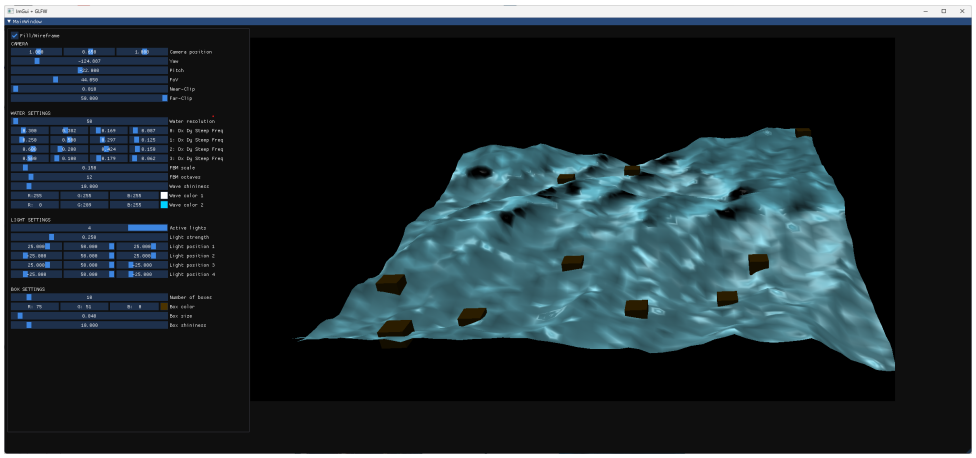

Figure 5: Default render of 50x50 mesh with 12 FBM octaves and 10 boxes.

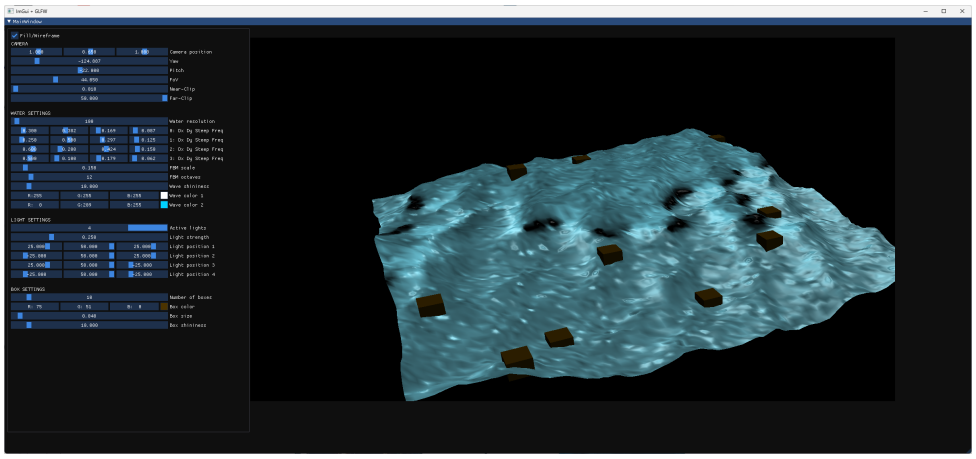

Figure 6: Default render of 100x100 mesh with 12 FBM octaves and 10 boxes.

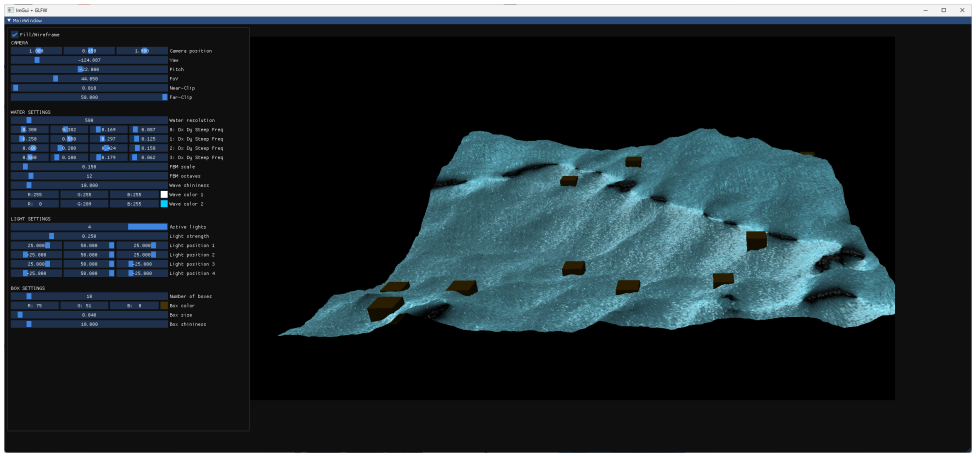

Figure 7: Default render of 500x500 mesh with 12 FBM octaves and 10 boxes.

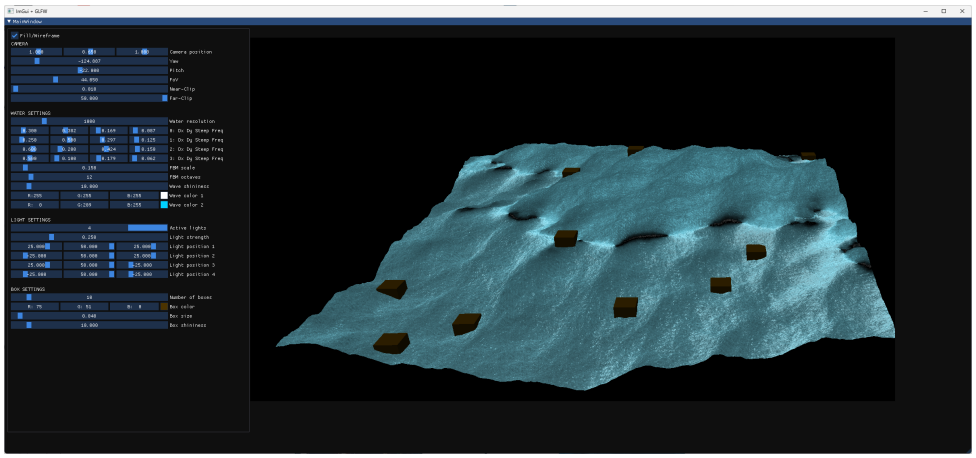

Figure 8: Default render of 1000x1000 mesh with 12 FBM octaves and 10 boxes.

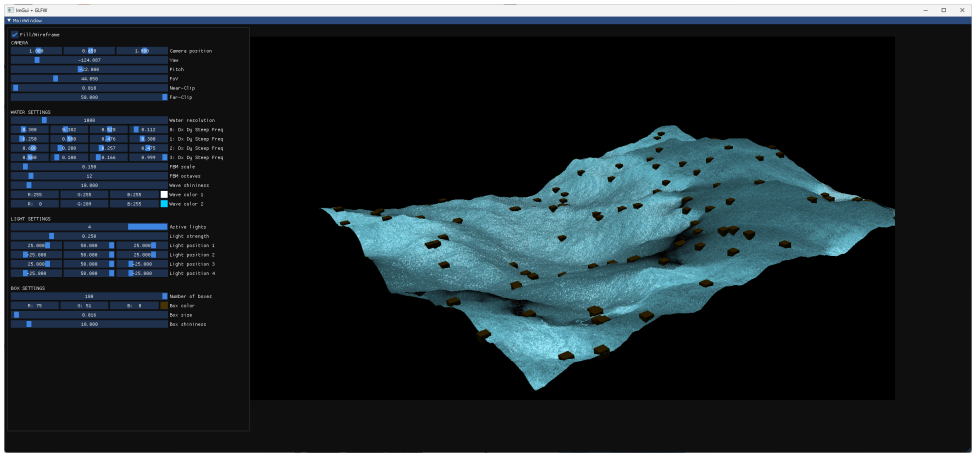

Figure 9: Default render of 1000x1000 mesh with 12 FBM octaves and 100 boxes.

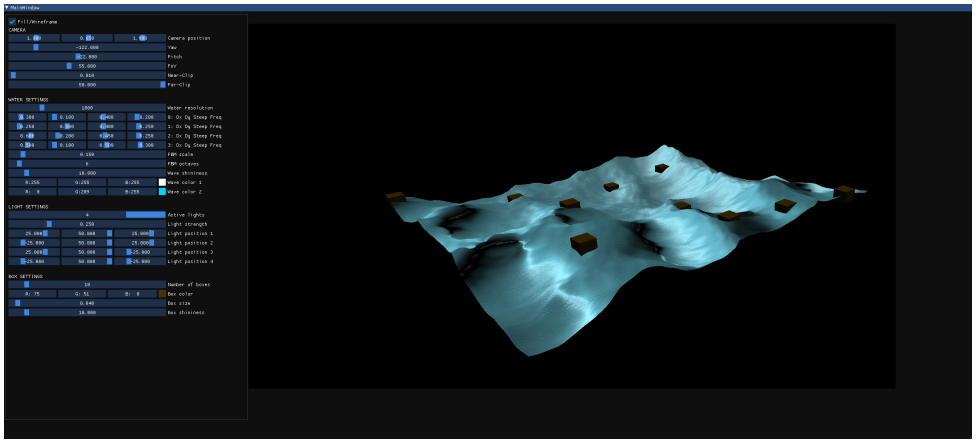

Figure 10: Default render of 1000x1000 mesh with 4 FBM octaves and 10 boxes.

5 Discussion

The resulting figures shows that the goal was achieved, although still images do not make much sense of the interactivity that the utility of the parallel GPU processing provides. Thus, if further investigation is wanted, it would be recommended to run the project locally by downloading the code from Github[2].

Using Gerstner waves seemed to yield interesting waves that to an untrained eye, may seem physically correct despite that not being the case. Thus, procedurally generating Gerstner waves is prove to be a useful option for applications where performance is more critical than accuracy, for example in video games. Physically correct waves would require particle simulation, which entails a significantly higher computational cost. However, these simulations might also be possible to do efficently on the GPU to achieve higher performance.

A drawback with Gerstner waves is that they only describe the displacement of a surface and not the entire body of water being simulated. Although it is possible to generate several layers of Gerstner waves at various depths, for applications where simulation of a body of water is required, Gerstner waves don't seem to provide much benefit. An example of such use case would be when water displacement from objects moving through the medium is sought after.

As for the boxes rendered, geometry shaders proved useful for generating more complex mesh from very little data. However, the implementation utilized geometry shaders, which has the drawback of invoking an extra step in the OpenGL shader pipeline. Therefore, it would have been interesting to further explore and compare performance using other methods such as instancing. Instancing would allow for one drawcall to render several primitives at once. Another interesting thing to explore would be to rotate the boxes along the movement of the waves. In the current implementation, all of the boxes are simply following the altitude of the waves, i.e. they are not rotated along the waves normals.

The normals were computed using the triangulation method for both the waves and the boxes. If no fractal Brownian motion would have been applied, the normals could have been computed mathematically using the gradient of the wave function. That would mean that three extra points would need to be computed just to find the normal. Perhaps, adding more individual waves with various parameters could have yielded similar results to using noise with higher performance. That could also be explored further.

6 Conclusion

The graphics card is a very powerful processing unit due to its parallel architecture. It is however rather different than programming on the CPU in a non-parallel manner.

Noise has proved to be a very powerful tool to generate complex geometry that doesn't need to be explicitly generated and that doesn't have constraints of resolution for textures etc. This is a tool that I will bring with me into further projects.

References

[1] Stefan Gustavson. *ClassicNoise2D*. https://github.com/stegu/webgl-noise/blob/master/src/classicnoise2D.glsl. 2021. (accessed: 29.12.2022).

[2] Rasmus Hogslätt. *Procedural Waves*. https://github.com/RasmusHogslatt/Procedural-waves. 2023. (accessed: 04.01.2023).

[3] Jen Lowe Patricio Gonzalez Vivo. *The book of shaders*. 2015. URL: https://thebookofshaders.com/13/. (accessed: 29.12.2022).