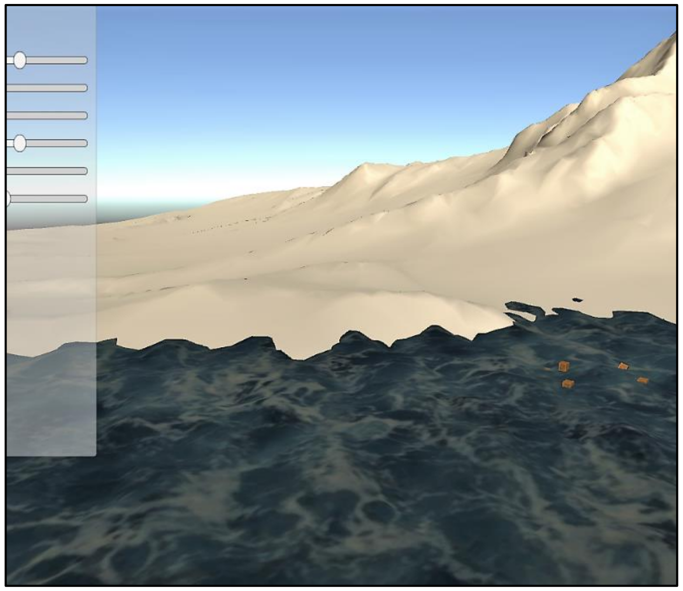
Advanced Technologies: Water Simulation

Dudley Dawes [15017153]
Department of Computer Science and Creative Technologies
University of the West of England
Coldharbour Lane
Bristol, UK
dudley2.dawes@live.uwe.ac.uk



Abstract
The aim of this project is to develop a realistic ocean water simulation, optimised to run as a plugin for Unity game. The main areas of research include; buoyancy, fluid dynamics and implementations of water simulations created developers in a professional environment.

Author Keywords
Gerstner wave; Vertex shader; Unity; Fluid Dynamics;

Introduction
The Ocean is an incredibly intricate, environmental phenomenon, produced through a combination of tidal flows, wind and varying densities of water. On a molecular level, water has an erratic and diverse behaviour, changed by differing temperatures and depth pressure.

Whether it's the stylised Water Simulations of, *The Legend of Zelda: Wind Waker* (Nintendo EAD) or the realistic simulations of *Assassins Creed IV: Black Flag* (Ubisoft). Developers have always found ways of utilising hardware to render aesthetically satisfying water simulations. The aim of this project is to research techniques used by developers in professional environments and understand how the same processes could be applied on a small scale application.

The video games industry is known for having tight briefs and constraints. Therefore, researching how simulations effected by such pressures, the type of mathematical equations utilised for the motion of water, will help determine a best solution.

Background Research

When researching the development of a water simulation, it is clear that the technique used represents the desired outcome of the simulation. Water is extremely difficult to model as it is the product of the behaviour of millions of molecules. For this reason, different mathematical solutions have been adopted to replicate the characteristics of water.

Research shows that the mathematical equations needed to simulate water are typically implemented with an equation, fundamentally driven by sinusoidal waves. Equations include; the Fast Fourier Transform (FFT), the Gerstner Wave, Vector Displacement Maps, used in *Assassins Creed IV: Black Flag* (Longchamps and Bucci, 2017) and Wave Particles used in *Uncharted 3: Drakes Deception* (Gonzalez-Ochoa, Holder and Cook, 2012).

Compared with the Gerstner Wave, the FFT is a more complex approach. However, it is commonly used in visual effects in the film industry (Tessendorf, 2001). The Gerstner Wave, on the other hand, is a less complex solution and like the FFT is used in the film and game industry. Furthermore, water simulation equations can be combined together to produce a variety of results. One example, is in *Hitman* (IO Interactive). To create a natural and computationally efficient simulation, Eidos LABS layered a Gerstner Wave eight times to cover every direction along with a low-frequency noise texture to break the repetition in the swells (Longchamps and Bucci, 2017).

Water simulations have the potential to be computationally expensive. To avoid this, the equations can be calculated on the Graphics Processing Unit (GPU). A GPU implementation of water could be done in multiple game engines including Unity, DirectX 11 or OpenGL. These game engines provide the ability to write Vertex Shaders. The Vertex Shader is where the manipulation of vertices takes place and has a plethora of uses (Nvidia, *no date*).

Another technique would be to use the GPU for general purpose calculations using a Compute Shader. Unlike the Vertex Shader, the Compute Shader does not have a specific position in the GPU pipeline (Green, 2009). Therefore, it could be used to run the complex fluid calculations and pass the result as input into a Vertex Shader which would apply the data. Other optimisation techniques of a water simulation include; dynamically altering the level of detail (LOD) of the water mesh and efficiently calculating buoyancy.

When developing the water simulation in *Hitman* (IO Interactive) LODing, the process of reducing the number of vertices in a mesh whilst keeping the original scale, was used. This was because, at times, the ocean would be rendered to 70% of the screen which, depending on the number of vertices, can be a performance hit (Longchamps and Bucci, 2017).

If required, buoyancy calculations, like water calculations, can become computationally expensive if not done correctly. One of the most efficient ways to implement buoyancy, in a simulation, is to use hydrodynamic calculations (Kerner, 2015). This technique can be computed on the GPU. However, the main issue surrounding this approach is that data would have to be passed to the central processing unit (CPU) impacting the simulations performance. To mitigate such an issue, the data calculated in the shader used to position the vertices could be replicated on the CPU without transforming vertex data. Instead, when required the same calculation could be run on the CPU to test the height of the vertex and then used in the hydrodynamic equation.

Outcomes

The application was built using the Unity engine. This is because unlike low-level engines such as DirectX11 and OpenGL, Unity allows the creation of materials which can be easily edited in the inspector. This functionality allowed the system to be built in the style of a plugin or Asset, similar to those found in the Unity Asset Store.

The mathematical equation used for this simulation was a hybrid system centring around a Gerstner wave. This is because compared to the alternatives the Gerstner wave is less complex to understand. Moreover, the hybrid system allows for more customisation of the waves aesthetics. As seen in *Figure 1* the Gerstner wave was layered with two other *cosine* waves used to amplify the simulation and to send subtle ripples across the surface of the water. Layering *sin* wave calculations this way creates the potencial for various interesting customisations through the material editor.

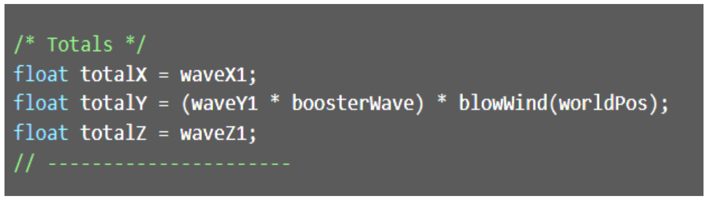

```
/* Totals */
float totalX = waveX1;
float totalY = (waveY1 * boosterWave) * blowWind(worldPos);
float totalZ = waveZ1;
// ---------------------
```

Figure 1 The results of all *sin* and *cosine* waves. *waveX1*, *waveY1* and *waveZ1* are the results scalar values taken from the Gerstner wave output.

To make the process of customisation more user-friendly a user interface (UI), which could be viewed and edited at runtime, was implemented. This system acted as the interface between the user and the material editor and created a fluid user experience. The UI system also enabled default values to be reapplied at any time during runtime.

The simulation runs on the GPU and utilises the Unity version of the High-Level Shader Language (HLSL). The HLSL is the Shader language used in DirectX which is mainly documented by Microsoft however, the Unity additions to the language are documented by Unity.

The buoyancy technique used in this project was influenced by the hydrodynamic equation and implemented on the CPU using C-Sharp (C#). The full calculation required to perform an exact hydrodynamic calculation needs information from the floating object which is complex to acquire. Instead of calculating the volume of the submerged area, the area of a single face named, cubeFaceSurfaceArea was used.

This lead to the problem where the data output from calculations run on the GPU were needed in the hydrodynamic equation calculated on the CPU. To solve the issue, the same calculation was computed on both the CPU and GPU, where only the displacement of vertices happens in the GPU. This meant that the closest vertex to the centre of the floating object could be acquired, which meant subsequent calculations could be run to apply force either *up* or *down* on the floating object based on the closest vertex position.

Evaluation

The water simulation was not calculated to meet specific aesthetic criteria but instead is the product of various iterations of manipulating the Gerstner wave and additional *cosine* waves. This technique for development is not ideal, despite having an aesthetically pleasing result, because it is difficult to replica due to their being no specific process or steps to achieve the outcome. For this reason, the application would not be viable option for a water simulation plugin.

Creating a water simulation involves looking at the application of the final product. The aim of this project was to create a simulation which is believable and is optimised enough to work in a game environment. These criteria were met but the simulation would need to be expanded on in order for a more optimised solution to be developed. The implemented simulation would work in specific situations such as demonstrations or quick solutions.

One issue which causes the simulation to be less adaptable as a plugin is the fact that the mesh cannot be scaled without drastically changing the shape of the waves. This is due to the way scaling works, as stretching the mesh causing the vertices to be stretched further apart of each other. A solution to this issue would be to create a separate integrated system which scales a mesh by adding more vertices. Such a system could be a good addition to a water simulation plugin. The plugin could also contain a system for changing the LOD dynamically based on the position of the camera.

The normal calculations are using a different method than that suggested in the GPU Gems implementation (Finch, 2005). By using the right normal calculations, the simulation may not suffer from the black patches seen when the customisable variables are all at their maximum value.

The implementation of getting the vertex height isn’t as good as it could be as the same calculations have to be run both on the CPU and GPU. A better implementation for this would be to output a height map to a render texture where each pixel or set of pixels is the height of each vertex, or use a Compute Shader which could output a Structured Buffer. Both of these implementations present a way for the Shaders output to be used as Shader input, thus removing the relationship with the CPU.

Developing water simulation in Unity is a good way to learn about the syntax and methodologies used in low-level rendering API such as DirectX11 or OpenGL. If implemented again more emphasis should be made on the integration with other game systems and different applications of the simulation within a single game engine, similar to the *Hitman* (IO Interactive) water simulation implementation (Longchamps and Bucci, 2017). The system however fulfilled a niche criterion which has the potential to be applied in different, specific situations.

References

[1] Gonzalez-Ochoa, C., Holder, D., Cook, E. (2012) From a calm puddle to a stormy ocean - Rendering water in Uncharted. *SIGGRAPH '12 ACM SIGGRAPH 2012 Talks*. Los Angeles, CA, USA. New York, NY, USA: ACM, Article 3.

[2] Finch, M. (2005) Chapter 1. Effective Water Simulation from Physical Models. In: Pharr, M., ed. *GPU Gems 2* [online]. Taunton, MA, USA: Addison-Wesley.

[3] Green, S. (2009) *DirectCompute PROGRAMMING GUIDE*. 05th May 2018.

[4] Kerner, J. (2015) Water interaction model for boats in video games. *Gamasutra* [blog]. 27th February. Available from: https://www.gamasutra.com/view/news/237528/Water_interaction_model_for_boats_in_video_games.php [Accessed 15th December 2017].

[5] Longchamps, N., Bucci, J-N., (2017) From Shore to Horizon: Creating a Practical Tessellation Based Solution. *GDC Vault* [videocast]. Available from: https://gdcvault.com/play/1023964/From-Shore-to-Horizon-Creating [Accessed 15th December 2017].

[6] Nvidia (no date) Vertex Shaders. Available from: http://www.nvidia.com/object/feature_vertexshader.html [Accessed 05th May 2018].

[7] Tessendorf, J. (2001) Simulating Ocean Water. SIG-GRAPH'99 Course Note.