Gerstner Wave emulation

Aldo Curtis

April 2, 2015

Word count:1085
Frederic Fol Leymarie.
AldoCurtis@gmail.com
https://github.com/Dizzly/SineWaveSimulation

Contents

1 Introduction

Efficient water simulation for games can come in may forms. A simple and efficient approach to modelling waves in both large and small sources is using sine waves to simulate wave movement.

2 Research

Simple but plausible simulations can be created without additional geometry and can still look convincing, for games where water is a main feature some physical simulation is normally required including a 3 dimensional representation. Texture and bump-map morphing can be used on flat planes to create the simulation of wave-fronts without adding geometric complexity, this can be done with perlin-noise for generative textures and sine wave generate bump maps to give the illusion of depth. Full fluid simulation is also possible for interactive and smaller water areas, assigning physics objects to "globdules" of water and physically modelling their interaction allowing the player to create splashes or pour water dynamically.

Most normally all of these simulations are far from realistic, instead balancing their runtime efficiency with the need for complexity as dictated by the games mechanics. For large bodies of water with little interactivity sine waves can be an effective and efficient tool to generate both geometry and normals for wave simulation. A enhancement to normal sinewaves is with the use of Gerstner waves, which naturally clump vertices near the peak of waves, automatically increasing the fidelity of areas with high curvature.

Sine waves have multiple perimeters to change their effects. Wavelength controls the frequency of waves, moving the peaks of the wave further apart of closer together, while amplitude controls how far the peaks are away from origin. A time value can be used to sample across the wave, effectively causing the representation of the wave to move at a set speed which can be controlled with the speed parameter, the speed is based on the frequency of the waves so that higher frequency waves will appear to move faster. The wave is projected onto a given direction, modelling it in 2D space, and a optional Q value can be included to sharpen the peak of the waves, and counteract the roll generated by sine waves.

3 Features

The program allows the saving and loading of files representing Gerstner waves. The files are human readable and should be easy to edit manually if need be, if no files is detected it will load some default waves to start out which can be edited and then re-saved at the users digression. Once your desired file is loaded into the program it will simulate the file perpetually, or on loop given your specifications, new waves can be added or removed and the peramiters of each wave can be edited and their effects viewed in real time.

The camera can be controlled through keyboard keys and space can be used to view the waves in wireframe mode.

4 Implementation

The project comes with 3 wave generation specific classes, 3 shaders, and the application section which is mainly responsible for UI and data management.

GerstnerWave is a class that holds a representation of a Gerstner wave, controlled by a few vital perimeters. It requires wavelength, amplitude, speed and direction and can take a Q value as an optional tweaking parameter. The main function for this class is Apply, which takes a position and a floating point number representing time. The position will be changed to the appropriate position on the wave based on the time. There are two methods of generating normals, one is using the function in the GerstnerWave class and providing each vertex with its appropriate normal, however the class also offers another way. The internal struct ShaderWaveStruct is created and aligned to be included as a uniform in the shader where normals can be created more precisely.

The equation used to generate the positions is as so.

$$\mathbf{P}(x,y,t) = \begin{pmatrix} x + \sum\left(Q_i A_i \times \mathbf{D}_i.x \times \cos\left(w_i \mathbf{D}_i \cdot (x,y) + \varphi_i t\right)\right), \\ y + \sum\left(Q_i A_i \times \mathbf{D}_i.y \times \cos\left(w_i \mathbf{D}_i \cdot (x,y) + \varphi_i t\right)\right), \\ \sum\left(A_i \sin\left(w_i \mathbf{D}_i \cdot (x,y) + \varphi_i t\right)\right) \end{pmatrix}.$$

Where A is the amplitude, D is the direction of the wave, w is the frequency of the wave and φ is the speed as relating to speed/frequency.
The GLSL shader code to generate normals is such.

```
x= -(dir.x*frequency*amp *cos(frequency*dot(dir,pos.xz)+time*speed))
y =1-(q*frequency*amp*sin(frequency*dot(dir,pos.xz)+time*speed)),
z =-(dir.y*frequency*amp *cos(frequency*dot(dir,pos.xz)+time*speed))
```

WavePlane holds a n*n mesh plane on the xz axis, which it will modify depending on wave parameters given to it. The plane is indexed and given texture co-ordinates across it and can be scaled up or down during creation or by modifying scaling through its matrix. To generate the new positions of its geometry it loops through each vertex and sums the total effect of every wave stored. It has two separate counts for objects, one being the total number of waves stored in the plane, and the other being how many of those waves are considered active, and will contribute to the vertex positions. This is used to control the number of waves and to phase them in and out without deleting them.

The waves use their own vertex and pixel shader to do some of the geometric work. As mentioned before normals a calculated in the vertex shader for each wave which are passed in as floating point uniforms. The fragment shader includes some simple specular and diffuse shading as well as texturing support.

5 Conclusion

The project works functionally as a tool however it has limited features and lacks a polished look. Further tools to create tiling and repeating waves could be included to aid the creation of game assets as well as procedural texture generation and bump mapping wave to make it look more realistic.

The simulation of oceans hold many nuances, and benefits from detailed shading using techniques such as BRDF shading as sub-surface scattering is responsible visual uniqueness of water.

6 References

Anonymous. 2015. Habib's Water Shader
Available at:http://habib.wikidot.com/ Enright, D et al. 2002. Animation and

Rendering of Complex Water Surfaces.
Available at:http://graphics.stanford.edu/papers/water-sg02/water.pdf Mark,

F 2004. GPU Gems. Addison Wesley. Effective Water Simulation from Physical Models.
Available at:http://graphics.stanford.edu/papers/water-sg02/water.pdf Lengle

Q. 2013. Water Waves, GLSL Perlin Noise
Available at:http://www.cornflex.org/?p=1019