|  |  |
| --- | --- |
| http://www.ccwater.org.uk/wp-content/uploads/2014/09/blue_wave_of_water.jpg  simulated worlds  Soft-Body Water Simulation | Edward McDowell  UFCFG5-30-2 Simulated Worlds |

Table of Contents

[**Introduction** 1](#_Toc416902307)

[**Research** 3](#_Toc416902308)

[Spring Models 3](#_Toc416902309)

[Smoothed Particle Hydrodynamics 3](#_Toc416902310)

[Verlet Integration 4](#_Toc416902311)

[**Implementation** 5](#_Toc416902312)

[Program Structure 5](#_Toc416902313)

[Setting Up the Plane 5](#_Toc416902314)

[Sine Simulation 7](#_Toc416902315)

[Verlet Simulation 7](#_Toc416902316)

[On-Screen Parameters 9](#_Toc416902317)

[**Evaluation** 10](#_Toc416902318)

[**References** 12](#_Toc416902319)

## **Introduction**

Water is everywhere. It is the very foundation of life on this planet; without it life as we know it would not exist. Given its importance to our planet and its abundance, it is understandable that we would want to create simulations for numerous reasons. Yet so many things need to be considered when modeling fluids, and this changes drastically depending on what effect you wish to achieve.

Some simulations may want to simply mimic reality as closely as possible to help further out understanding of fluid mechanics, or to model real life situations to map out ocean currents.

Figure 1: Cities Skylines dynamic water flow [5]



Figure 2: Water effects in Half-Life 2 [6]

In the context of games it can add an element to a world that makes it feel more real. In the recent title “Cities: Skylines”, a water simulation is used to great effect. The simulation models flow of water down rivers, carrying pollution and overflowing banks. The player can build a dam across rivers to provide power for the city, but if the water flow is too fast it can back up and overflow the banks of the river. This reinforces the game’s theme of city planning as bad planning can have serious consequences and undo your work. This effect suits Cities: Skylines, but other games may want need something different.

In the title “Half Life 2”, a level exist where the player uses a hovercraft to move around over water and land. The water in this situation does not need to flow, but it reflects the world around, refracts objects underneath the surface and a small splash/ripple occurs where any object interacts with it. This simulation is very different but fits well in the situation.

## **Research**

There are various different approaches for actually coding a simulation, and almost all have their own individual advantages and disadvantages depending on what may be required e.g. fast real-time simulation vs. accurate but with a slow calculation time, or only a surface vs. a fully-fledged fluid which can flow.

### Spring Models

Spring models rely on creating a set of point masses to form the shape of an object. These are then connected using various springs obeying a form of Hooke’s Law. Hooke’s Law states that:

F= k X\,

where  F is the magnitude of the force pulling on the free end of the spring,  X is the displacement of the spring from its resting position and k is the spring constant, a positive real number associated with that spring.

Usually it is written as:

F= -k X\,

to give the restoring force of spring(the force the spring is exerting to restore itself to the resting position).

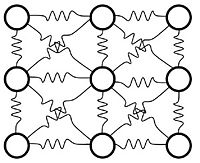


Figure 3: A set of nodes connected by springs

(Image taken from source [1])

In practice the spring model creates something akin to Figure 2. The nodes may be the vertices on the surface of a shape or `virtual` points forming the internal structure of an object. By tweaking the parameters of the springs, a shape can be allowed to compress and bounce back to its original shape, or with a low spring constant it could simple crumple up.

While the spring model is a good starting point for soft body objects with a particular shape, in terms of a water simulation it is rather poor at imitating flow. However the concept of springs is one to take note of.

### Smoothed Particle Hydrodynamics

This is a computational method for simulating fluid flow. It was not initially designed for fluids, but rather for simulating theoretical astrophysics (stellar collisions and the formation of galaxies and stars being a few examples) [2]. The model works by using a large set of elements (referred to as particles) to model the fluid which are able to freely flow around each other. Each of these particles can be given a variety of different properties. The basic formula gives quantity A at any point \mathbf{r} : 
A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(| \mathbf{r}-\mathbf{r}_{j} |,h),


where   m_j  is the mass of the particle   j ,  A_j  is the value of the quantity  A  for particle   j ,   \rho_j  is the density associated with particle  j , \mathbf{r} denotes position and  W  is the kernel function [4] which determines the range of particles to take into account in the formula (various different kernel formulas exist). An example of this kind of simulation can be seen here [3].

One advantages of using this method are that different fluids may interact as the particles can be given different densities, which will let them disperse within each other and still be able to separate out and be distinct. It can also be used for creating a full-fledged fluid with flow. However there are some drawbacks to this method; to create a simulation that resembles fluid some extra rendering is needed to create a surface geometry over all the particles. It also tends to require a huge number of particles to cover the same size area as other approaches which would be less intensive. In the wider context of most games this is a significant drawback, and the extra fidelity of the simulation is not worth the processing cost.

### Verlet Integration

This approach is best applied to simple surfaces/planes, which is ideal in the consideration of a game engine. There are various forms of Verlet integration but the appropriate formula is as follows:



Applied in the context of a surface, X is any given vertex on the plane, *n* is the current loop iteration and *A* is the acceleration of the vertex.

In plain terms, the vertex’s next position in an axis is equal to twice the current position, minus the previous position, plus an acceleration value multiplied by the current position and the square of the time passed since the last iteration.

However this alone will not give the desired effect. For a water simulation the position of vertices around must be taken into account. This is done with a formula for calculating wave diffusion in a height field. In this instance the equation will take the position of the points around it. [7]

In plain terms, a diffusion gradient is calculated using the by summing the height positions of the vertices surrounding the current vertex and subtracting the current vertex’s height multiplied by the number of surrounding vertices. All this is multiplied by a factor denoting the rate of diffusion, the result is multiplied by the current time step and added onto the position calculated from the main verlet equation.

Using a method like this allows for a believable representation of the surface of water being disturbed, with ripples easily flowing outwards. It is also less computationally expensive compared to other methods, as the underlying formula uses basic mathematical operators.

## **Implementation**

### Program Structure

The simulation is coded in C++, using DirectX ToolKit. It relies upon the use of class based inheritance, allowing a given surface to inherit the base properties of a game object; these include a position vector, orientation variables, a rotation matrix, a world matrix and Tick and Draw functions, allowing the surface to be called with all other game objects. This is built upon to create an object created with vertices that can be drawn into the world and from there has its own variables and functions governing the actual water based behavior.

Another important element is a moveable object that can interact in some way with the water surface, allowing for the user to instigate dynamic behavior. This also inherits from the base game object; however it is created from a model rather than a set of vertices. The model loading behavior will also be inherited. The class itself will simply move around based on user input.

### Setting Up the Plane

As the plane needs a set of vertices to manipulate, it must inherit from the class VBGO. This class sets up the data required for storing and drawing the shape. Some of the functions that are inherited can be seen in Figure 4. This includes creating an Index Buffer and a Vertex Buffer for an object, and Raster States among other things. Most of the initial codebase for this class can be left untouched. However the simulation requires that we are able to manipulate the vertices positions, so a dynamic vertex buffer needs to be set up.

A new function exist within the VBGO class that when called builds a dynamic vertex buffer. The only difference in the code between this and the default vertex buffer is the usage is set to dynamic, and the CPU Access is set to write allowing the CPU to change data stored in this vertex buffer. Figure 5.

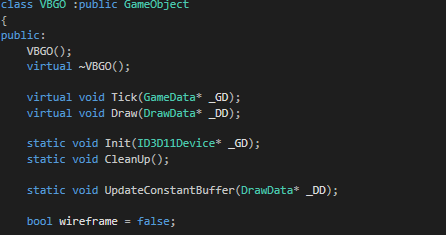
0

Figure 4: The public properties within VBGO

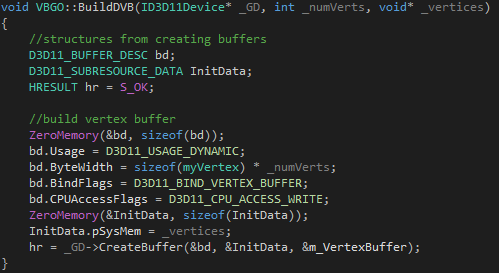


Figure 5: A custom function for creating a Dynamic Vertex Buffer

The water surface class (VBPlane) is a VBGO, so is able to call any of the functions within the VBGO class. During its initialization, it sets up its own vertices with the required data (position, colour, texture co-ordinates and normals) then calls the inherited BuildDVB function to create its own dynamic vertex buffer that is populated with these vertices.

BuildDVB(GD, numVerts, m\_vertices);

This sets the initial vertex data. However the positions of the vertices can change every tick, so a method is need to write in the new data to the vertex buffer. This requires disabling the GPU access to the buffer, allowing the CPU to write in the new data, and then re-enable the GPU access. The code for this can be seen in Figure 6.

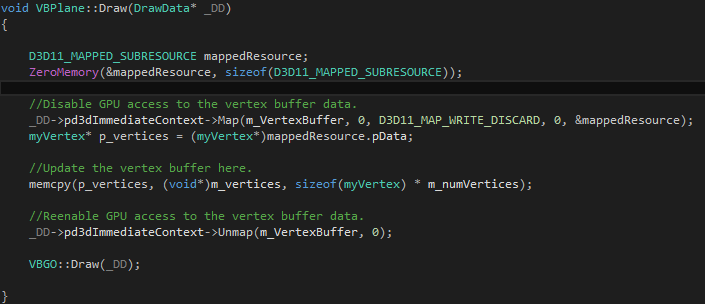


Figure 6: Disabling and re-enabling GPU access to the vertex buffer.

This Draw Function is called over the inherited VBGO Draw, but the VBGO Draw functionality is still required, so this is called after the vertex data is re-mapped into the vertex buffer.

### Sine Simulation

Once the dynamic vertex buffer was established, a method was required to test it was working properly. This started as a simple sine wave function but gave rise to a simulation that relied on using sine waves to simulate water and ripples. This equation took the form of:

where is a three dimensional vertex position, is the amplitude of the wave, is the frequency, is the total time elapsed and is the wavelength. Using this function a vertex will oscillate in the y-axis and a continuous sine wave will appear to travel across the plane in the x-axis. This creates an easily parametrized effect that can be fairly reminiscent of a water surface.

From here, a class was created to make ripples. The class was contains a single function of type float called Calculate. The VBPlane has a vector of pointers to ripple instances. When the RETURN key is pressed, a random point on the plane is generated, and a new instance of the ripple class is created. The random point is passed in to the ripple class, as are the desired values for each of the sine-wave formula variables. Every tick a for loop runs through each vertex and calls each Ripple instances’ Calculate function. The vertex’s x and z positions are passed in.

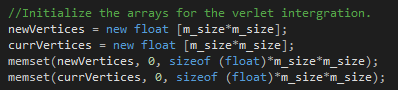
In the Calculate function, Pythagoras’ theorem is used to find the distance between the current vertex and the center of the ripple this is then used in place of in the sin formula, and the amplitude, frequency and wavelength are scaled down depending on this distance. The formula gives a float value that is returned. This signifies the y value for the current vertex based on that ripple. Reach ripples returned value is added together and a mean is taken. This is then averaged with the standing normal traversing wave. This gives a final value that the vertex’s y value gets set to. Every frame a falloff value is taken away from the each ripples amplitude, so over time each ripple becomes smaller and smaller. Once the amplitude falls below zero, the ripple is deleted and erased from the vector of ripple instances.

This simulation produces very nice effects with a single ripple. However, the more ripples are added the slower the simulation runs. One of the goals for the end simulation is to be able to have a wake following a user controlled object. This would rapidly grow the number of ripples to untenable levels. Therefore this simulation is not suitable, which is no surprise as it was an experiment born from testing the dynamic vertex buffer.

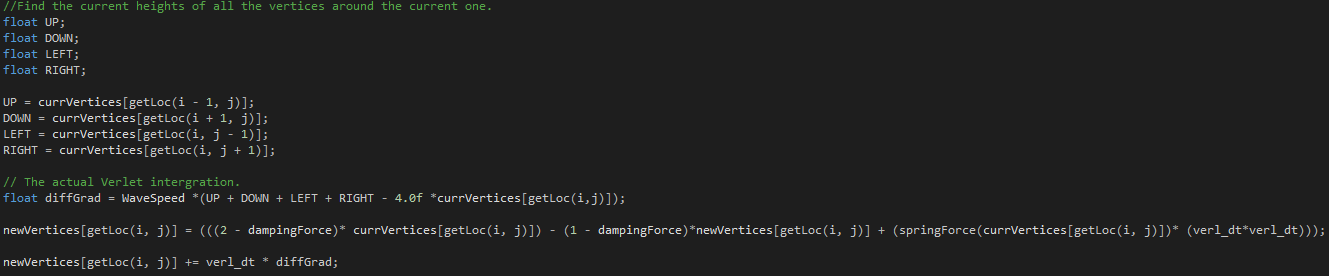
### Verlet Simulation

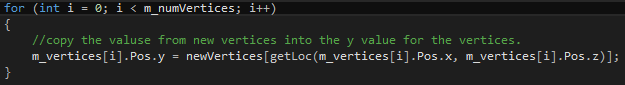
This implementation uses the verlet integration and diffusion formulas discussed in the research. However in the actual implementation some tweaks were necessary as it could be unstable. The method requires the use of previous values for the vertices heights, so a method of storing these is needed. A set of pointers to type float are used to store the heights. There are declared in the header file, then set to be arrays of the correct size in the planes initialization.





At the beginning of a tick, currVertices will be storing the current heights of the vertices, while newVertices will be storing the heights from the start of the previous tick. As soon as the previous heights are used they are overwritten by the new height gained from the verlet integration. The diffusion gradient is then applied to this new value, and finally the actual vertex position (in m\_vertices) is set to the new value.



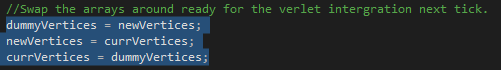


The getLoc function used throughout is a function that converts 2 dimensional to the nearest 1 dimensional co-ordinate within an array. This needs to be done numerous times throughout the code so separating it into its own function is the most logical choice.

In the verlet equation, dt has been replaced with the variable verl\_dt, which is a float with a fixed value of 0.01. This is due to the fact that the simulation can become very unstable if the number is too large.

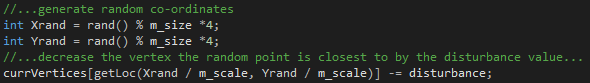
There is also a new variable added into the equation; dampingForce. This is tweakable by the user, and its effect on the simulation is to reduce the spread of the waves, which allows for much smaller ripples to be created.

There is also a call to a separate function called springForce. As previously stated, the concept of springs is one to take note of. The function springForce takes in the current height of the vertex and uses it as the displacement value in the Hooke’s Law equation to return a value used that is then used as the acceleration variable in the verlet equation. Doing this helps to stabilize the simulation, as the force is always pulling it back to the resting value of 0.



After all the vertex heights have been replaced with new values, the stored heights need to be swapped around ready for the next tick. dummyVertices is used to hold values while the arrays are set to each other.

To cause a disturbance in the surface, the value of currVertices is altered before the verlet solver runs for that tick. An example can be seen below:



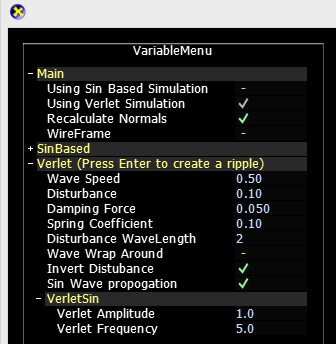
This code is run to create a disturbance at a random point in the plane. It simply subtracts the disturbance parameter from the current vertex height. The code for creating a disturbance at the player is virtually identical, except the players x and z values are fed in in place of the randomized ones.

#### Verlet Sin

Since the sine wave based simulation is implemented, it was a simple matter to re-use some of the code to make the vertices along one edge of the plane oscillate in a sine pattern. The sine formula in this instance doesn’t take in a wavelength. The outputted value is the same regardless of the vertex inputted. The value is applied to the vertices along one edge of the plane, and the verlet based simulation propagates the wave across the plane.

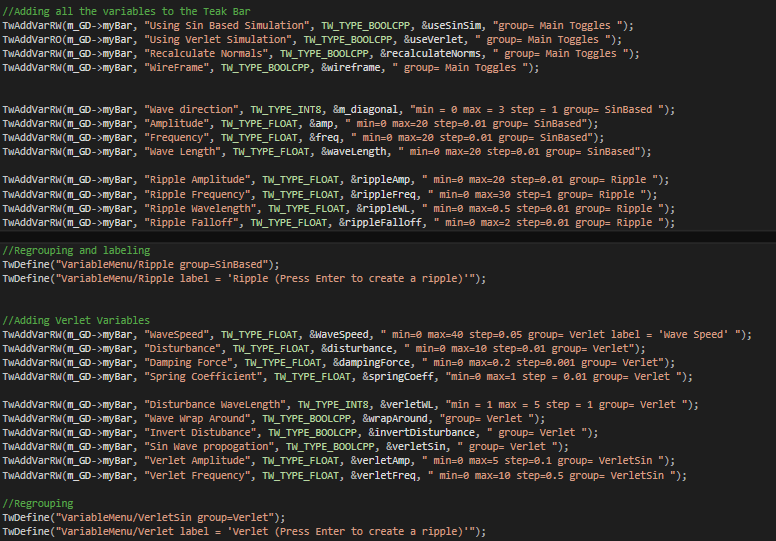
### On-Screen Parameters

The simulation has many parameters that can be tweaked, and a suitable method was required for altering these on the fly. The solution in this simulation is an external library called AntTweakBar [8]. It allows the creation of GUI menus, which can directly take mouse and keyboard input. The parameters can be added to a menu and grouped together. Since there are two different simulations within this codebase, this grouping is effective for keeping each simulations parameters separate.

.

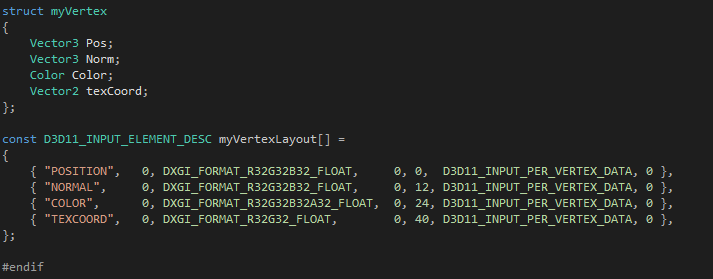
## **Evaluation**

Through the course of development, it was necessary to understand exactly how the desired simulation ought to behave. The desired outcome needed to meet the expectations of how a water surface would behave. Flow however was not necessary, which meant more advanced methods like Smoothed Particle Hydrodynamics where unnecessary.

This implementation is of course, not perfect. For instance, the handles for altering parameters through the AntTweakBar interface were set up within the VBPlane class. Just a section of this code shows how bloated the class became. 

All of this code does not need to be within the plane itself, but could be in a separate class that just deals with AntTweakBar. This could easily be done by just creating a pointer in that class to the VBPlane, and accessing the variables through that, but during development it was simpler to add variables as they were needed.

The ideal for this simulation is to make it as simple as possible to drop it into a separate game engine.

This simulation is currently not in that state. As mentioned above, all of the interface for the menu would need to be removed, but this is fairly simple. If that was done though the VBPlane class could still not be lifted out as is. Since it inherits from the VBGO class, which in turn inherits from the gameObject class, these would both need to be set up in an engine. The VBGO class also makes use of the myVertex Struct which requires the following code: 

Assuming the game engine has some analogous version of these classes then the simulation should be easily ported.

The sine-based simulation could also use a lot of optimization. A method for doing this would be to create a lookup table of values, as the way it is currently set up, every vertex needs a sine calculation for every ripple. If a lookup table were made it could reduce the processing needed substantially and make the simulation more viable as a practical method.

A good method would be to create a new function in the VBPlane to deal with disturbances. An Event Manager could generate an event and call the function if an object were to collide with the surface. The function could take in the world co-ordinates of the object creating the disturbance, convert them to the local co-ordinates of the VBPlane, and then use them to increment the nearest vertex down.

Further improvements could be made by making the VBPlane scalable. Currently it scales uniformly in each direction, but that could be changed by separating the one variable into an X scale and a Z scale variable. Doing this could allow the size but the simulation would stretch in one axis if they weren’t even. A possible method for solving this would be to calculate the factor difference between each axis’ scale value. This can then be used in the diffusion gradient formula.

The variable below can be multiplied by the appropriate scale factors to even out the diffusion in all directions. UP and DOWN would be scaled by the Xscale variable and LEFT and RIGHT could be scaled by the Zscale value.



This change would make the plane much more adaptable, as currently it works solely as a square.

A final addition that could be added to this simulation is the use of shaders to create reflection on the surface.

One method to achieve reflection would be to create a camera needs to be set up below the water surface, directly below the main gameplay camera, with an opposite pitch to the main gameplay camera. An initial render pass needs to be done for this camera, set to only draw objects above the surface. This needs to rendered to a texture which must then be passed back in to the surface and applied.

## **References**

[1] Wiki.blender.org, (2015). Doc:2.6/Manual/Physics/Soft Body/Interior Forces - BlenderWiki. [online] Available at: http://wiki.blender.org/index.php/Doc:2.6/Manual/Physics/Soft\_Body/Interior\_Forces [Accessed 14 Feb. 2015].

[2] Owen, J., Villumsen, J., Shapiro, P. and Martel, H. (1998). Adaptive Smoothed Particle Hydrodynamics: Methodology. II. ASTROPHYS J SUPPL S, 116(2), pp.155-209.

[3]4GamerMovie, (2014). *Smoothed Particle Hydrodynamicsベースの波表現*. [video] Available at: https://www.youtube.com/watch?v=3Y\_QTY4C08I [Accessed 13 Mar. 2015].

[4] Wikipedia, (2015). Smoothed-particle hydrodynamics. [online] Available at: http://en.wikipedia.org/wiki/Smoothed-particle\_hydrodynamics#Method [Accessed 14 Apr. 2015].

[5] Cities Skylines. (2015). Paradox Interactive & Colossal Order Ltd.

[6] Half-Life 2. (2004). Valve.

[7] Miklós, B. (2015). *Real-Time Fluid Simulation Using Height Fields*. [online] p.16. Available at: http://www.balintmiklos.com/layered\_water.pdf [Accessed 15 Apr. 2015].

[8] Decaudin, P. (n.d.). *AntTweakBar*. Available at: <http://anttweakbar.sourceforge.net/doc/>

[Title Image] Consumer Council for Water http://www.ccwater.org.uk/, (n.d.). [image] Available at: http://www.ccwater.org.uk/wp-content/uploads/2014/09/blue\_wave\_of\_water.jpg [Accessed 15 Apr. 2015].