

Waveequation based water surface animation

Helmut Bressler

January 17, 2007

1 Overview

This work is based on the paper “Fast Water Animation Using the Wave Equation with Damping” of Y. Nishidate and G. P. Nikishkov. First a brief overview of the suggested approach is provided. The following sections deal with the implementation details as well as with the interface for application programmers.

2 Problem description

As described in the paper modeling the behavior of a water surface with the traditional 2D wave equation is not preferable for interactive graphical applications. This is due to the fact that the 2D wave equation does not consider the amount of energy lost when a particle deviates from the zero energy state (that is if a particle resides on the water surface). Therefore it is proposed to add a term which should lead to a damping effect. This results in the following equation:

$$\frac{\partial^2 h}{\partial t^2} + k \frac{\partial h}{\partial t} = c^2 \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right) \quad (1)$$

where h is the height of the water surface, t is the time and (x, y) is the 2D position vector. The parameters k and c determine the behavior of the water: c is the wave speed and k is the damping factor.

3 Solving the equation

A finite differencing approach is used to solve the equation. All second order derivatives are obtained by a central differencing approach, the first order time derivative is approximated by a backward differencing approach. This leads to the following discretized version of the equation:

$$h_{x,y}^{t+1} = h_{x,y}^t + (1 - k\Delta t)(h_{x,y} - h_{x,y}^{t-1}) + c^2 \frac{\Delta t^2}{\Delta x^2} L(h^t)_{x,y} \quad (2)$$

where $L(h)$ is:

$$L(h^t)_{x,y} = -4h_{x,y}^t + h_{x+1,y}^t + h_{x-1,y}^t + h_{x,y+1}^t + h_{x,y-1}^t \quad (3)$$

The authors of the paper suggest that the following setting of $L(h)$ leads to a better visual appearance:

$$L(h^t)_{x,y} = -4h^t_{x,y} + h^t_{x+1,y} + h^t_{x-1,y} + h^t_{x,y+1} + h^t_{x,y-1} + \frac{1}{2}(-4h^t_{x,y} + h^t_{x+1,y-1} + h^t_{x-1,y+1} + h^t_{x+1,y+1} + h^t_{x-1,y-1}) \quad (4)$$

This setting of $L(h)$ considers all neighbours of h .

3.1 Boundary conditions

The suggested implementation treats points on the boundary in a special way. The lacking height values outside the domain are replaced by interior points during calculation. This should enhance the reflection of waves at the border. My implementation treats values outside the domain as zeros, which still makes the occurrence of reflections possible while the implementation is straightforward and saves some computational work (in particular in cases where the domain is not a rectangle).

4 Implementation

The implementation is splitted into two parts: One solely performs the simulation, which is platform and graphics API independent and the other one is responsible for visualization. The whole simulation part resides in the class `WaveEquationWater` provided in the file `waveeqwater.h` and `waveeqwater.cpp`. The visualization is performed by a class derived from `WaveEquationWater`. From now on the `WaveEquationWater` class will be called simulation class and the derived class will be called visualization class. The simulation class provides the following data to the visualization class:

- *position* (x, z) position of the vertices representing the grid. The (x, z) position of a vertex never changes.
- *height* the y value of a grid vertex. This value is adjusted during each simulation step.
- *normals* the surface normals pointing towards positive y . The normals also change during the simulation
- *indices* describes a list of triangles which can be used for rendering the water surface. 3 indices are used to describe a triangle. The indices may change. Whenever this happens `updateIndices()` is called in order to notify the visualization class.

4.1 WaveEquationWater base class

Basically `WaveEquationWater` runs the simulation and supplies the derived class with data structures necessary for building up the geometry data for rendering. Since processing the simulation is an independent task the `WaveEquationWater` class is written in such a way that the simulation and the rendering methods can be executed in separate threads. However its not allowed to call all methods once the simulation thread has been initiated and for performance reasons only very few error checks are made. Typically the following steps are performed to setup a single threaded simulation environment:

```
WaveEquationWater* waveEqWater = new MyWaveEquationWater();
waveEqWater->init(xSize, zSize, xResWater, origin.x, origin.y, origin.z);
```

Here `MyWaveEquationWater` is the name of visualization class. The setup of the data structures is performed in `init()`. The two first parameters `xSize` and `zSize` specify the size of water surface. The `origin` vector specifies the corner of the rectangular water surface with the lowest `x` and `z` values. The `xResWater` parameter contains the number of grid points along the `x`-axis. The number of grid points for the `z`-axis is determined by the ratio between the `xSize` and `ySize` such that the grid has an equal spacing on both axis in world space. The `init()` methods needs not to be overwritten in the visualization class. In order to perform any setup based on the parameters passed to `init()` the method `initRenderingStructures()` may be overwritten. This method will be called by `init()` after the simulation has been brought into an initial valid state. During the simulation loop calling `simulate()` tells the water simulation to proceed one timestep:

```
static float lastTimeStamp;
float currentTimeStamp;

...

waveEqWater->simulate(currentTimeStamp-lastTimeStamp);
lastTimeStamp = currentTimeStamp;
```

The `simulate()` method expects as parameter a float containing the elapsed time in seconds since the last call was made. Calling `simulate()` does not immediately invoke the whole simulation procedure (which is very timeconsuming). The reason for this is that the simulation always is conducted with a fixed timestep size for stability reasons (actually to prevent the elapsed time from becoming too large). Whenever `simulate()` has been called the content of the `height` and the `normal` members may have changed. There is no notification mechanism which makes the visualization class aware of this. This is because in the multithreaded version the rendering methods are expected to run in a different thread than the simulation procedure is taking place. Therefore it is recommended to update the rendering structures which depend on the height and normal values each frame.

4.1.1 Simulation properties

The behavior of the simulation is determined by the following methods, which should be invoked right after `init()` and before the simulation loop is entered:

```
void setSimulationStepsPerS(unsigned sps);
void setDampingConstant(float d);
void setWaveSpeed(float s);
```

`setSimulationStepsPerS()` sets the duration of the fixed sized timestep, $\Delta t = \frac{1}{sps}$ to be precisely. Higher values make the simulation appear more realistic and are also beneficial for simulation stability. However setting the number of simulation steps to a value larger than the actual elapsed time between calls to `simulate()` slows down the propagation of the waves.

The damping constant (thats k in equation (1)) set by `setDampingConstant()` influences how fast the wave is loosing energy. It should be chosen such that $0 \leq k\Delta t \leq 1$. The effect can be seen in equation (2): With $k\Delta t = 0$ (2) becomes the discretization of the ordinary wave equation, no damping occurs. For $k\Delta t = 1$ equation (2) is a discretized version of the heat equation. The method `setWaveSpeed()` determines the propagation speed of waves. The parameter s is c in equation (1). The unit is in world coordinates per second. Setting c to a high value has an impact on simulation stability.

4.2 Simulation stability

The paper does not contain a detailed stability analysis. But the following statements can be seen easily when considering high frequency waves:

- $L(h)$ tends to pull $h_{x,y}^{t+1}$ into the direction of its neighbors
- the term $(h_{x,y} - h_{x,y}^{t-1})$ supports $h_{x,y}^{t+1}$ in its current movement

Here is a necessary condition for the stability: Lets consider a chess board like setup for h : the black fields $h^t = 1$ and for the white fields $h^t = -1$ and for h^{t-1} it is the other way around. For the sake of stability h^{t+1} must not exceed h^{t-1} . Assume $h^t = 1$. In this case $|L(h)| = 8$. When inserting into (2) this leads to the following condition:

$$-1 \leq 1 + 2(1 - k\Delta t) - 8c^2 \frac{\Delta t^2}{\Delta x^2} \quad (5)$$

after simplifying we get:

$$k\Delta t + 4c^2 \frac{\Delta t^2}{\Delta x^2} \leq 2 \quad (6)$$

Surprisingly the damping constant can be trade for either a larger timestep, a higher grid resolution or a higher wave speed.

4.3 Setting up the boundary

In addition to the border also interior points of the grid can be marked as fixed. Fixed grid points have zero excitation and thus act as reflective border if they are neighbors of non fixed grid points. Interior fixed points cannot be set manually (at the moment, as there is no need for such a feature). Instead they are generated automatically whenever the method:

```
void setHeightMap(HightMapInterface* heightMap);
```

is called. The class `HeightMapInterface` is an abstract class which only provides very basic services of a heightmap such that it can be integrated in any existing heightmap class easily. A grid point is becoming a fixed point if it resides below the surface defined by the `heightMap`. Additionally `setHeightMap()` regenerates the set of triangle indices used for rendering as fixed points are invisible for now and need not to be rendered. The visualization class is notified about this through `updateIndices()`. In a multithreaded setup this method has to be called before the simulation thread has been initiated.

4.4 Interaction

Setting and retrieving height values of the water can be achieved through the following methods:

```
void setHeight(int x, int z, float h);
float getHeight(int x, int z);
```

where the tuple (x, z) must be provided in grid coordinates. World coordinates can be transformed into grid coordinates by utilising the line intersection methods:

```
bool getIntersectElement(float* pos, float* dir, int* x, int* z);
bool getIntersectCoords(float* pos, float* dir, float* x, float* y, float* z);
```

pos, *dir* must be a 3 component float array, specifying the direction and the position of the line. The line intersection test is performed with the plane representing the rectangular grid (all points are considered to have zero height). If the intersection tests succeeds (the return value is true) `getIntersectElement()` writes the grid coordinates of the point which is closest to the intersection point into *x* and *z* and `getIntersectCoords()` writes the intersection point in world coordinates into *x*, *y* and *z*. In case the line defined by *pos*, *dir* does not intersect with the grid plane both methods return false.

Note that `setHeight()` does not immediately write the height value into the grid. Instead the height values are set during the next simulation step (not necessarily the next call of `simulate()`). This behavior has been chosen in order to reduce synchronization overhead in cases where the simulation has its own thread. Both methods `setHeight()` and `getHeight()` make use of thread synchronization mechanisms so they might be slow.

4.5 Running the simulation in its own thread

Since the simulation class was intended to be platform independent it does not contain any threading API code. However, in order to support development of a multithreaded setup there is static method provided which runs the simulation loop:

```
static void run(void* instance);
```

here *instance* must be of the type `WaveEquationWater*`. This method cooperates with the `kill` method:

```
static void kill(WaveEquationWater* instance);
```

terminates the simulation loop. When `kill()` returns `run()` has already reached the `return;` statement. From now on it's safe to call `delete` on *instance*. Though the use of this both methods is limited to cases where a static method only is provided to a create thread function and leaving this methods leads to the termination of the thread. Synchronization between the two threads is achieved by making use of a single mutex. Methods for accessing this mutex must be provided by the derived (visualization) class:

```
virtual void acquireLock();
virtual void releaseLock();
```

The mutex should be created in the constructor of the visualization class. In certain applications (i.e. where the number of active threads exceeds the number of available processors) it can be desirable to limit the amount of execution time spent in the simulation loop in the `run()` method. This is what following method is for:

```
virtual void slowdown();
```

This method is called by `run()` whenever a call to `simulate()` has performed a simulation step. It needs not be implemented in the visualization class, by default it simply does nothing.

In order to determine the elapsed time between two simulation steps the method

```
virtual double getTimeStampInS();
```

is invoked which returns a timestamp in seconds. In the simulation class this method is implemented by using `clock()` which is platform independent but has some other drawbacks (see manpage). Therefore the derived class should provide its own implementation.

4.5.1 Creating a custom simulation thread

Except for `acquireLock()` and `releaseLock()` the methods presented above only serve as reference implementation for a multithreaded setup of the water simulation. In fact they are used nowhere else and can be completely replaced by a custom implementation. Also a lot of processing time can go to waste if `slowdown()` is not implemented properly. In particular if more than one water instance is used or other physical time based effects simulations it might be preferable to choose the number of simulation loop threads as the number of processors - 1, where each loop contains several simulation entities. Such situations would certainly require some more sophisticated simulation loops.

At this place it's worth mentioning that I hardly ever managed to restrict the processing time of `run()` by using `slowdown()` in such a way that a single setting works on all hardware/platforms. According to my experiences the effect is always highly dependent on the operating system and even on individual hardware components (number of CPUs, number of threads running on the CPU simultaneously, graphics chip, ...). I recommend using a multithreaded water simulation only in cases where more than one CPU is available.