Creating Pretty Water Scenes

Scott Skirlo

Background and Motivation

I explored some basic simulations of gravity waves in water. There are relatively few examples of pretty and or accurate real-time water simulations. The work I followed was one of the first main examples which implemented a computationally reasonable and semi-realistic water simulation which could potentially be run in real time [1]. In this work they were able to demonstrate basic effects like wave-reflection and interference, and more advanced effects like wave refraction and the formation of isolated “puddles”. This work seemed like a reasonable starting point for a project since the algorithm seemed straightforward and the basic results of the paper looked interesting.

Computational Method

The basis of the method was an implicit solution to the 1D gravity wave equation. In this method the equation for gravity waves in water is rewritten in the following way:

$$\begin{aligned} h_i(n) = {} & 2h_i(n-1) - h_i(n-2) \\ & - g(\Delta t)^2 \left(\frac{d_{i-1} + d_i}{2(\Delta x)^2} \right) \left(h_i(n) - h_{i-1}(n) \right) \\ & + g(\Delta t)^2 \left(\frac{d_i + d_{i+1}}{2(\Delta x)^2} \right) \left(h_{i+1}(n) - h_i(n) \right) \end{aligned} \quad [1]$$

Here n denotes the time index and i is the space index. The key feature of this equation is that the spatial derivatives are effectively taken for the field we are going to compute. This is what gives the implicit method it’s stability, we are computing derivatives we need now, from future results, which allows us to more closely follow the true solution and maintain stability.

The expression can be rearranged into a matrix equation of the following form:

$$\mathbf{A}h_i(n) = 2h_i(n-1) - h_i(n-2)$$

$$\mathbf{A} = \begin{pmatrix} e_0 & f_0 & & & & \\ f_0 & e_1 & f_1 & & & \\ & f_1 & e_2 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & e_{n-3} & f_{n-3} \\ & & & & f_{n-3} & e_{n-2} & f_{n-2} \\ & & & & & f_{n-2} & e_{n-1} \end{pmatrix}$$

$$e_0 = 1 + g(\Delta t)^2 \left(\frac{d_0 + d_1}{2(\Delta x)^2} \right)$$

$$e_i = 1 + g(\Delta t)^2 \left(\frac{d_{i-1} + 2d_i + d_{i+1}}{2(\Delta x)^2} \right)$$

$$e_{n-1} = 1 + g(\Delta t)^2 \left(\frac{d_{n-2} + d_{n-1}}{2(\Delta x)^2} \right)$$

$$f_i = -g(\Delta t)^2 \left(\frac{d_i + d_{i+1}}{2(\Delta x)^2} \right)$$

This relates our states from the past two steps to our future state through the matrix A, computed with the depth d defined as h-b, where b is the profile of the “ocean” floor. The matrix A is in tridiagonal form, which allows it (in principle) to be inverted and computed efficiently, for even very large matrices. This type of sparse matrix occurs very often in physical systems, so the methods for handling them and populating them are well developed.

There are a few other details that need to be taken care of to ensure that the simulation runs properly. One issue arises when h goes below b (i.e. the water depth is negative). Obviously this is not a physically acceptable situation, we know that there should be no dynamics if the water is below the ground! The way we solve this is by taking d=0, if d=h-b<0, before we calculate the matrix A. Another point we correct the simulation is after computing the new h(n). At this point h could also lie below the ground, so we take it to be h(n)=b-epsilon, where epsilon is some evil number (we’ll see more about that shortly).

It turns out this entire game ruins the volume conservation. This means technically that we have to keep track of every separated bit of water and adjust the height of each separated bit of water to conserve volume every timestep. I made an attempt to implement this part of the algorithm legitimately in 1D, but in the end for my shore simulation, I just ensured that the volume was globally conserved, instead of for each volume.

Taking this all together, the pseudocode for computing the waves takes the following form:

for each timestep:
    compute d from h(n-1).
    if d<0, d=0
    compute A

compute h(n) from A^-1(2*h(n-1)-h(n-2))
adjust h(n) in “filled” areas to conserve the overall volume
if h(n)<b, h(n)=b-epsilon

Implementation

I used matlab for developing and debugging quickly. It would be difficult to do this rapidly in C++, but if I had done this, the algorithm would have been able to run in real time. In matlab it took about 5 min to produce an 8 sec video. Integrating forward in time wasn’t the hard part, since the implicit solver could take quite large timesteps I was always planning on ray casting images, so I could get “refraction” effects, but if I had gone with opengl, perhaps I could have done things faster because I could have “instantly” seen the effects of my changes in the code. Only looking at snapshots from matlab was not a very effective method for spotting issues. After I was satisfied with a given scene, I moved to render this with the pset 5 ray tracer. It a few hours to render a 7-8 sec movie with about 250 frames.

Creating the scenes

The first scene was a simple fluid with sine perturbations at two points. This was able to generate some pretty refraction effects. It’s also easy to see the reflection of the generated waves off the boundaries of the simulation cell.

I thought about generating these kinds of “radial” point sources with some procedural/analytic methods, but it seemed difficult. Since in principle (with a constant depth) this is a 2D scalar wave equation, it is possible to expand any set of initial conditions in terms of the eigenmodes and let the system evolve. In practice if we do this with basis functions from a rectangular simulation cell, the results don’t look very good. They were not very “radial”; they were too boxy. Naturally then if we want a more “radial” solution, the answer is to use “radial” basis functions. In 2D for the scalar wave equation, it turns out these are Bessel functions, which are at least well tabulated. However, if we want to actually evaluate the coefficients for the eigenmodes for a delta function perturbation, it turns out the integral we have to perform numerically is notoriously difficult to get right. There is an “approximate” radial solution, but unfortunately the region where it’s valid grows rapidly with time!

Creating the shore scene was more interesting. I decided to create a “cone-shaped” shoreline. I initialized a gaussian distortion away from the shoreline. After starting the simulation, this gaussian distortion split into one forward and one backward propagating wave. After reflecting off of the back boundary, both waves propagated towards the shoreline. As the waves approached the shoreline, the left hand

side sped up relative to the right hand side, because the speed of gravity waves goes like (g*d)^0.5. This meant the shallower waves went slower, creating a bending effect. After the waves hit the beach, they broke up a little into separate pieces, creating interesting sliding and sloshing effects. After the waves fell back into the water, they generated waves which propagated and interfered with other waves traveling away from the shore.

Getting this simulation to work properly turned out to be more difficult than I expected. Many of my difficulties originated from the “evil epsilon” I mentioned earlier. When I was trying to find the best parameters for the shore scene, I would use large timesteps to quickly get a feel for what the simulation looked like, and then use small steps to create the video. When I started doing this, I began seeing an error from the detailed videos where the shoreline appeared to “repel” the water. For a long time I concluded this was connected to a bad implementation of the volume conservation routines or the depth correction methods. However, it turned out the entire thing was an artifact of the relative scale of epsilon to the timestep. When the timestep was much bigger than epsilon (as was the case when I was doing my fast simulations), everything appeared normal. However, when I did the fine simulations, epsilon became relatively close to the timestep which began to cause significant errors. Thankfully making epsilon extremely small fixed the issues which allowed me to have a nice simulation of the coastline.

Sources

[1] Kass, Michael, and Gavin Miller. "Rapid, stable fluid dynamics for computer graphics." *ACM SIGGRAPH Computer Graphics*. Vol. 24. No. 4. ACM, 1990.