Interactive Water Surfaces

Jerry Tessendorf – Rhythm and Hues Studios
jerryt@rhythm.com

Water Surfaces in Games

Realistic computer generated ocean surfaces have been used routinely in film features since 1996, in such titles as *Waterworld*, *Titanic*, *Fifth Element*, *Perfect Storm*, *X2 XMen United*, *Finding Nemo*, and many more. For the most part the algorithms underlying these productions apply Fast Fourier Transforms (FFT) to carefully crafted random noise that evolves over time as a frequency-dependent phase shift [Tessendorf02]. Those same algorithms have found their way into game code [Jensen01, Arete03] without significant modification, producing beautiful ocean surfaces that evolve at 30+ frames per second on unexceptional hardware.

What the FFT algorithms do not give, however, is *interactivity* between objects and the water surface. It would be difficult, for example, to have characters wade through a stream and generate a disturbance that depends directly on the motion that the player controls. A jet ski thrashing about in the water would not generate turbulent waves. Waves in a bathtub cannot bounce back and forth using FFT based simulation. And in general, it is not possible in the FFT approach to place an arbitrary object in the water and have it interact in a realistic way with the surface without substantial loss of frame rate. For practical purposes, wave surfaces are restricted in the ways that the height data can be modified within a frame and between frames.

This chapter provides a new method, which has been dubbed *iWave*, for computing water surface wave propagation that overcomes this limitation. The three scenarios for the stream, jet ski, and bathtub are handled well with iWave. Objects with any shape can be present on the water surface and generate waves. Waves that approach an object reflect off of it realistically. The entire iWave algorithm amounts to a two-dimensional convolution and some masking operations – both suitable for hardware acceleration. Even without hardware assistance, a software-only implementation is capable of simulating a 128x128 water surface height grid at over 30 fps on GHz processors. Larger grids will of course slow the frame rate down, and smaller grids will speed it up – the speed is directly proportional to the number of grid points. And because the method avoids FFTs, it is highly manipulative and suitable for a wide range of possible applications.

Linear Waves

Lets begin with a quick reminder of the equations of motion for water surfaces waves. An excellent resource for details on the fluid dynamics is [Kinsman84]. The equations that are appropriate here are called the “linearized Bernoulli’s equation.” The form of this equation we use here has a very strange operator that will be explained to some degree. The equation is [Tessendorf02]

$$\frac{\partial^2 h(x,y,t)}{\partial t^2} + \Gamma \frac{\partial h(x,y,t)}{\partial t} = -g\sqrt{-\nabla^2}\, h(x,y,t) \qquad (1)$$

In this equation $h(x,y,t)$ is the height of the water surface with respect to the mean height at the horizontal position (x,y) at time t. The first term on the left is the vertical acceleration of the wave. The second term on the left side, with the constant Γ, is a velocity damping term, not normally a part of the surface wave equation, but which is useful sometimes to help suppress numerical instabilities that can arise. The term on the right side comes from a combination of mass conservation and the gravitational restoring force. The operator

$$\sqrt{-\nabla^2} \equiv \sqrt{-\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2}}$$

is a mass conservation operator, and we will refer to it as a vertical derivative of the surface. Its effect is to conserve the total water mass being displaced. When the height of the surface rises in one location, it carries with it a mass of water. In order to conserve mass, there is a region of the surface nearby where the height drops, displacing downward the same amount of water that is displaced upward in the first location.

The next section describes how to evaluate the right hand side of (1) by expressing it as a convolution. Throughout the rest of this chapter the height is computed on a regular grid, as shown in Figure 1. The horizontal position (x,y) becomes the grid location (i,j) at positions $x_i = i\Delta$ and $y_j = j\Delta$, with the grid spacing Δ the same in both directions. The indices run $i = 1,...,N$ and $j = 1,...,M$.

Vertical Derivative Operator

Like any linear operator acting on a function, the vertical derivative can be implemented as a convolution on the function it is applied to. In this section we build up this convolution, applied to height data on a regular grid. We also determine the best size of the convolution and compute the tap weights.

As a convolution, the vertical derivative operates on the height grid as

$$\sqrt{-\nabla^2}\, h(i,j) = \sum_{k=-P}^{P} \sum_{l=-P}^{P} G(k,l)\, h(i+k, j+l) \qquad (2)$$

The convolution kernel is square, with dimensions $(2P+1)\times(2P+1)$, and can be precomputed and stored in a lookup table prior to start of the simulation. The choice of the kernel size P affects both the speed and the visual quality of the simulation.

The choice $P = 6$ is the smallest value that gives clearly water-like motion. This issue is examined more below.

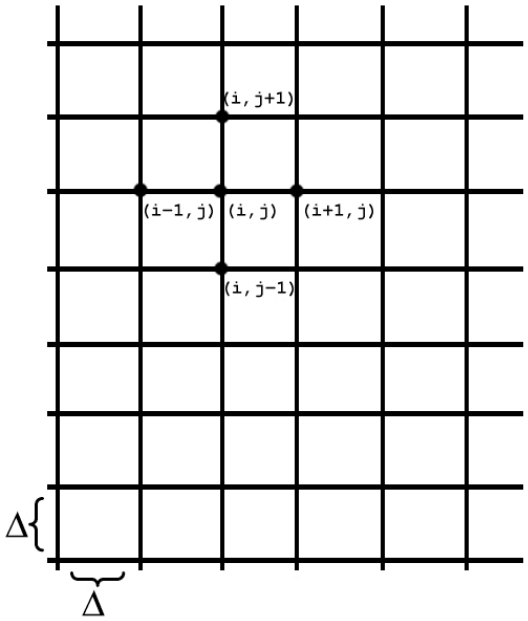

Figure 1. Layout of the grid for computing wave height.

Figure 2 shows the kernel elements $G(k,0)$ as a function of k. The two dashed vertical lines are at the spots k = 6 and k = -6. You can see from the plot that at larger values of k, the kernel is pretty much zero, and including values of k outside the dashed lines will not contribute much to the convolution. If we stop the convolution at smaller values, say k=5 and k=-5, evaluating the convolution is faster, but we will miss some small contribution from the k=6, k=-6 terms. Experience shows that you can get really good looking waves keeping the terms out to 6, but if you are pressed for computation time, stopping the convolution short of that can work also, just not be as visually realistic. Terminating the kernel at a value $|k| < 6$ sacrifices significant amounts of oscillation. This analysis is why the choice $P = 6$ is recommended as the best compromise for reasonable wave-like simulation.

Computing the kernel values and storing them in a lookup table is a relatively straightforward process. The first step is to compute a single number that will scale the kernel so that the center value is one. The number is

$$G_0 = \sum_n q_n^2 \exp(-\alpha q_n^2)$$

For this sum, $q_n = n\Delta q$ with $\Delta q = 0.001$ being a good choice for accuracy, and $n = 1,\ldots,10000$. The factor α makes the sum converge to a reasonable number, and the choice $\alpha = 1$ works well. With this number in hand, the kernel values are

$$G(k,l) = \sum_n q_n^2 \exp(-q_n^2) J_0(q_n r) \Big/ G_0$$

with the parameter $r = \sqrt{k^2 + l^2}$. The computation time for the kernel elements is relatively small, and all of the cost is an initialization – once the elements are computed they are fixed during the simulation.

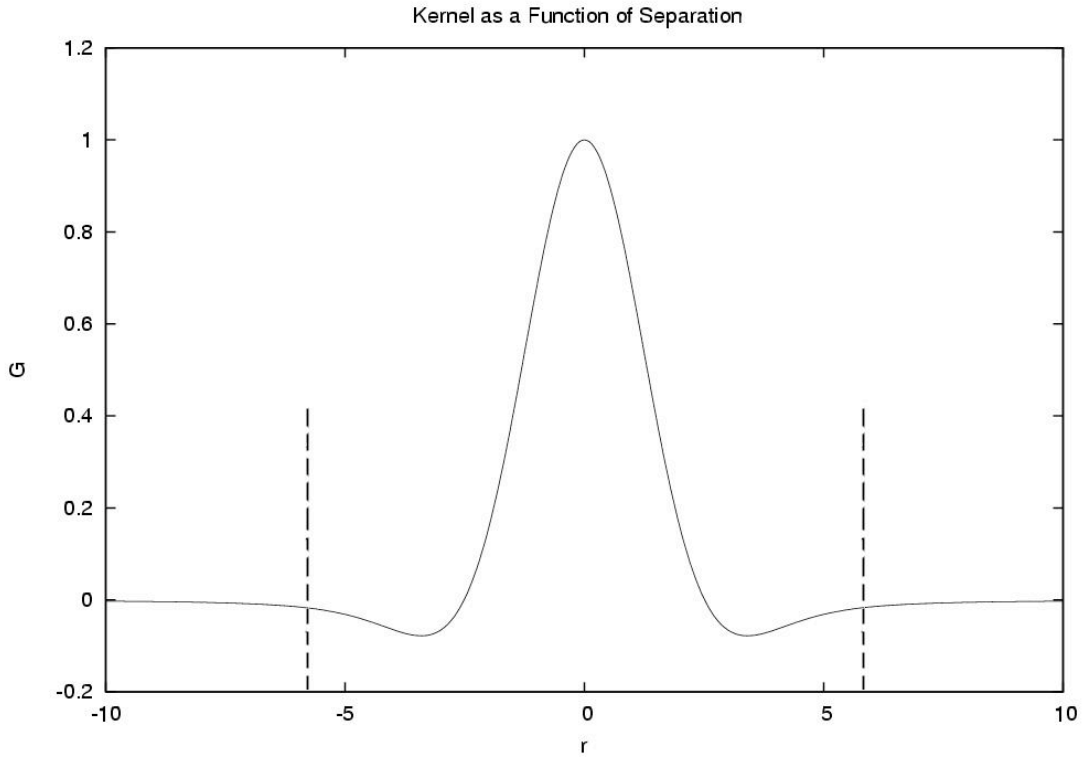

Figure 2. The vertical derivative kernel in cross section. Between the two dashed lines is the P=6 region.

One remaining item needed to compute the convolution kernel elements is a formula for the Bessel function $J_0(x)$. This is included in the C standard math library as j0. If you do not have accesses to this, a very convenient approximate fit for this function is provided in [Abramowitz72]. Although the formula there is a fitted parametric form, it is accurate to within single precision needs, and works well for the purposes of this simulation.

When you perform the convolution at each time step, there are opportunities to optimize its speed, both for particular hardware configurations and in software. Software optimizations follow because of two symmetries in the convolution kernel: The kernel is rotation symmetric, i.e. $G(k,l) = G(l,k)$, and the kernel is reflection symmetric about both axes, i.e. $G(k,l) = G(-k,-l) = G(k,-l) = G(-k,l)$. Without applying any symmetries, evaluating the convolution in equation (2) directly requires

$(2P+1)^2$ multiplications and additions. Applying these symmetries, the convolution can be rewritten as (using the fact that $G(0,0)=1$ by construction)

$$h(i,j)+\sum_{k=0}^{P}\sum_{l=k+1}^{P}G(k,l)\left(h(i+k,j+l)+h(i-k,j-l)+h(i+k,j-l)+h(i-k,j+l)\right)$$

In this form, there are still $(2P+1)^2$ additions, but only $(P+1)P/2$ multiplications.

Hardware optimizations of the convolution are possible for because this kind of convolution can be cast in a form suitable for a SIMD pipeline, so graphics cards and DSPs can execute this convolution efficiently.

Wave Propagation

Now that we are able to evaluate the vertical derivative on the height grid, the propagation of the surface can be computed over time. It is simplest to use an explicit scheme for time stepping. Although implicit methods can be more accurate and stable, they are also slower. Since we are solving a linear equation in this chapter, an explicit approach is fast and stable when the friction term is used and time step sizes can be what is needed for the display frame rate. In practice, the friction can be kept very low, although for game purposes it may be preferable to have the waves dissipate when they are no longer driven by sources.

To construct the explicit solution, the time derivatives in equation (1) must be written as finite differences. The second derivative term can be built as a symmetric difference, and the dissipative friction term as a forward difference. Rearranging the results terms, and assuming a time step Δt, the height grid at the next time step is

$$\begin{aligned} h(i,j,t+\Delta t) = {} & h(i,j,t)\frac{2-\gamma\Delta t}{1+\gamma\Delta t} - h(i,j,t-\Delta t)\frac{1}{1+\gamma\Delta t} \\ & - \frac{g\Delta t^2}{1+\gamma\Delta t}\sum_{k=-P}^{P}\sum_{l=-P}^{P}G(k,l)h(i+k,j+l,t) \end{aligned} \tag{3}$$

In terms of data structures, this algorithm for propagation can be run with three copies of the heightfield grid. For this discussion, the grids are taken to be float arrays `height`, `vertical_derivative`, and `previous_height`. During the simulation, the array `height` always holds the new height grid, `previous_height` holds the height grid from the previous time step, and `vertical_derivative` holds the vertical derivative of the height grid from the previous time step. Before simulation begins, they should all been initialized to zero for each element. The pseudo-code to accomplish the propagation is

```
float height[N*M];
```

```
float vertical_derivative[N*M];
float previous_height[N*M];

// ... initialize to zero ...

// ... begin loop over frames ...

// --- This is the propagation code ---
// Convolve height with the kernel
// and put it into vertical_derivative
Convolve( height, vertical_derivative );

float temp;
for(int k=0;k<N*M;k++)
{
   temp  = height[k];
   height[k] = height[k]*(2.0-
                          alpha*dt)/(1.0+alpha*dt)
             - previous_height[k]/(1.0+alpha*dt)
             - vertical_derivative[k]
                *g*dt*dt/(1.0+alpha*dt);
   previous_height[k] = temp;
}
// --- end propagation code ---

// ... end loop over frames ...
```

The quantities in `vertical_derivative` and `previous_height` could be useful for embellishing the visual look of the waves. For example, a large value in `vertical_derivative` indicates strong gravitational attraction of the waves back to the mean position. Comparing the value in `previous_height` with that in `height` at the location of strong `vertical_derivative` can determine roughly whether the wave is at a peak or a trough. If it is at a peak, a foam texture could be used in that area. This is not a concrete algorithm grounded in physics or oceanography, but just a speculation about how peaks of the waves might be found. The point of this is simply that the two additional grids `vertical_derivative` and `previous_height` could have some additional benefit in the simulation and rendering of the wave height field beyond just the propagation steps.

Interacting Obstructions and Sources

Up to this point we have built a method to propagate waves in a water surface simulation. While the propagation involves a relatively fast convolution, everything we have discussed could have been accomplished just as efficiently (possibly more efficiently) with a FFT approach such as the ones mentioned in the introduction. The

real power of this convolution method is the ease with which some additional 2D processing can generate highly realistic interactions between objects in the water, and can pump disturbances into the water surface.

The fact that we can get away with 2D processing to produce interactivity is, in some ways, a miracle. Normally in a fluid dynamic simulation the fluid velocity on and near a boundary is reset according to the type of boundary condition and requires understanding of geometric information about the boundary such as its outward normal. Here we get away with effectively none of that analysis, which is critical to the speed of this approach.

Sources

One way of creating motion in the fluid is to have sources of displacement. A source is represented as a 2D grid $s(i, j)$ the same size and dimensions as the height grid. The source grid should have zero values where ever no additional motion is desired. At locations in which the waves are being "poked" and/or "pulled", the value of the source grid can be position or negative. Then, just prior to propagation step in equation 3, the height grid is updated $h(i, j) += s(i, j)$. Since the source is an energy input per frame, it should change over the course of the simulation, unless a constant build up of energy is really what is wanted. An impulse source generates a ripple.

Obstructions

Obstructions are shockingly easy to implement in this scheme. An additional grid for obstructions is filled with float values, primarily with two extreme values. This grid acts as a mask delineating where obstructions are present. At each grid point, if there is no obstruction present, then the value of the obstruction grid at that point is 1.0. If a grid point is occupied by an obstruction, then the obstruction grid value is 0.0. At grid points on the border around an obstruction, the value of the obstruction grid is some intermediate value between 0.0 and 1.0. The intermediate region acts as an anti-aliasing of the edge of the obstruction.

Given this obstruction mask, the obstruction's influence is computed simply by multiplying the height grid by the obstruction mask, so that the wave height is forced to zero in the presence of the obstruction, and left unchanged in areas outside the obstruction. Amazingly, that is all that must be done to properly account for objects on the water surface! This simple step causes waves that propagate to the obstruction to reflect correctly off of it. It also produces refraction of waves that pass through a narrow slit channel in an obstruction. And it permits the obstruction to have any shape at all, animating in any way that the user wants it to.

Combining the source and obstruction, the pseudo-code for the application of these is:

```
float source[N*M], obstruction[N*M];
// ... set the source and obstruction grids
```

```
for(int k = 0; k < N*M; k++)
{
   height[k] += source[k];
   height[k] *= obstruction[k];
}

// ... now apply propagation
```

Wakes

Wakes from moving objects are naturally produced by the iWave method of interactivity. In this special case, the shape of the obstruction is also the shape of the source. Setting `source[k] = 1.0-obstruction[k]` works as long as there is an anti-aliased region around the edge of the obstruction. With this choice, moving an obstacle around in the grid produces a wake behind it that can include the V-shaped Kelvin wake. It also produces a type of stern wave and waves running along the side of the obstacle. The details of the shape, timing, and extent of these wake components are sensitive to the shape and motion of the obstacle.

Ambient Waves

The iWave method is not very effective at generating persistent large scale wave phenomena like open ocean waves. If the desired application is the interaction of objects with “ambient waves” that are not generated in the iWave method, there is an additional procedure to follow to generate that interaction without explicitly simulating the ambient waves.

The ambient waves consist of a height grid that has been generated by some other procedure. For example, FFT methods could be used to generate ocean waves and put them in a height grid. Since we are only trying to compute the interaction of the ambient waves with an obstruction, the ambient waves should not contribute to the simulation outside the region of the obstruction. The pseudo-code for modifying the height grid, prior to propagation and just after application of obstructions and sources as above, is

```
float ambient[N*M];

// ... set the ambient grid for this time step

// ... just after the source and obstruction, apply:
for(int k = 0; k < N*M; k++)
{
   height[k] -= ambient[k]*(1.0-obstruction[k]);
}
```

```
// ... now apply the propagation
```

With this, ambient waves of any character can interact with objects of any animating shape.

Grid Boundaries

Up to this point we have ignored the problem of how to treat the boundaries of the grid. The problem is that the convolution kernel requires data from grid points a distance P in all four directions from the central grid point of the convolution. So when the central grid point is less than P points from a boundary of the grid, missing data must be filled in with some sort of criterion. There are two types of boundary conditions that are fairly easy to apply: periodic and reflecting boundaries.

Periodic Boundaries

In this situation, a wave encountering a boundary appears to continue to propagate inward from the boundary on the opposite side. In performing the convolution near the boundaries, the grid coordinates in equation 3 $i+k$ and $j+l$ may be outside of the ranges $[0, N-1]$ and $[0, M-1]$. Applying the modulus `(i+k)%N` is guaranteed to be in the range $[-N+1, N-1]$. To insure that the result is always positive, a double modulus can be used: `((i+k)%N + N)%N`. Doing the same for the $j+l$ coordinate insures that periodic boundary conditions are enforced.

Reflecting Boundaries

Reflecting boundaries turn a wave around and send it back into the grid from the boundary the wave is incident on, much like a wave that reflects off of an obstacle in the water. If the coordinate $i+k$ is greater than $N-1$, then it is changed to $2N-i-k$. If the coordinate is less than 0, it is negated, i.e. it becomes $-i-k$, which is positive. An identical procedure should also be applied to the $j+l$ coordinate.

To efficiently implement either of these two types of boundary treatments, the fastest approach is to divide the grid into 9 regions

1. The inner portion of the grid with the range of coordinates $i \in [P, N-1-P]$ and $j \in [P, M-1-P]$.
2. The right hand side $i \in [N-P, N-1]$ and $j \in [P, M-1-P]$.
3. The left hand side $i \in [0, P-1]$ and $j \in [P, M-1-P]$.
4. The top side $i \in [P, N-1-P]$ and $j \in [0, P-1]$.
5. The bottom side $i \in [P, N-1-P]$ and $j \in [M-P, M-1]$.
6. The four corners that remain.

Within each region the particular boundary treatment required can be coded efficiently without conditionals or extra modulus operations

Surface Tension

So far the type of simulation we have discussed is the propagation of gravity waves. Gravity waves dominate surface flows on scales of approximately a foot or larger. On smaller scales the character of the propagation changes to include surface tension. Surface tension causes waves to propagate faster at smaller spatial scales, which tends to make the surface appear to be more rigid than without it. For our purposes, surface tension is characterized by a length scale L_T, which determines the maximum size of the surface tension waves. The only change required of our procedure is a different computation of the convolution kernel. The kernel calculation becomes

$$G(k,l) = \sum_n q_n^2 \sqrt{1 + q_n^2 L_T^2} \exp(-q_n^2) J_0(q_n r) \Big/ G_0$$

Other than this change, the entire iWave process is the same.

Conclusion

The iWave method of water surface propagation is a very flexible approach to creating interactive disturbances of water surfaces. Because it is based on 2D convolution and some simple 2D image manipulation, high frame rates can be obtained even in a software-only implementation. Hardware acceleration of the convolution should make iWave suitable for many game platforms. The increased interactivity of the water surface with objects in a game could open new areas of game play that previously were not available to the game developer.

References

[Abramowitz72] Milton Abramowitz and Irene A. Stegun, Handbook of Mathematical Functions, Dover, 1972. Sections 9.4.1 and 9.4.3. http://members.fortunecity.com/aands/page_369.htm

[Arete03] Arete Entertainment. http://www.areteis.com

[Jensen01] Lasse Jensen, on-line tutorial, 2001. http://www.gamasutra.com/gdce/jensen/jensen_01.htm

[Kinsman84] Blair Kinsman, *Wind Waves*, Dover, 1984.

[Tessendorf02] Jerry Tessendorf, "Simulating Ocean Water," *Simulating Nature*, Siggraph Course Notes, 2002. http://home1.gte.net/tssndrf/index.html