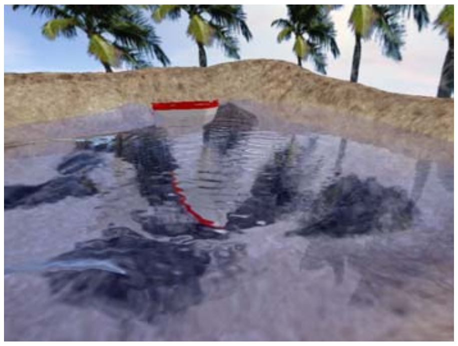
SDK White Paper

Vertex Texture Fetch Water

October 19, 2004

Abstract

This document describes a method to simulate and render small to medium bodies of water on the GPU. The simulation involves integrating the 2D wave equation with a pixel shader. The simulation result is converted into geometry using Vertex Texture Fetch(VTF) in a vertex shader. Screen-space reflection and refraction maps are combined using a Fresnel reflection term to give the water a realistic appearance.



Jeremy Zelsnack

sdkfeedback@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

The goal of the water simulation is to produce a realistic looking water surface with reasonable computation time. One potential way to simulate water would involve solving the Navier-Stokes equations for incompressible fluids. This would provide very realistic fluid mechanics for the water. Unfortunately, it is a fairly heavy-weight computation. A simpler and less costly way to simulate water is to solve the 2D Wave Equation on a uniform grid of points. This trades some realism for simplicity and speed. This paper describes the 2D Wave Equation technique.

$$\frac{\partial^2 y}{\partial t^2} = c^2 \left(\frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 y}{\partial z^2}\right)$$

Equation 1: 2D Wave Equation

The 2D Wave Equation is listed as Equation 1. The equation as written dictates that points will undulate up and down in the **y**-dimension. The **c** term is the speed that the wave travels. For a little intuition, the equation basically says that the acceleration of a point up and down is proportional to how quickly the steepness of the surface is changing.

At this point, you might be asking, "how does one solve such an equation?" The GPU doesn't really have a good understanding of what a partial derivative is; it really only directly understands spatial positions. If we integrate the left side of the 2D Wave Equation twice with respect to **t**, we end up with **y** (Equation 2). If we combine the computed **y** for a point with the known **x** and **z** positions of the point, we end up with a 3D position; this is something the GPU can understand.

$$\int \frac{\partial^2 y}{\partial t^2} dt = \frac{\partial y}{\partial t}$$

$$\int \frac{\partial y}{\partial t} dt = y$$

Equation 2: Integrating the 2D Wave Equation

The next stumbling block is figuring out how to integrate the right hand side of the equation. There are many numerical integration techniques to choose from. Each has its advantages and disadvantages. For our application, we want stability and speed. Stability is important because this differential equation is fairly stiff. The **c²** term on the right hand side of the equation lends itself to large values on the left hand side. This rules out simple integration techniques like Euler because of its instability. One could use a more robust integration technique like Runge-Kutta integration. Runge-Kutta has the draw-back that it involves computing intermediate terms and requires explicit storage of velocity. This is too expensive and complex for this application. Verlet integration is computationally inexpensive, stable and does not require the explicit storage of velocity. Unfortunately, I forgot the Verlet equation and couldn't find a reference after 30 seconds of searching, so I recreated something similar to Verlet.

If the only information you have about a particle is its position at t=0 [**p(t₀)**], position at t=1 [**p(t₁)**], acceleration at t=1 [**a(t₁)**], some basic high school physics knowledge and a desire to compute the position at t=2 [**p(t₂)**] you can arrive at Equation 3. The derivation assumes that a constant step size is used; thus $\mathbf{t_1 - t_0 = t_2 - t_1 = t_{n+1} - t_n}$. A non-constant step size could be used, but it muddies the algebra.

$$v(t_1) = \frac{p(t_1) - p(t_o)}{t_1 - t_0}$$

$$p(t_2) = p(t_1) + v(t_1)\cdot(t_2 - t_1) + \frac{1}{2}\cdot a(t_1)\cdot(t_2 - t_1)^2$$

$$p(t_2) = p(t_1) + \frac{p(t_1) - p(t_o)}{t_1 - t_0}\cdot(t_2 - t_1) + \frac{1}{2}\cdot a(t_1)\cdot(t_2 - t_1)^2$$

$$h = (t_2 - t_1) = (t_1 - t_0)$$

$$p(t_2) = p(t_1) + \frac{p(t_1) - p(t_o)}{h}\cdot h + \frac{1}{2}\cdot a(t_1)\cdot h^2$$

$$p(t_2) = p(t_1) + p(t_1) - p(t_o) + \frac{1}{2}\cdot a(t_1)\cdot h^2$$

$$p(t_2) = p(t_1) + p(t_1) - p(t_o) + \frac{1}{2}\cdot a(t_1)\cdot h^2$$

$$p(t_2) = 2\cdot p(t_1) - p(t_o) + \frac{1}{2}\cdot a(t_1)\cdot h^2$$

Equation 3: Integration Derivation

Equation 3 is very similar to the actual Verlet integration technique (Equation 4). It differs by a factor of 2 on the acceleration term. In the hacky world of game development, what's a factor of 2 between friends? Especially considering that there's always going to be some tweaking to make things look "right".

$$p(t_2) = 2\cdot p(t_1) - p(t_o) + a(t_1)\cdot h^2$$

Equation 4: Verlet Integration Equation

So Equation 3 describes a technique for numerical integration. There's one problem with it, it's not as stable as it could be and it's jittery (nothing ever comes to a rest). It needs a little bit of dampening to absorb some excess energy that results from the inexact numerical integration. This can be accomplished by reducing the inferred velocity by a constant factor (Equation 5).

$$p(t_2) = p(t_1) + \alpha \cdot v(t_1) \cdot h + \frac{1}{2} \cdot a(t_1) \cdot h^2$$

$$p(t_2) = p(t_1) + \alpha \cdot [p(t_1) - p(t_o)] + \frac{1}{2} \cdot a(t_1) \cdot h^2$$

$$p(t_2) = (1+\alpha) \cdot p(t_1) + \alpha \cdot p(t_o) + \frac{1}{2} \cdot a(t_1) \cdot h^2$$

$$\alpha = 0.99$$

$$p(t_2) = 1.99 \cdot p(t_1) + 0.99 \cdot p(t_o) + \frac{1}{2} \cdot a(t_1) \cdot h^2$$

Equation 5: Integration with Dampening

Now that we have an integration technique, we can almost integrate the right hand side of the 2D wave equation. There's the final remaining problem of figuring out how to compute the partial derivatives on the right hand side of the wave equation. One way can do this is by fitting a surface to the simulation points. With a fitted surface, we can analytically derive an expression for the partial derivatives.

So what kind of surface should we fit to the simulation grid points? We want something that is fast, simple and reasonably accurate. A collection of cubic splines meets our needs. We will simplify things by separating the problem into two 2D cubic splines. This leads us to Equation 6. With a cubic equation, we have to specify four constraints. Three obvious constraints are the position of the point and the positions of the neighboring points. The fourth constraint will come from forcing the derivative at the point to be parallel to the neighboring points. Figure 1 depicts the situation. Assuming that the derivative is parallel to the neighbor points and that we're on a uniform unit grid, we can derive an equation for the 2nd partial derivatives (Equation 7). Using this partial derivative, the right hand side of the 2D wave equation becomes Equation 8.

$$f(x) = c_0 \cdot x^3 + c_1 \cdot x^2 + c_2 \cdot x + c_3$$
$$f'(x) = 3 \cdot c_0 \cdot x^2 + 2 \cdot c_1 \cdot x + c_2$$
$$f''(x) = 6 \cdot c_0 \cdot x + 2 \cdot c_1$$

Equation 6: Cubic Equation and Derivatives

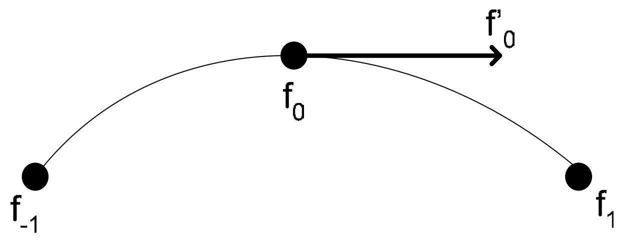


Figure 1: 2D Cubic Representation

$$
\begin{aligned}
& f(x) = c_0 \cdot x^3 + c_1 \cdot x^2 + c_2 \cdot x + c_3 \\
& f'(x) = 3 \cdot c_0 \cdot x^2 + 2 \cdot c_1 \cdot x + c_2 \\
& f''(x) = 6 \cdot c_0 \cdot x + 2 \cdot c_1 \\
& f(-1) = f_{-1} \\
& f(0) = f_0 \\
& f(1) = f_1 \\
& f'(0) = \frac{1}{2} \cdot (f_1 - f_{-1}) \\
& f(-1) = -c_0 + c_1 - c_2 + c_3 = f_{-1} \\
& f(0) = c_3 = f_0 \\
& f(1) = c_0 + c_1 + c_2 + c_3 = f_1 \\
& f'(0) = c_2 = f'_0 \\
& f''(0) = 2 \cdot c_1 \\
& f_{-1} + f_1 = 2 \cdot c_1 + 2 \cdot c_3 \\
& c_1 = \frac{1}{2} \cdot (f_{-1} + f_1) - c_3 \\
& c_1 = \frac{1}{2} \cdot (f_{-1} + f_1) - f_0 \\
& f''(0) = 2 \cdot c_1 = f_{-1} + f_1 - 2 \cdot f_0
\end{aligned}
$$

Equation 7: Computing the 2nd Derivative

$$
\begin{aligned}
& c^2 (\frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 y}{\partial z^2}) = c^2 ([y_{-1,0} + y_{1,0} - 2 \cdot y_{0,0}] + [y_{0,-1} + y_{0,1} - 2]) \\
& c^2 (\frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 y}{\partial z^2}) = c^2 (y_{-1,0} + y_{1,0} + y_{0,-1} + y_{0,1} - 4 \cdot y_{0,0})
\end{aligned}
$$

Equation 8: Computing the Right Hand Side

Now that we have a simple analytical technique for calculating the right hand side of the wave equation and an integration technique, we can compute the actual **y** values for the grid points. Since we've made the assumption that the grid points are spaced at unit intervals (which probably isn't the case in world space), we need to adjust the speed of the wave so it looks "right".

Now that we have a simple iterative technique to solve the wave equation, we can have the GPU solve it for us. We store the heights of a uniform grid as a texture. We also keep track of the previous set of heights from the previous time step (also a texture). With our solving method and the height textures, we can compute the next value for the height textures. A simplified HLSL pixel shader code snippet that solves the wave equation is included below.

```
// Look up all the neighbor heights
height_x1y1 = tex2D(heightSampler, i.texCoord);
height_x0y1 = tex2D(heightSampler, i.texCoord + half2(-1/w, 0));
height_x2y1 = tex2D(heightSampler, i.texCoord + half2( 1/w, 0));
height_x1y0 = tex2D(heightSampler, i.texCoord + half2( 0,-1/h));
height_x1y2 = tex2D(heightSampler, i.texCoord + half2( 0, 1/h));

// Look up the height from the previous time step
previousHeight_x1y1 = tex2D(previousHeightSampler, i.texCoord);


// Compute the acceleration
acceleration = cSquared * (height_x0y1 + height_x2y1 +
                           height_x1y0 + height_x1y2 –
                           4.0 * height_x1y1);

// Do Verlet integration
newHeight = 2 * height_x1y1 – previousHeight_x1y1 +
            0.5 * acceleration * dt * dt;
```

The height textures are stored as 16 bit floating point (fp16) textures. 8 bit integer is inadequate. 16 bit integer precision might suffice, but it's easier and better to use floating point. The VertexTextureFetchWater SDK sample stores a combined height / normal texture in a D3DFMT_A16B16G16R16 texture. The textures are combined to reduce the number of texture fetches in the vertex shader (which are relatively costly instructions).

So now that have a height texture for the simulated water surface, what do we do with it? Shader Model 3.0 hardware supports texture fetches in the vertex shader (in VS 3.0). To render the water, we pass in a grid mesh for the water with valid **x** and **z** values. A texture fetch in the vertex shader looks up the **y** value and the normal. The vertex texture fetch effectively allows us to convert a texture into geometry (Illustrated in Figure 2). See our whitepaper on vertex textures for more details (ftp://download.nvidia.com/developer/Papers/2004/Vertex_Textures/Vertex_Textures.pdf).

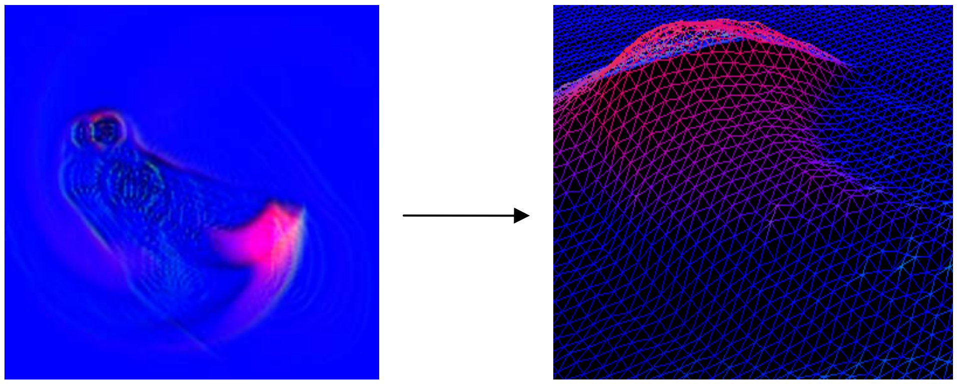


Figure 2 : Simulation Texture to Geometry

It should be noted that GeForce 6 series hardware does not support the D3DFMT_A16B16G16R16 format in vertex shaders. This requires the program to blit the fp16 wave equation solution to an fp32 texture. You might ask, why not just render the wave equation solution to an fp32 texture in the first place? The answer is that it would be slower to render and fp32 render targets do not support blending or texture filtering. The lack of fp32 blending would make the control of the water more cumbersome. The lack of fp32 texture filtering would reduce the caustic rendering quality (discussed later).

Now that we have a way to simulate the water surface, we need some way to control it. Luckily, this is very easy with Verlet integration. To make the water move, we simply have to render height displacements into the water height texture. The Verlet integration implicitly understands a velocity from the height displacement and reacts accordingly. On GeForce 6 series hardware, the height displacement is rendered directly into the height texture using fp16 blending. This is the natural and efficient way to do this. On hardware prior to the GeForce 6 series, this would not be possible due to the lack of support for fp16 alpha blending. Without blending, you would have to render height displacements to a secondary non fp16 render target. Then you would have to render a quad over the height texture to add the displacements in. This is less efficient and can suffer from precision problems.

Moving onto performance, how fast is the water simulation? It seems like a lot of work. Fortunately, for the GPU it's not much work at all. You're basically rendering to a small (let's say 128x128) render target with a simple pixel shader that does highly localized texture accesses. Intuitively, this doesn't seem like much work for a modern GPU. A little bit of benchmarking "science" reveals that the frame rate on a GeForce 6800 GT changes from 263fps to 268fps after disabling the simulation. Disregarding the poor benchmarking technique, this implies that the simulation takes .07 milliseconds; that's not long. You're more limited by the vertex texture fetch performance than the actual simulation.

Rendering

Now we have the ability to simulate the motion of the water. The next step is rendering it in a realistic manner. The simplest way to render water would be to render the surface as tinted alpha-blended geometry. This is very easy to do and very fast. Unfortunately, it looks a little too 1996.

We're going to use screen-space reflection and refraction maps to get a more realistic result. For the refraction map, assuming that the viewer is above water, we render everything that is under water in the scene. A user clip plane is used to roughly prevent geometry above the water's surface from getting rendered into the refraction map (Oblique frustum clipping can be used in place of clip planes http://download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/NearClipPlane.zip). For the reflection map, assuming that we're above water, we render everything above water after reflecting the viewer's position under the water. If the viewer is under water, the refraction map contains the above water geometry and the reflection map contains the underwater geometry. An example of these maps is included Figure 3 assuming the viewer is above the water. The astute reader will notice that the skybox is not clipped with the water surface plane. This is done to avoid unsightly artifacts introduced by the fact that the water's surface is not actually a plane.

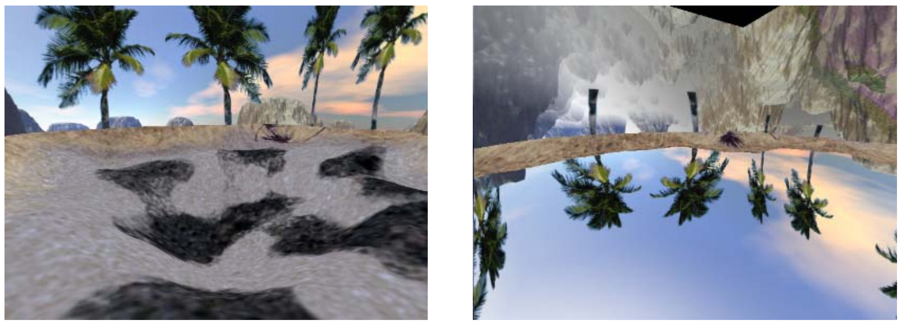


Figure 3 : Refraction and Reflection Maps

After we have rendered reflection and refraction maps, how do we use them? The first thing we need to compute are texture coordinates for the water vertices. If we just directly take the screen spaces positions of the vertices and use them as texture coordinates (with a little scaling and biasing), we get something that just looks like we rendered the scene directly (if we just use the refraction map). We need to take into account that when light goes from one medium to another medium it refracts.

Based upon the view vector, the surface normal of the water, some shaky assumptions about the geometry below the water and the indices of refraction for water and air, we can compute refraction map texture coordinates (Figure 4). Plugging in the view vector, normal and index of refraction ratios into the HLSL refract() function we get the refraction vector. Assuming that we know next to nothing about the geometry below the water's surface, we can just walk a fixed length along the refraction vector. We compute our refraction map texture coordinate by projecting this point back into screenspace.

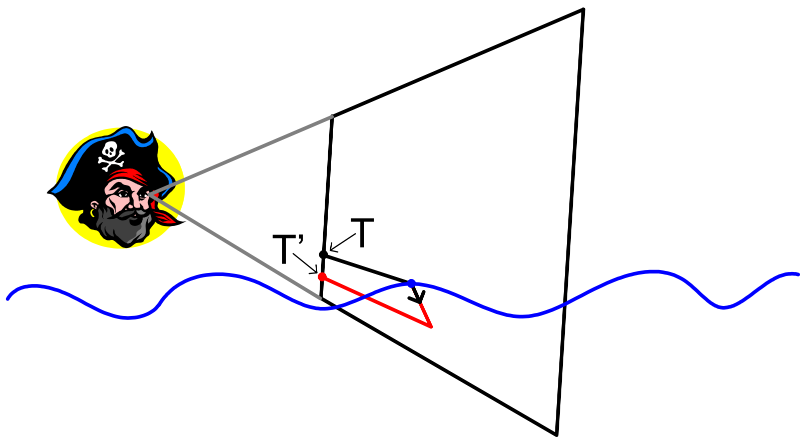


Figure 4 : Texture Coordinate Generation

The reflection map texture coordinates are computed in a similar manner. The view space position is walked along the reflected ray by a constant amount and then re-projected back into screen space.

As just discussed, underwater and above water geometry are rendered to the refraction and reflection maps (respectively assuming an above water viewer). We run into problems with geometry that is inter-penetrating the water's surface. If we use perturbed texture coordinate for this geometry, we see unsightly discontinuities at the water's surface. To deal with this, geometry that inter-penetrates the water is rendered to a separate refraction map. In this demo the inter-penetrating geometry doesn't have reflections; this was done as an aesthetic choice.

Using perturbed texture coordinates, you can run into situations where more geometry is visible in the reflection or refraction than what was rendered to their respective maps. To compensate for this problem, the reflection maps and refraction maps are drawn with a larger field of view than normal. The texture coordinates are scaled down to reflect this overdraw. Sometimes, even with the overdraw, there isn't enough geometry in the refraction and reflection maps. To deal with this, the maps are faded out on the edges to eliminate very unsightly artifacts.

Now that we have nice refraction and reflection terms, how do we combine them? Do we just add them together? No, that doesn't look very natural. A better solution is to combine them using a simplified Fresnel Reflection Term (http://developer.nvidia.com/object/fresnel_wp.html). The Fresnel Reflection term describes how reflection and refraction are combined for a material. Basically, the more directly you look at a material, the more refraction you see. At glancing angles, you see more reflection. Figure 5 demonstrates the Fresnel Reflection Term. On the left hand side, the viewer is at a glancing angle to the surface so the viewer will see a lot of reflection (white is 1). On the right hand side, the viewer is looking more directly through the water surface so she will mostly see refraction (black is 0).

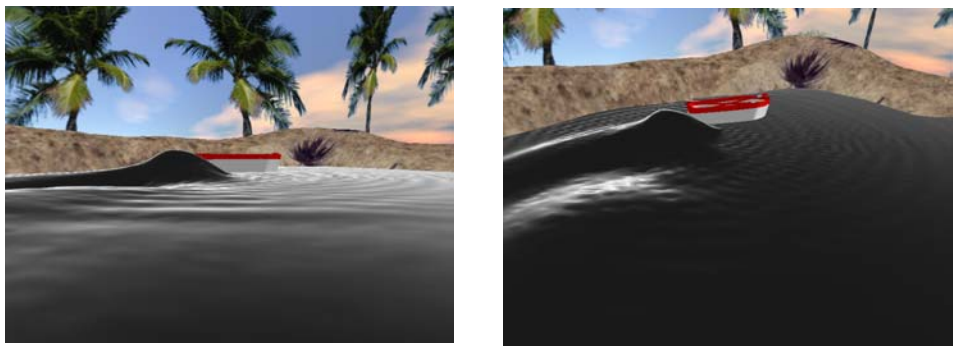


Figure 5 : Fresnel Reflection Term

To enhance the underwater rendering, a simple caustics approximation was used. Anybody who has been underwater in a pool has noticed the undulating light on the bottom and sides of the pool. The correct simulation of caustics would be very difficult and slow on the GPU. We use a very rough approximation that gets a reasonable result. We assume that all the light hitting the bottom is directly refracted through the surface of the water. We use the water surface normal to look at how much sun we can see. On the GeForce 6 series GPUs we can do fp16 texture filtering so the caustics look smooth. Without fp16 texture filtering, the caustics look blocky (Figure 6).

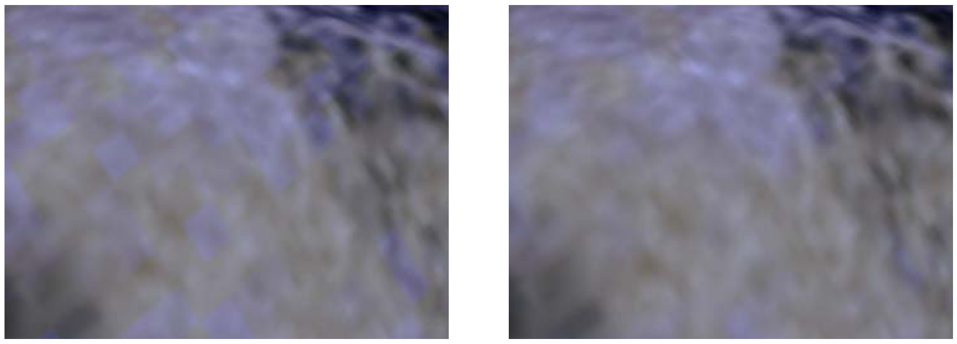


Figure 6 : Caustics Rendering Quality

Another phenomenon that anybody who has been underwater has experienced is total internal reflection. At a certain critical angle when going from a material with a higher to lower index of refraction (from water to air), the solution to Snell's Law (Equation 9) has no real solution. Since imaginary light is fairly difficult to see, you no longer see any refraction; hence you have total internal reflection. Total internal reflection is approximated by blending out the refraction term based upon the angle between the viewer and the water's surface.

$$n_i \cdot \sin(\theta_i) = n_r \cdot \cos(\theta_i)$$

Equation 9 : Snell's Law

Improvements

The Verlet integration in this sample was implemented for clarity. A more realistic integrator would probably have to take into account variable time steps (unless you have a constant physics frequency). A variable time step involves re-solving the integration equation (Equation 3) without the assumption that the time steps are the same size.

As for performance improvement suggestions, you could elect to only render refraction and reflection maps when the viewer moves. You could also render lower resolution reflection and refraction maps when the viewer is moving; when the viewer stops you can render the full resolution maps. It would also be an option to render only important geometry at low resolution LODs to the refraction and reflection maps when the viewer is moving. When running at a high frame rate, you have the option to render the refraction and reflection maps every N frames. This will result in objectionable tearing at lower frame rates or when very rapid viewer movement is occurring.

This technique could be extended to larger bodies of water. You could keep a few water simulations active depending on the viewer's position. When the viewer moves beyond a certain distance from a patch of water, the water could reduce its simulation frequency or stop simulation entirely. The technique could also extend to large bodies by using a non-uniform simulation grid. Suppose you are on an island, you could use high grid detail near the shore. Farther away from the shore, you could use a lower density mesh. This would require revisiting the Verlet integration equations (Equation 3) and throwing away the assumptions about the uniform grid.

The technique could have a fallback for non Shader Model 3.0 hardware by simply not doing the vertex texture fetch; the water's surface would remain flat. The texture coordinate generation is similar to virtual displacement mapping or offset mapping or parallax mapping or whatever you want to call it, so it would probably still have a reasonable appearance.

Bibliography

Dudash, B., "Custom Clip Plane."

http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html#NearClampPlane

Gerasimov, G., Fernando, F., Green, S. "Shader Model 3.0 Using Vertex Textures."

ftp://download.nvidia.com/developer/Papers/2004/Vertex_Textures/Vertex_Textures.pdf

Guardado, J., Sanchez-Crespo, D. (2004). Rendering Water Caustics. In R. Fernando, GPU Gems: Programming Techniques, Tips And Tricks, Ch. 2. Aw Professional.

Wloka, M., "Fresnel Reflection." http://developer.nvidia.com/attach/6664

The collective consciousness of computer graphics.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com