Rendering Ocean Water in OpenGL

Ashwin Chetty

In this paper, I describe the methods which I used to perform a real-time OpenGL/GLSL-based ocean-water rendering. This rendering constructs a 3D ocean surface perturbed with Gerstner (trochoidal) waves, quickly approximates a normal-aware surface-reflection calculation, and allows support for arbitrarily large bodies of water.

Wave Model

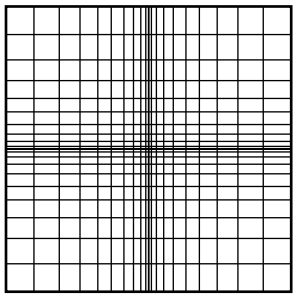
Gerstner waves describe a relatively simple (though visually robust) solution to the wave-equation, and provide a computationally easy method to approximate the surface of an ocean. Tessendorf describes the equations for Gerstner wave simulation as, for a particular world-location x_o:

$$x' = x_0 - \left(\frac{\boldsymbol{k}}{k}\right) A \cdot \sin(\boldsymbol{k} \cdot x_0 - \omega t)$$

$$y' = A \cdot \cos(\boldsymbol{k} \cdot x_0 - \omega t)$$

The precise equations are less important for our discussion than the fact that computation of these waves can be computed very quickly. All Gerstner-wave perturbations were performed within the vertex shader. A surface consisting of tessellated flat triangles were passed to the vertex shader, which then perturbed each point according to the equations given above. Future work may reasonably offload the tessellation from the CPU into the GPU.

LOD



The effective resolution of rendered geometry decreases at a rate directly proportional to its distance from the camera. This simulation exploits this fact by reducing the density of heightmap vertices linearly with respect to distance from the camera. The set of polygons passed to the vertex shader is of a similar

form to what is presented in the figure to the right. For given horizontal and vertical resolutions w and h, the set of division points is given by

$$p_x = \frac{1}{2} + \frac{2}{w^2} \cdot |\frac{x-w}{2}|(\frac{(x-w)}{2})$$

$$p_y = \frac{1}{2} + \frac{2}{h^2} \cdot |\frac{y-h}{2}|(\frac{(y-h)}{2})$$

with $x \in [0, \dots, w], y \in [0, \dots, h]$.

During rendering, the position of the water mesh is always kept at the same *xz* point as the player, to ensure that the closest point to the player has the greatest density. Points are perturbed appropriately to account for the movement of player relative to water.

Shader Pipeline

Rendering proceeds in two passes. First, all non-reflective world geometry (ie. excluding the water surface) is rendered to a framebuffer. The camera's world-position is reflected along the upward-axis in order to allow for rendering of world reflections. The resultant RGB/D images are then passed to a renderer to render the water surface; and the rest of the world geometry is rendered again.

Reflected Geometry

After a first pass renders geometry reflected along the vertical axis, we must determine the pixel color of each point on the water's surface. For each pixel position $p \in \mathbb{R}^3$, this calculation is performed by tracing a ray between the camera and p, reflecting the ray across the water's surface by a predetermined amount (in the simulation, it is 5 units), and projecting this reflected point into screen-space in order to determine the particular texture coordinate to look up. Initial attempts involved using the z-buffer depth (instead of the arbitrary *5*) to approximate the distance between ocean-surface and reflected geometry. However, this created extremely noticeable artifacts with the stark contrasts in z-buffer depth between skybox and world objects.

A diffuse and specular Phong shading calculation is then performed to determine pixel color. Future work would involve modulating the extent of reflections based on the angle from which the viewing the water point.

Additional Techniques

Water surface height gradually reduces to 0 near the horizon in order to simulate the curvature of the Earth.

A linear fog is applied to world geometry, water surface, and skybox, in order to reducing the jarring color changes that may occur near the horizon. Fog is also consistent with ambient atmospheric conditions.

Conclusion

The resulting rendering engine is able to create a visually appealing real-time water simulation with support for reflections and volumetric waves. Water refraction is less difficult to implement than reflection, and may be added next. Rendering artifacts with objects which intersect the water plane may also be visually occluded with the addition of refractions.

*The code for this rendering is available at https://github.com/palebluepixel/submarineRacers, on the **water** branch.*

Works Cited:

[1] Tessendorf, Jerry. "Simulating ocean water." *Simulating nature: realistic and interactive techniques. SIGGRAPH* 1.2 (2001): 5.

[2] Mitchell, Jason L. "Real-time synthesis and rendering of ocean water." *ATI Research Technical Report* 4.1 (2005): 121-126.