Reflection of a Horizon Image on the Surface of Water

Chris Stuetzle* RPI
Steve Warner† RPI
John-Paul Mayer‡ RPI

Figure 1: *A beautiful sunset.*

Abstract

We have created a piece of software that renders the reflection of a ”landscape” texture on a wavy fluid surface. The fluid is represented by a height field and a quad mesh. Waves are created by using wave particles, non-physical entities that determine the influence a wave has on the fluid around it. The scene is ray traced by sending a ray through each pixel, bouncing it off the water and against the texture backdrop. The color of the texture is quieried and influences the color of the water surface. Surface normals are calulated by interpolating vertex normals across each face. The ray tracing is also parallelized, and speeds up to an aymptotic point, leveling off at the number of cores of the machine being used. We produce very attractive non-photo-realistic renderings of landscapes reflecting off a liquid surface.

Keywords: ray tracing, liquid, surface, wave particle

1 Introduction and Motivation

We have created a piece of software that will render the reflection of a bitmap image in a wavy surface of water. The software combines fluid dynamic techniques with ray tracing and texture mapping to create its images. The ray tracing is parallelized for speed considerations, and the initial push was to make the software semi-real-time, allowing the user to see the waves as they flowed. This goal proved unattainable for the time frame that we had to work with.

The homework assignment regarding fluid dynamics earlier in the semester presented a challenge that some of us felt we did not overcome to our satisfaction. It was for this reason that we began discussing possible fluid dynamics topics for our final project. We had also been very interested in ray tracing, as it provided beautiful and realistic results for reflections. It was along these lines that we began thinking about combining the two areas of study and came up with the reflection of scenery in the surface of a body of water.

As a note, most of the images in this paper (due to time constraints) were taken before the final version of the code was implemented. Only those that are considered ”results” reflect our finalized software.

*e-mail: stuetc@rpi.edu
†e-mail:warnes2@rpi.edu
‡e-mail:mayerj@rpi.edu

2 Review of Literature

Our project is based on previous work in the areas of ray tracing and water surface simulation.

2.1 Ray Tracing

We began by thinking about how we would create an adaptation of the ray tracing algorithm presented by [Whitted 1980]. His paper speaks of rays intersecting with solid objects that reflect more rays from them in the direction dictated by the normal of the surface the original ray struck. This process becomes difficult when accurately modeling light reflecting off of the surface of water. To do this quickly, the rays cannot be traced to each part of the water’s surface, especially since the shape of the surface is changing constantly. Therefore, we began to wonder how we would be able to parallelize the ray tracing algorithm. Whitted noticed that ray tracing lends itself naturally to being parallelized, and most current generation ray tracing software uses this technique. [Notkin and Gotsman 1997] describes two different methods for parallelizing ray tracing. A data-driven method, in which each element of the geometry of the scene is assigned a processor, and all computations involving that particular geometric element are handled by that processor. This method is very memory efficient, but the overhead grows considerably as you add more processors, and so it does not scale up in that way. The second method is called demand driven computation. In this method, a region of the image space is assigned to each processor, and all rays that are generated from this region are the responsibility of that processor. In this method, none of the regions must exchange data between them. There is an inherent trade-off here, as the more data-driven the simulation, the move overhead is necessary to pass data between processors. We chose to use this method for our ray tracing.

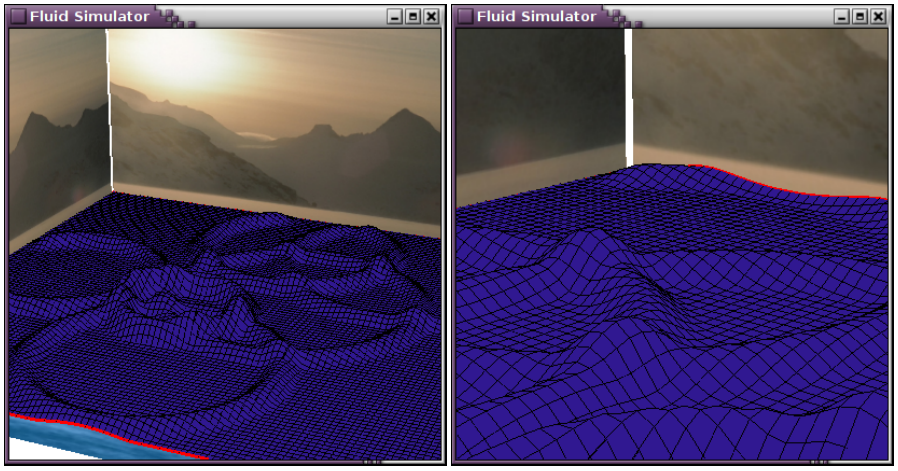

Figure 2: *Waves as they are generated and interacting with one another.*

2.2 Water Simulation

In order to simulate a wavy water surface, we use elements from the Wave Particles method by [Yuksel et al. 2007]. Their method is simple, fast, and unconditionally stable; important qualities for our project due to the additional complexity of the ray tracing component. Their method models the surface of a liquid as a mesh, set up in a grid. They introduce the use of wave particles to simulate surface water waves. The wave particles themselves are arbitrary invisible particles which travel along a plane parallel to the water's surface influencing the height field used to represent the water's surface. When two wave particles are nearby each other, their proximity results in a local height field value which is the sum of both their contributions. The end result of the contributions of all the wave particles in the scene is a realistic wave simulation at real-time speeds.

It is important to note that the wave particles method is not a physical simulation. The wave particles have no physical meaning and the method's results are only representative of incompressible, flowless scenes for areas of shallow water. In addition, our project did not require and therefore did not implement the whole wave particles method, such as object interaction, wave reflection, or the extended height field in which waves can "crest".

2.3 Other Resources

Other references included the OpenGL Programming Guide [Shreiner et al. 2008], a superb resource for OpenGL code, and several websites with useful code snippets. VideoTutorialsRock.com has a series of very helpful tutorials, one of which is instructions on texture mapping [Jacobs 2008]. This site explains in detail making a texture on a 2D surface.

We also used the `imageloader` class from [Jacobs 2008]. This class was used to load a bitmap image from a file and invert it so it becomes right-side-up (it is read in upside-down). It keeps the pixel color information in a height x width x 3 char array. The array stores the RGB color of the ith pixel by three successive characters, and so it appears as array[R_1 / G_1 / B_1 / R_2 / G_2 / B_2 / ... / R_n / G_n / B_n], each entry a single byte, where n is the number of pixels.

3 Surface of the Water (Chris and Steve)

3.1 Representation

The surface of the water is represented with a mesh and a height field. The surface is stored as a grid of coordinates, whose default height and width are each 100, with an increment 0.25, make the mesh 400 x 400. The software has the capability to read in a different resolution for the mesh, but at the point this paper was written, it does not accurately depict any other width and height.

The height field is its own class, and features a grid (a vector of pairs of doubles, X and Y coordinates) , the heights (a vector of doubles, indexed correspondingly to the vector of the grid coordinates), a vector of wave particles (see section 4), a mesh for the surface, a mesh for the floor, a mesh for each texture (see section 5), and the texture itself, stored as an image from the imageloader class, discussed in section 2.3.

Each point on the grid, given an associated X and Y local coordinate, has a corresponding height. Our method relies on the superposition of wave particle influences on each height field point. The original method of updating height field values consists of going to every point on the height field and computing the contribution from each wave particle in the scene. In this method, the heights are updated according to a the sum of local deviation functions. A height field is defined as a continuous function of fluid level z over the horizontal position [Yuksel et al. 2007]. The height field can be represented by (1).

$$z(x,t) = z_0 + N(x,t) \tag{1}$$

In (1), N(x,t) is the deviation field of point x at time t. The deviation field is found by summing the local deviation fields, defined as (2).

$$N(x,t) = \sum_{i=0}^{k} D_i(x,t) \tag{2}$$

In (2), D_i(x,t) is the deviation value of the i^th wave particle (of k wave particles in the system) with respect to position x and time t. Wave particles and their functions are discussed in section 4. The individual deviation function is defined as (3).

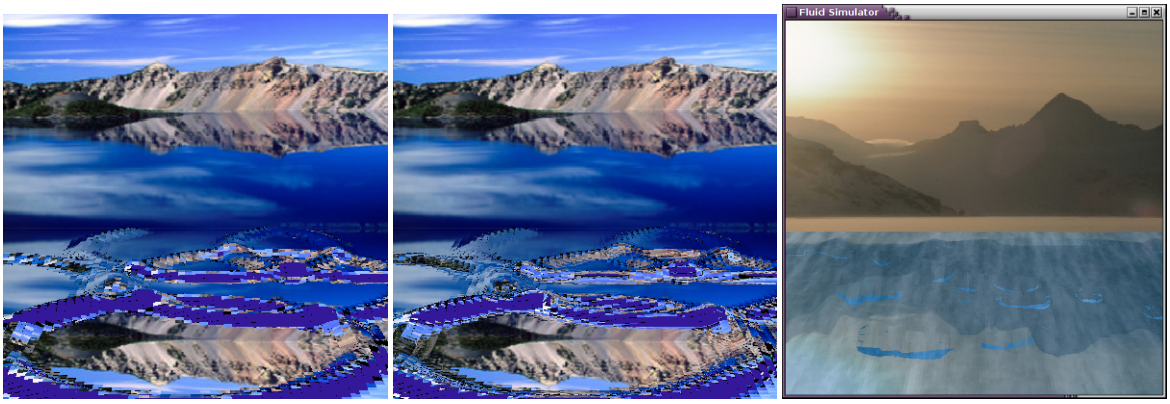

Figure 3: *Redering without weighted normal interpolation (left) and with average normals (center). Very little difference can be seen. Vertex normal interpolation based on quadrilization of mesh face (right).*

$$D_i(x,t) = a_i W_i(x - x_i(t)) \tag{3}$$

where a_i is the amplitude of the ith wave particle and W_i(x x(t)) is the waveform function with respect to position x and the position of the i^th wave particle at time t.

This method is inefficient in that there are many particles in the scene which have their contribution calculated only to find it is minuscule. In order to minimize wasted computations, instead of visiting every point on the height field, we go through every wave particle and calculate its contribution to just the nearby height field values. This optimization significantly increased performance with no loss in visual quality. The actual height field contribution is calculated through the use of a deviation function. Due to the strictly radial nature of the waves used in our project, we opted to use the radial local deviation function in [3]

$$D_i(x,t) = \frac{a_i}{2}\left(\cos\left(\frac{\pi\,|x - x_i(t)|}{r_i}\right) + 1\right)\Pi\left(\frac{|x - x_i(t)|}{2r_i}\right) \tag{4}$$

where r_i is the radius of the wave particle and Π is the rectangle function.

As mentioned in the paper, as long as the distance between two connected wave particles was less than one-half of the wave particle radius, a constant defined value in our case, the radial waves appeared radial and not like a group of individual particles.

3.2 Rendering

When deciding how to render the surface of the water, a number of factors were taken into consideration. Originally, a 3D spline was planned. A spline would allow for realistic flow of waves by approximating the shape of the surface using the height values along the mesh. With ray tracing implemented, the rays would reflect off of the surface, creating the image of a flowing and ebbing liquid surface. This idea was eventually dismissed for a variety of reasons. First, the math would be very difficult to work with considering the minimal improvement in realism. Secondly, the speed improvement would also be minimal, if there would be one at all. This is due to the fact that there are a series of optimizations that were used to speed up the rendering of the mesh representation. Third, rendering the mesh was deemed to be sufficient to represent the surface.

Ultimately, it was decided to render the surface as a series of quad faces in a grid as it was stored. The results, discussed in section 8, were adequate for our purposes. Each face is drawn as a quadrilateral, with the default color blue. To render, the glCanvas class cycles through each face in the mesh and draws it as an OpenGL quad (GL_QUAD), using the four vertexes of the face. The liquid surface is naturally drawn using the height field information, thereby creating rendered waves before ray tracing is applied. Details about the ray traced rendering of the grid will be discussed in section 6.

Rendering is done using a Phong Shading model. When first rendering the scene, those reflection rays that did not intersect the back texture had no coloring at all, and their corresponding pixels appeared black. To rectify this, we added the floor texture, and diluted the image on the surface with the floor texture, sending two rays from each mesh intersection point; one to the floor and one in the reflected direction. However, artifacts were still visible at the points on the mesh where those faces whose reflected rays would intersect the textures and those who wouldn't.

To rectify this situation we used the average normal of each of the eight neighboring faces to the face in question. We applied a weighting mask to the eight neighbors, making their influence on the normal of the intersected face differ according to their position relative to the face. This did not work either, as there were still clear division lines between those faces facing the texture and those whose reflected rays would not hit it.

We then decided to use the vertex normals and interpolate them across each face, instead of averaging the neighboring faces' normals. To calculate the normal at a point hit by a ray on a face, we first found the distance from that point to each corner vertex on the face, and used these distances as weighting masks. This also did not give us the desired results.

Finally, we interpolated each vertex's normal across the face by quadralizing the face, using the intersection point as the center of quadrilization. Therefore, each corner vertex forms a quadrailateral with the intersection point. We measure the area of each quad, and then weight the vertex's influence on the point's normal according to that area. This method produces satisfactory results.

4 Wave Particles (Steve)

The wave particle is the backbone of the fluid simulation, representing the properties of a section of a surface wave. Multiple wave

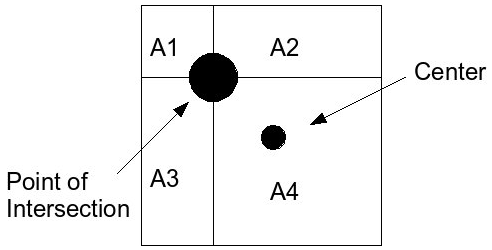

Figure 4: *Quadrilization of a mesh face. A1 through A4 are the areas that contribute to the weighting mask. Those with smallest areas (A1 referring to the vertex in the upper left) contribute the most.*

particles in a circle are used to create radial waves, just as wave particles in a line create a straight ocean wave. Each wave particle is completely independent of all other wave particles, significantly simplifying calculations compared to physically-accurate fluid simulation methods. Each individual particle stores values for position (both birth and current), amplitude, angle (both direction and dispersion), and birth time. The position values are used to keep track of where the particle originated from and where it is at the current time. The amplitude changes with time and eventually becomes so small that the particle is removed. The direction is the radial angle which the particle is traveling, and the dispersion angle represents the spread angle between two adjacent wave particles.

The fluid simulation works through iteration, removing old particles, subdividing current particles, and finally moving the particles around. Particle positions are compared to a bounding box of the scene and particles which are outside are removed. In addition, particles with amplitudes below a certain limit are removed from the scene. Next, if the distance between two particles becomes too large, the particle is subdivided. This consists of creating two new particles with same angle plus or minus one third of the middle particle's dispersion angle. The amplitudes of all 3 particles are divided by 3 so that the overall wave amplitude remains the same. Once subdivision is complete, all of the new and remaining particles are moved along in their respective directions at a constant predefined velocity. Once this iteration is done, the height field values are updated.

5 Texture Mapping (Chris)

The program allows users to input a bitmap file (.bmp) as a parameter. This bitmap is then transferred into an OpenGL texture to be drawn in the scene. The imageloader class contains a character array that holds the RGB values of each pixel in the texture. The texture is loaded into the system by the loadBMP() method, which also inverts the image (as it is read in upside-down). The texture is then bound to OpenGL, given the coordinates of the corners of the texture face.

The quadrilateral that the texture is bound to is stored as a 1x1 mesh. It is a mesh instead of a single quad because of the ray intersection code (discussed in section 6) is already in place for intersecting with a mesh face, and so the intersections are consistent. The texture is drawn above the water, like a horizon. Its colors are then reflected in the surface of the water, ebbing and flowing with the waves.

The texture is then repeated three times, creating "walls" around the water. This is done to ensure that every ray that is cast hits a texture. This is to prevent large gaps of color from the floor texture in the rendering that would occur if a ray shot beyond the texture without hitting it.

To query a texture color, the coordinates of the mesh that are struck with a ray are recorded, and converted into coordinates in the texture. These coordinates are then used to query the `char` array holding the colors. The texture's color at that location is then returned and used to discern the color of the fluid at the source of the ray that struck the texture. More on this procedure in section 6.

6 Ray Tracing (JP)

Raytracing a water scene has proved to be a very difficult problem. Due to the way the waves were represented to the raytracer, it became very computationally intensive to try to detect intersections against every quad in our water mesh (our mesh contains over 9000 quads). It was necissary to come up with a more efficient way of testing rays for intersection against the mesh.

The first way that we came up with was to test for intersection on one large quad that was placed at the rest location of the wave mesh. From the intersection data we would be able to detect faces that we might have intersected. This strategy was straightforward to implement, but does not yield results that are useful in our simulation. For shallow angles of intersection, this solution is inadequate, because it requires that you expand the block of faces that you have to grab by as many as all the faces in your system. This is obviously not what we want to do.

We could not simply find where it intersected the flat mesh because of the heights of the waves. If a ray passes through a wave, the intersection might not be noted by the system becuase it did not occur close enough the the flat mesh intersection point.

We needed a way to step down a ray and find faces that we might be intersecting. We need to do it in a way that does not result in checking every quad in the system for intersection. The solution that we settled on was to identify the quads that the ray was passing over, and just check intersection on those. This approach resulted in less than %1 of the mesh needed to be checked for intersection with the ray. This percentage could be lowered even farther by several clever optimizations. For instance, we can stop searching the mesh when the height component of the ray falls below the rest depth of the wave mesh. This works because the wave mesh can not sink, it can only get higher. Another optimization is only to test faces that are within some certain epsilon of the height of the mesh at that point.

We use a traditional recursive raytracer to trace into the system. We are testing 4 different objects in the system for intersections. They are the wave mesh, the floor mesh of the water, the texture mesh we are reflecting, and an enviroment texture (left, right, and back walls) to fill in the gaps where the waves are not reflecting the texture.

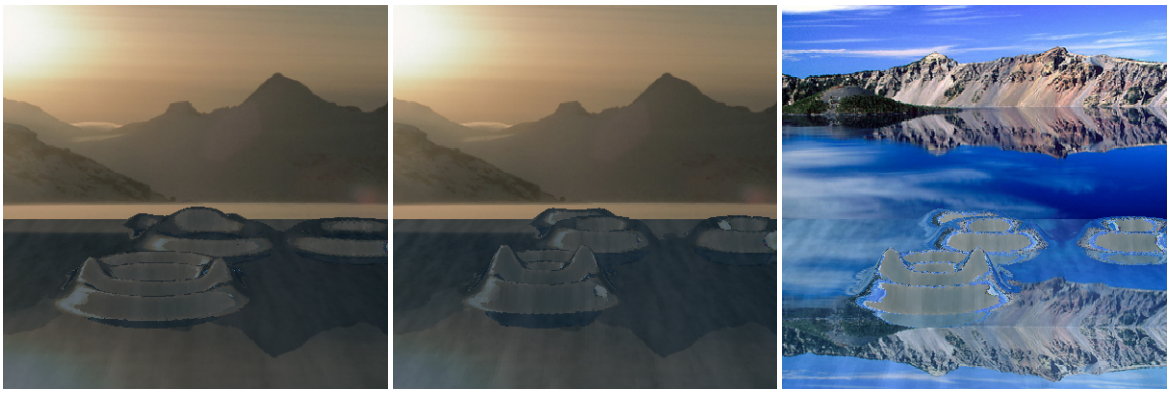

Figure 5: *The left and middle examples show 70-30 and 60-40 weighting of the influence of background texture and the floor texture (divided by 2, making it "oily"), respectively. The right picture shows a mountain view texture rendered.*

7 Difficulties and Shortcomings

There were several areas in the project in which roadblocks were encountered. Originally, the idea was the triangulate the mesh for easier normalization. This, however, proved to be futile because the normalizing code was already in place in the Mesh class. Also, the program can only take in bitmap files when the dimensions are powers of 2, because of the way the imageloader class was written. The mesh only renders correctly when the ratio of height to width (or vice versa) is 1:1 or 2:1. This is a bug we did not have time to fix.

The software is far from perfect. The first noticeable problem is seen when ray tracing. Those waves whose fronts face the camera have normals that do not intersect the texture and therefore are only being affected by the floor texture, thus creating odd artifacts. This is seen clearly when the amplitude is high and the mesh is coarse.

The texture backdrop shifts over a couple dozen pixels when ray tracing. This is the result of slightly-off indexing of the colors in the texture's color array, though it was a bug we were unable to fix. Also, the program is very picky about the size of the pictures that are sent to it. The backdrop texture must be a rectangle, preferrably one that is wider than it is tall. Also, it does not do well with anything smaller than 256 x 512.

There are several efficiency issues. When waves are added, it often takes almost a second to update the waves, especially when there are several waves in the scene at once. Also, the ray tracing code could be much faster. As it stands now, on a Intel Core-2 Quad machine, it renders a single scene in about 10 seconds.

8 Results

On an Intel Core-2 Quad machine, with 30 threads, and 5 waves spread throughout the scene, it takes approximately 28.1 seconds to render. The same scene with 5 waves and 15 threads takes 29.8 seconds, so threading beyond 15 does not improve performance. Running the same scene with 5 threads yields 31.5 second rendering, and 1 thread renders the scene in 111.6 seconds. At 4 threads, the scene is rendered in 32.2 seconds and 2 threads renders the scene in 55.7 seconds. Threading does improve performance, conceivably lineraly, but not past 4 threads. This is intuitive, as there are 4 cores in this machine.

We then rendered a scene with 20 waves and it took several (8) minutes to render the scene. The number of waves in a scene drastically affects the performance of the program, because the rays between waves reflect several times, thus slowing the algorithm down significantly.

Included are several examples of the final product, at various levels of background influence, wave number, wave height, and backgrounds.

9 Conclusion

9.1 Discussion

We have provided a piece of software that can read in a bitmap and reflect the picture in the waves of a body of water. The rendering is slow, and far from any real-time goals we had when we set out, but the rendering is clear and realistic looking with regard to the reflections. We have been able to produce a reflection off of the surface of wavy water. With the use of wave particles, rippling water is possible. Because the water is represented as a height field mesh, reflection of rays off of the surface, striking a user input texture, creates a reflection of the texture in the surface. Because of the waves, this reflection is realistically rendered.

Performance could be significantly improved. As it is, on an Intel Core-2 Quad processor, each scene takes roughly 10 seconds to draw with 5 threads. Clearly, this can be significantly improved. There were several roadblocks that we came across in our coding, most significantly ones that took our time away from optimization procedures. We learned quite a bit from this project. If we could do it over again, there would be several things we would change, such as using hard-coded numbers and making a better interface between the classes we each were working with.

9.2 Future Work

Several areas of the program could be improved for future work. First, this software would make an excellent software release. However, in order to accomplish that, several things need to be updated. First, the ability to click on the water surface and drag the pointer around to create a wave should be implemented. This would allow the user to dictate not only the starting location of the wave but also the shape and direction.

Also, the program should be able to read in bitmaps of any size and resize them to have dimensions that are powers of 2.

The waves shoudl have variable amplitudes, depending on how long the mouse button is pressed. At present, all of the waves have ran-

Figure 6: *Results involving several (>20) waves, a nice sky reflection, and a sunset reflection*

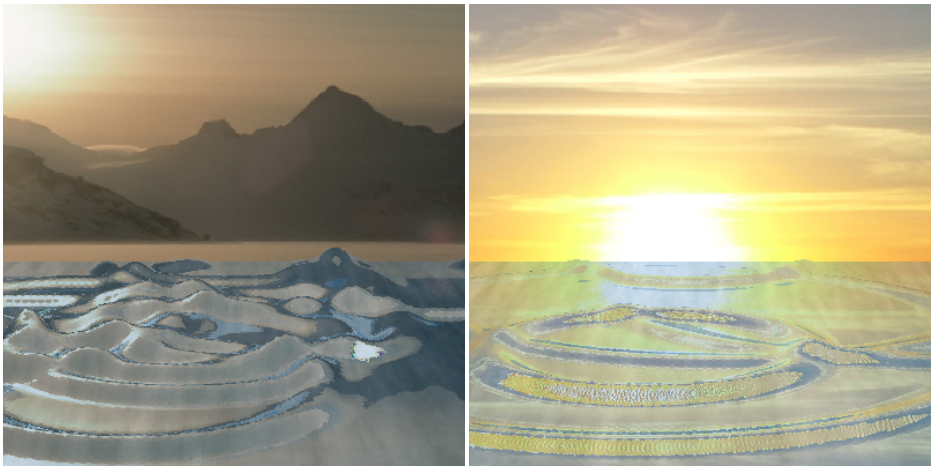

Figure 7: *A desert reflection and a sunrise reflection.*

dom amplitude (a standard deviation away from the mean amplitude). Also, the waves should deflect off of the boundaries and the texture.

An interesting and ready-made addition to the program would be to allow for the user to input cube textures. That is, a scene that is 360 degrees made into a texture divided into 6 sections, one for each inward face edge of a cube. This would allow for the scene reflected on the surface of the water to truly appear to be 360 degrees surrounding the body of water. This would add another level of realism to the scene.

Finally, more optimizations can be made, in order to make rendering faster. The ray tracing algorithm can be run on the GPU, as it is done in [Yuksel et al. 2007], to significantly improve the performance of the program.

References

JACOBS, B., 2008. Opengl video tutorial - textures. Website - http://www.videotutorialsrock.com/opengl_tutorial/textures/text.php.

NOTKIN, I., AND GOTSMAN, C. 1997. Parallel progressive ray-tracing. *Computer Graphics Forum 16*, 1, 43–53.

SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2008. *OpenGL Programming Guide*. Addison-Wesley.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June), 343–349.

YUKSEL, C., HOUSE, D. H., AND KEYSER, J. 2007. Wave particles. *ACM SIGGRAPH 2007*.

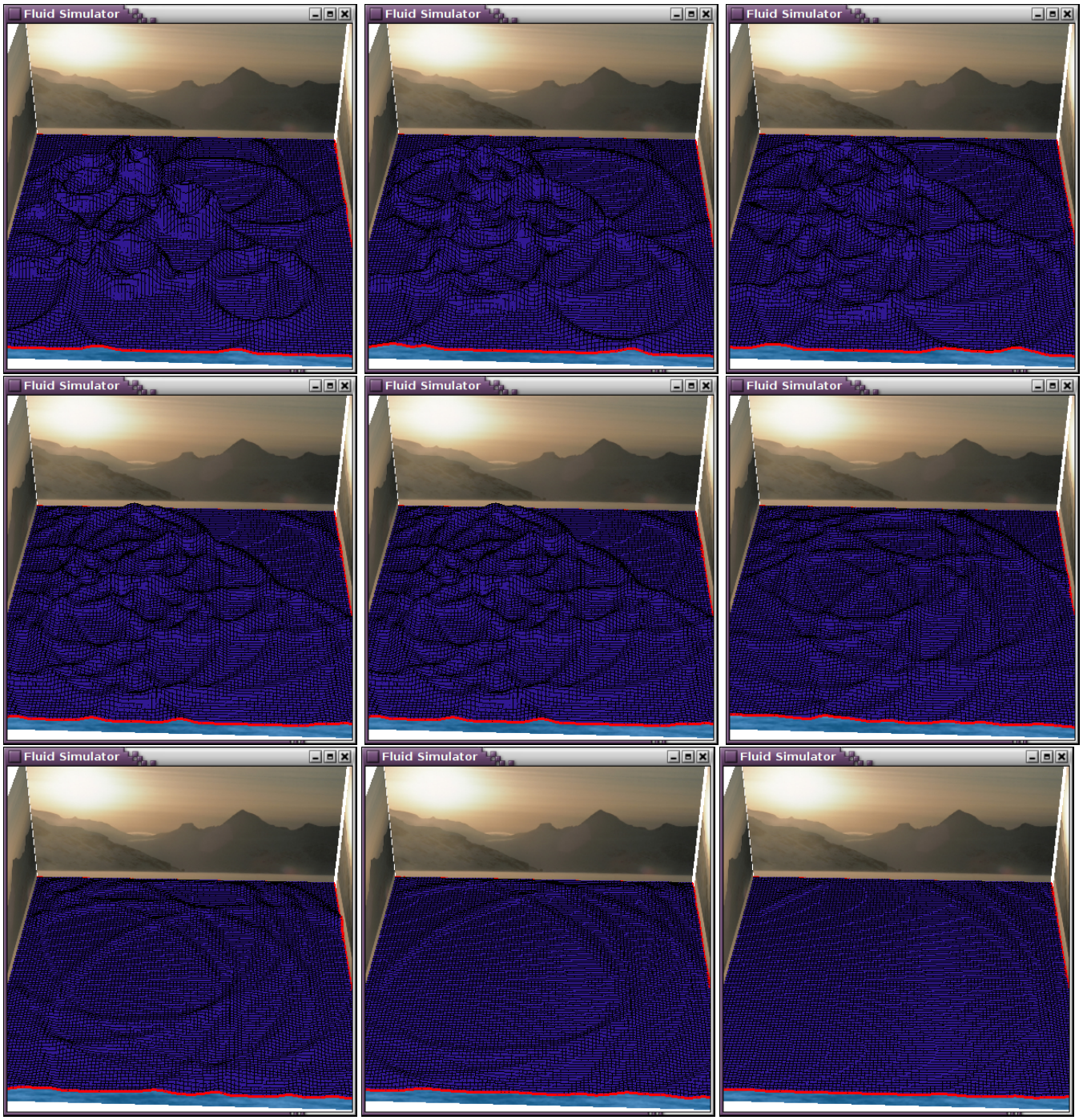

Figure 8: *10 wavefronts in the scene.*