Real-time Water Rendering

Nicolas Sanglard, Louis Lettry, Damien Firmenich

EPFL

1 Motivation

In old games, water is usually represented as a planar image, sometimes with a texture applied to it, but now as the power of computation increase, graphics are getting better so water rendering cannot be as simple as before because of the contrast with new graphics. Indeed in almost all games or 3D animations the moment defining the rendering realism is when you see the water because since it is an omnipresent element on earth, our brain quickly recognize differences between poorly rendered water and real water. Rendering very realistic water can be seen from two points of view : the real-time and the animation movies points of view. In animation movies the computational cost of simulating water is not the most important because what matters is to get the desired scene with the highest precision: the difficulty is to find the best model to simulate water and since then rendering a frame can take hours without being a real issue. But in real-time the challenge is completely different: the rendering must be in the magnitude of a fraction of second but the water must be as realistic as possible and to do this we have to find very fast models which approximate the fluid behavior.

The real-time water example is a perfect illustration of an actual problematic in the world of computer graphics because the trade-off between realism and computation time is an obligation: if you want to be very realistic you have to do a lot of computation but since real-time cannot tolerate this because of time constraints, these computations should be replaced by some "hacks" that reproduce water's behaviour as well as possible.

In addition to the physical complexity, rendering the right color of water is also a challenging and interesting part of the rendering because it is defined by a lot of different parameters such as the sky color, sun position and ground color.

(a)

(b)

Figure 1: Water rendering

As a result, we have chosen to render real-time water because first of all, well rendered water is very nice to see. The process of finding "hacks" to minimize computation time is also a very interesting topic because you can ask yourself if one day real-time will be as accurate as non-real time 3d rendering. Just by seeing the two pictures above you can clearly see that there is a convergence between these two worlds.

2 Related Work

Water simulation is a large field, and a lot of work have been done on it. There is four main techniques.

- Mathematical approach
- Bump mapping approach

- Particles approach
- Heightmap approach.

2.1 Mathematical approach

The basic idea behind this approach is to simulate water by using mathematical functions: the sine and cosine. A wave can be seen as a period of a sinus, the height of the wave is the range, and the length of the wave is the wavelength of the function. This similarity, the fact that we can sum sine and cosine and that they are continuous functions allow the developer to render water very precisely and very fast. But the negative point is that we can't interact with the water and the water doesn't have any imperfections which isn't realistic.

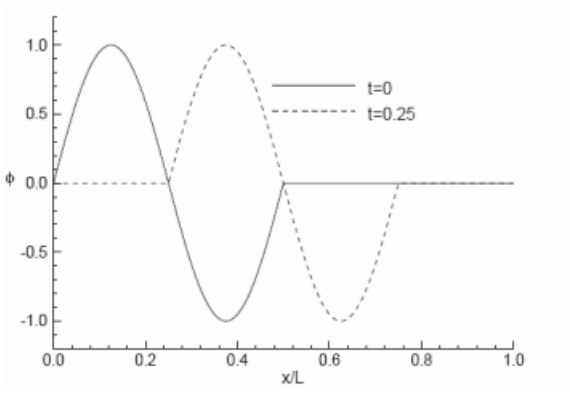
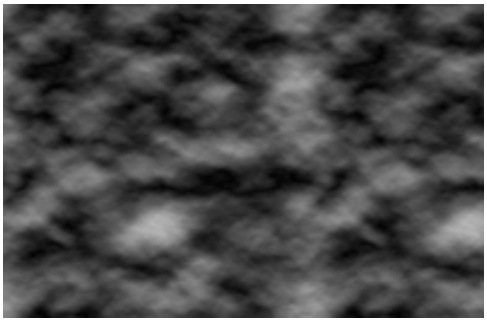


Figure 2: Mathematical approach

2.2 Bump mapping and texture approach

This approach means that with this method we will not try to simulate water but just make as if a surface was a water surface. There is many ways to do it: we can just create a texture which looks like water and make it move on a surface. You can also add bump-mapping to have an illusion of waves. To improve this illusion, the texture of the water can even make some static reflection of the environment so that the water looks really realistic. This method is cheap in term of computing power and make good render but we cannot interact with and there's some little glitch with it.



(a) Example of a bump map

(b) Example of fake water with reflection and bump-mapping. We can see that the character is not reflecting in the water, here is one of the glitches of this technique.

Figure 3: Fake water approach

2.3 Particles approach

This time we'll try to simulate water based on the physics. The water can be seen, at a microscopic level, as a set of atom represented by sphere that collide and move in the same space. So the particles approach will simulate these interactions of a huge number of little balls to recreate the behavior of the water at human level. This is the most realistic simulation we can do but the cost of computing all these interactions is really expensive and thus is practically not possible to do it during real time rendering.

2.4 Heightmap approach

This approach is in the same idea than the particles based method but instead of considering the water at a microscopic level, we'll consider it at a macroscopic level. This method split the water into little splices of water and will consider the physics for each of these splices and their neighbours. This is a really interesting method because it is not to expensive, we can interact with it and with some little modification

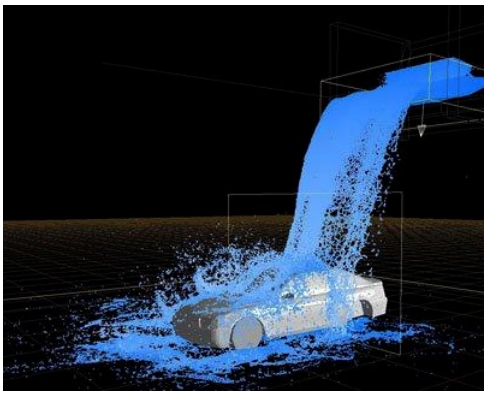


Figure 4: Particles approach

we can simulate interesting effects such that breaking waves or water flowing from a waterfall. This approach was inspired by [6, 5].

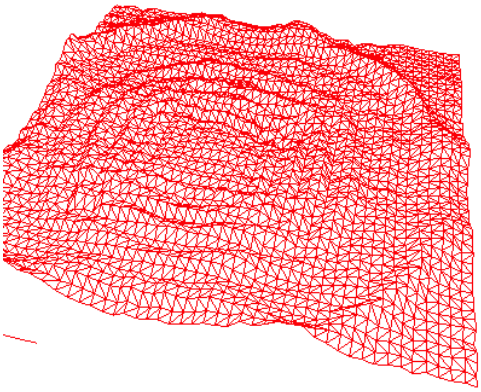
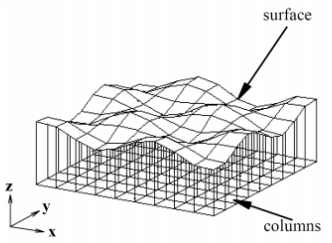


Figure 5: A heightmap rendered with triangle strips

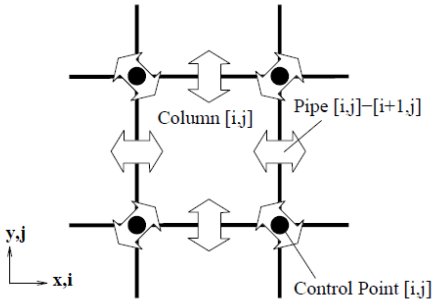
3 Method

3.1 Simulation

We decided to implement the heightmap method and try to see what we can add to it to improve the simulation. To simulate water with a heightmap we have followed the method described in [4, 3, 7] , the idea was to implement the breaking waves, but unfortunately we didn't manage to understand and implement well enough the second layer. But basically the idea is to simulate the physic of water by watching at some little columns of water interacting with their neighbours via some pipes that will make the water flow from one column to another.



(a)



(b)

Figure 6: Columns representation and the surface of water and of the pipes

To compute the amount of water flowing from a column to another we will stick to the reality by considering the static pressure of each columns.

$$H_{ij} = h_{ij}g + p_0$$

Where H is the static pressure, h is the column height, g the acceleration due to gravity and p_0 is the atmospheric pressure of the system. Then using the Newton's second law $F = ma$ we can derive the acceleration of the fluid in a pipe from column

$$a_{ij} \rightarrow kl = \frac{c\,(\Delta P_{ij} \rightarrow kl)}{m}$$

where c is the cross-sectional area of the pipe and m is the mass of the fluid in the pipe. Now we have equations that can simulate the behavior of the water in the columns and between them. So we use them to connect our columns. The basic idea of the algorithm is to iterate over all the columns each time and to compute the new amount of water of a

column depending on the water flowing in the pipes and then computing the new flow of water. This allow us to have a realistic behavior for our surface of water.

Now that we have some realistic water, we try to implement some objects that we could throw in the water. To do this we modified a bit the equation of the water's acceleration in the pipes to take into account the force of the object pushing onto the columns. The object will just be led by an original velocity vector and the gravity acceleration. When he will be touching the water, he will push it down and an additional force (Archimedes' principle).

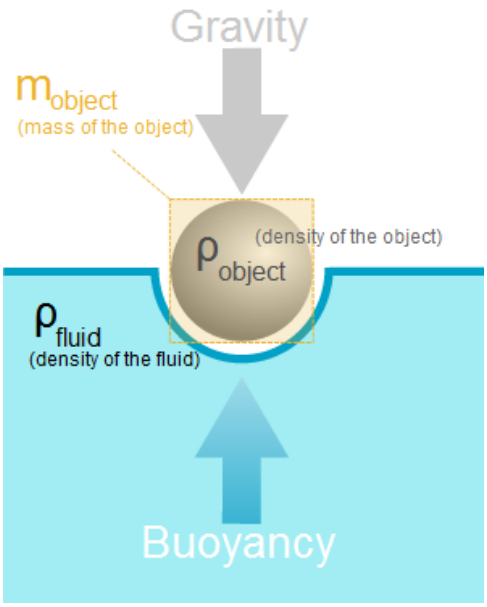


Figure 7: Illustration of the Archimedes' principle

3.2 Environment

A common issue when rendering a full scene in real-time is to display the environment and the objects in it. Speed would be an issue if all of them were rendered simultaneously, so a common method is to use an environment map, which is just a background texture, so less computing power is needed. This didn't affect the realism of the water because it isn't considered a crucial part and we don't have dynamically moving object in the background.

In our case, the texture was applied on a cube object using a basic ray method. The ray was computed using the camera's origin and the vertex's position by computing the difference vector between them in world coordinates. The texture lookup was done using the `textureCube()` implemented in the GLSL.

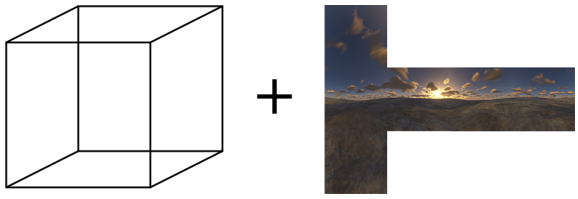


Figure 8: Environment texture applied on a cube object

The whole cube was rescaled big enough in order to look like a real environment with the viewer inside it.

In a first attempt, we tried a method which was to map a different texture to each face of the cube manually using texture coordinates. The problem with this technique is that we were unable to use the cubemap texture to compute reflection and refraction on the water, and there was inconsistencies between the environment map and the cube textures. Another issue was the emergence of artifacts at the cube's borders due to inaccuracy in the texture coordinates. We solved these issues by using a general environment and texture mapping as describe above.

3.3 Texturing

Water itself does not have a color of its own, it depends entirely on its surrounding. Light reflects on the water's surface and give us the color of what's above, and refracts what's below it. The final color of the water is a combination of both.

To simulate this kind of rendering, we used the same environment mapping as for the cube in order to have a consistent and realistic color. More specifically, the reflection component $\mathbf{R}$ was computed with the incidence vector $\mathbf{I}$ and the surface's normal $\mathbf{N}$ using the following reflection formula, implemented using the GLSL's `reflect()` function.

$$\mathbf{R} = 2\mathbf{N}(\mathbf{N} \cdot \mathbf{I}) - \mathbf{I}$$

The refraction component was computed using the same vectors by finding the angle of the refracted vector :

$$\theta_2 = arcsin\,(r \cdot sin\,(\theta_1))$$

where $r = \frac{n_1}{n_2}$ is the ratio between the two refraction indexes $n_{1,2}$ corresponding to the air and water indexes respectively, and $\theta_{1,2}$ the angle of incidence of the first ray an the refracted ray. The refracted ray is of unit length, so the component can be found direct using the computed angle. The whole computation was implemented using the GLSL's `refract()` function. To look up the color corresponding to each vector, we used the environment cube map as mentioned above and the GLSL's `textureCube()` function. After each color was retrieved, we then blent them together to have the final color.

The textures were taken from [1, 2].

3.4 Lighting

In this project, we used the phong lighting model, that means that the final intensity is defined by the ambient term plus the diffuse term plus the specular term:

$$I_{tot} = I_{amb} + I_{diff} + I_{spec}$$

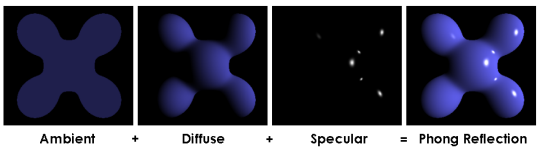


Figure 9: Phong lightning model

For the ambient term, we assume that the ambient light is uniform in the environment. But in reality the ambient light is the object's illumination caused by the light reflected from other objects in the environment, which is more heavy to compute and unnecessary four our purpose. So we can use the very simple formula for the reflected intensity :

$$I = I_{amb} * K_{amb}$$

where I_{amb} is the ambient light intensity and K_{amb} is the material ambient parameter.

The diffuse reflection, as opposed to the ambient reflection, depends on surface orientation and light position, but not on the camera position. As we can see on the image above the diffuse reflection is very important for the rendering because as it depends on the surface orientation, it gives a lot more information on the object's geometry than the ambient reflection. We used the well-known formula to compute the diffuse intensity:

$$I = I_p \cdot K_d \cdot (\mathbf{N} \cdot \mathbf{L})$$

where $\mathbf{N}$ is the normal vector, $\mathbf{L}$ is the light vector, I_p is the light source intensity, K_d is the material diffuse parameter.

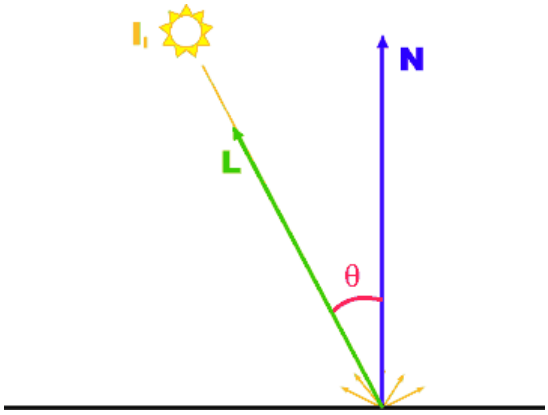


Figure 10: Diffuse reflection

And finally for the specular reflection we add some shininess to the rendering, which makes the water more realistic.

$$I = I_p \cdot K_s \cdot (\mathbf{R} \cdot \mathbf{V})^n$$

where $\mathbf{R}$ is the reflected vector, $\mathbf{V}$ is the view vector. K_s is the material specular parameter and n is the shininess parameter.

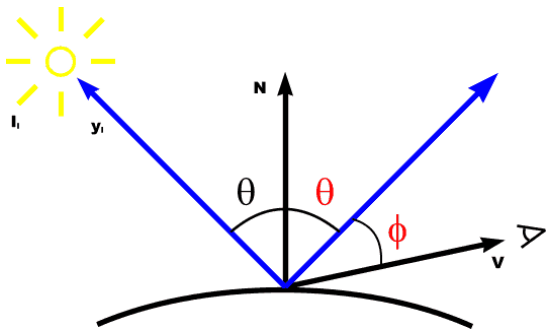


Figure 11: Specular reflection

3.5 Basic Interaction

We implemented some basic interactions with the water surface using a bump method. A bump is simply a way to modify the heightmap velocities, by adding a constant coefficient. Another technique was to modify the height values of the heightmap, but it is wrong because it actually generates water instead of just moving it around. So the velocities technique gives us a realistic looking interaction which we can use in various ways.

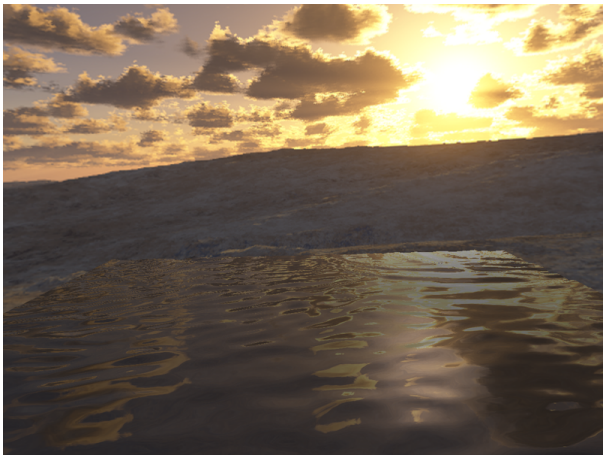
One way to use the bump method is simple user interaction with mouse clicks. The coordinates of the click is retrieved and then transformed from screen space to model space using the inverses of the matrices multiplied in reversed order as shown below, and a bump is applied at the computed coordinates in model space.

$$M_{scr \to mod} = M^{-1}_{mod \to wor} \cdot M^{-1}_{wor \to cam} \cdot M^{-1}_{cam \to scr}$$

Another way was to generated realistic looking random waves to make the water moving without user interaction. The technique was to simply apply a bump at a random location on a border, at random time, and with random strength. We used a random number generator and took the coordinates as $y = r \bmod n_{cols}$, and $x = n_{rows} - 1$ with r being the random integer and $n_{rows,cols}$ the size of our grid.

We also implemented a way to map a black and white image to our water surface using waves to display the image. The way it was implemented is by taking a image of the same size of our grid and make a bump wherever white pixel appears. A way to map an image of a size different from the grid would be to compute interpolation for the vertex and to bump accordingly.

(a)



(b)

Figure 12: Results with two different environment maps

4 Results

All the different techniques combined leads to a nice result, with simple water looking quite realistic. The normals for each vertex are interpolated between each other so the waves have a smooth surface, which allows us to compute nicely the reflection and refraction on those waves. It is important to note that, since the rendering of the water depends partly on the environment mapping, it is important to have a good texture.

The use of lighting does have its importance in the scene, but we noticed that mainly the ambient component of the phong lighting has a real impact on the rendering. The diffuse and specular components mixed with the lights present directly in the texture, so we mainly see those ones instead.

The speed of the rendering is perfectly reasonable, which means that the environment texturing is a good and fast approximation of real objects, this is a crucial part for the reflection and refraction. The simulation of the waves using the heightmap approach is also efficient, and real-time speed was not an issue with a reasonable grid size. In our experiments we used a grid of resolution 100x100 vertice, with a Core Duo 2.56Ghz, Nvidia 9400M 256Mb at a screen resolution of 1440x900.

So as a results, using a heightmap for real-time

rendering makes the water look nicer than procedural or bump-mapping technique, it runs faster than particle-based methods, and it is also easier to implement.

5 Discussion

Our implementation of height field simulated water has several limitations. The simulation is based on a rectangular grid, so we cannot extend the water to a more sophisticated surface such as a circle, lake, river, or fountain. The physics used for the simulation is simplified for the implementation, so if some properties and border cases are not well handled, we will see artifacts and side effects such as high frequency ripples which aren't realistic. For example, if only one vertex is moved up or down individually, the simulated waves won't behave properly because real water always moves as a group of particles. Another important limitation of heightmap simulated water is that complex phenomenons such as breaking waves and water spills cannot be approximated using a single heightmap because a single point on the grid cannot have multiple heights simultaneously.

Possible extensions to our system is maybe to combine a heightmap with a particles system to generate water spills. The grid simulation can also be generalized in order to give custom shapes to the water.

References

[1] Codemonsters.

[2] Humus.

[3] J. K. Hodgins J. F. O'Brien. Dynamic simulation of splashing fluids.

[4] Bálint Miklós. Real-time fluid simulation using height fields. Master's thesis, ETH Zürich, 2004.

[5] Matthias Müller-Fischer. Fast water simulation for games using height fields. 2008.

[6] Matthias Müller-Fischer and Robert Bridson. Fluid simulation.

[7] Matthias Müller-Fischer, Jos Stam, Doug James, and Nils Thürey. *Real Time Physics Class Notes.*