Oscar Choy
6/10/18
CS 114

CS114 Final Project Report

For the open-ended final project of this class, I decided to use WebGL to construct a simulation of water in a pool. The main focus of the simulation was to simulate reflection, refraction, and an implementation of (Fresnel equations) in water, with other objectives involving the movement of spherical objects above, below, and on the water itself. The overall goal was to show a simulated representation of a partially filled pool of water featuring various items in and around it, with special focus on showing what is reflected and refracted through the water. In addition, I had other objectives in mind at various points in the project which I ultimately pivoted away from; these involved the implementation of a moving water mesh and the implementation of light caustics. Unfortunately, for reasons I will discuss further in the report, I was not able to successfully implement these concepts and features into my simulation (though they would be interesting tasks to do for me to continue to try and do after this class is done).

The implementation of my project was, as I mentioned above, done in WebGL. I used quite a bit of code and libraries that I had previously used for the final project in my CS112 class; mainly, I made use of that project's "modelview" framework (how the canvas was set up and various libraries such as Vec3, Mat4, etc.), the "models" library (which included shapes such as spheres, cubes, toruses, etc.), the "trackball rotator" (which enabled camera and point of view movement with the mouse), and the implementation of the "Phong Illumination" shader code that project utilized. I also used that project's concept of tying position to a ticking timer in this project, as it seemed to be a logical way to animate the WebGL scene.

In addition, I utilized a combination of raytracing concepts from this project and from the "Project 2" assignment of this class (CS114). I calculated rays and intersections on the fragment shaders in a similar fashion to the CS114 project (trace from the modelview to the object and such), and used the concepts and equations involving ray-sphere intersection that were so vital to completing "Project 2" in this class. I did this because I did not wish to use the two "traditional" methods for simulating reflection and refraction, which involve using a cubemap for the reflection part and a premade texture as input for the refraction part. Instead, I wanted to calculate collisions of traced rays with objects in the scene in "real-time".

Using all the aforementioned elements as the base, I then implemented a "water mesh" to simulate the water that I would be showing in my WebGL scene. This "water mesh" was really just a height map implemented in the same file as the “models” I was re-using from the old 112 final project; I implemented the vertex and normal information in the same “models.js” file those other shapes were in, and my implementation tried to borrow a bit from the implementations of that project and the concepts found in our “Project 3” with the moving sheet/heightmap. In short, I tried to implement am analog of Project 3’s moving heightmap onto my chosen environment. I succeeded in making a working mesh, though I failed in implementing moving capability; I will talk more on that later in the report.

However, my water mesh was successful for what it needed to truly be: a heightmap mesh which I could use to implement my simulation of the surface of a pool of water. I could then build the pool and the spheres in the scene using the shapes in my “models” library, and then implement the water mesh at the height I wanted inside the pool such that the pool was “filled in” up to that height. The next step, of course, was to get that “water mesh” to actually look like water as opposed to a sheet of black (I had set the water mesh’s diffuse color to (0,0,0) in rob channels, so it would show up as “black” if it only had shown its diffuse colors. To do this, I need to implement reflection, refraction, and the Fresnel coefficient calculated by the Fresnel equations. The objective was to make the pool look like it was filled to a certain height with water (I tried to make it “moving” water, but the main objective was to at least make it look like still water) with objects floating on that water and objects moving under that water, with the water itself showing the reflection and refracted images of objects moving inside it, on it, and over/above it.

Implementing reflection was relatively basic. The equation for the reflectance vector “R” at a point “O” with normal “N” and incident vector “I” is:

(2 * (dot (N, I) * N) – I if the I vector is pointing away from the point “O”

(2 * (dot (N, -I) * N) + I if the I vector is pointing towards the point “O”

In these equations, “dot (N, I)” and “dot (N, -I)” were the dot product of N and I and the dot product of N and -I respectively. I then took this “R” vector, normalized it, and used it for the raytracing step to determine if this reflectance vector hit any of the other possible surfaces and objects in the scene. First, I saved the dot product of this vector with the incident vector to use in later calculations of the orientation, which I will discuss later. I then checked to see if it hit any of the spheres at or above the water using the ray-sphere intersect equations I made use of in “Project 2” (to be exact, there was an orange sphere placed on the wall above the water, and both a red and yellow sphere “floating” on the water); if it hit any of them, I would calculate the “t” value that would lead to an intersection, taking the lesser of the values if there were more than one. I then multiplied my reflectance vector with this value and took the dot product of this resulting vector and the incident vector. I then checked this value against my previously saved dot product of the incident and the reflectance vectors; if they had opposite signs, then the intersection was not valid because opposite signs implies the value of intersection is at a point on the opposite orientation of the reflection (in short, it wouldn't be a reflectance vector, it would be going backwards from that, as if it's looking INTO the water at the opposite angle that it should be reflecting at). Any invalid intersections resulted in a failed intersection.

If the reflectance vector did not intersect with any of the spheres in a valid fashion, I then checked to see if it instead hit any of the four pool walls. I did this by checking to see where the reflectance ray from the origin would hit the plane that the pool wall was on, for each wall iteratively. To do this, I saved both reference points and the surface normal at those reference points for each pool wall (one point and one normal for each wall) and saved the interior wall corners of the pool for later reference and calculations. I proceeded to try and solve the plane equation specifying where the reflectance ray from the water surface point hit the plane specified by the reference point and the surface normal at that point. Once I calculated that location, I calculated various vectors; one from the top-left of the specific wall I was evaluating to the intersection point (I called it "AM"), another from the top-left to the top-right of that wall (I called it "AB"), and a third from the top-left to the bottom left of the wall (I called it "AD"). I determined if the intersection point lied on the wall by calculating various dot products and checking two conditions:

1) If (0.0 < dot (AM, AB) < dot (AB, AB))
2) If (0.0 < dot (AM, AD) < dot (AD, AD))

If both of those conditions held, then the reflectance vector from the point of the water surface O" hit the wall being evaluated, and the surface would show a reflection of that wall at that point. I evaluated this for all 4 walls, determining if a reflectance vector hit any of the walls or missed all 4 of them. Overall, the idea was to use raytracing to see if a reflected vector going from a point on the surface of the water "O" hit a sphere, a wall, or nothing at all, saving an appropriate "reflectColor" if there was a hit at any shape/surface. These equations to determine plane intersection were specifically researched and implemented for this final project (unlike the sphere intersection equations which I had learned about before for "Project 2", even if the implementation was different).

I used a similar ray tracing and collision detection scheme for the refraction portion of the color calculation, but I had to factor in a couple additional elements. First, there was the pool floor to consider, such that I needed to check not only the 4 walls, but then check to see if a refraction ray that failed to hit the 4 pool walls was successful in hitting the "pool floor" plane within the boundaries of the pool floor (so I needed to check 5 planes for intersection, not 4). Also, whilst the number of possible spheres the refracted ray could collide with did not change, the orange sphere was replaced by a blue sphere that would always be under the water. Thus, while the raytracing concept was similar, the set of objects available for collision was different.

But the key difference of the refraction implementation was the simple fact that simulating refraction requires using a different set of calculations to find the refraction ray than the reflectance ray does. The calculation for the refraction ray makes use of a value called the “index of refraction”, which is related to the amount that light “bends” at the boundary/surface between two different mediums (such as air and water, as illustrated in this project). This relation is specified in Snell’s Law, which states the relationship of the angle of incidence (call it “theta-I”), the angle of refraction (call it “theta-T”), and the two indices of refraction of the two mediums separated by the surface (use “n1” for the index of the medium the incident ray is coming from, and “n2” for the index of the medium the incident ray is being refracted through). For the purposes of completion, the index of refraction of air is 1.0, and the index of refraction of water is 1.3). Snell’s Law is as follows:

(sin(theta-I) / sin(theta-T)) = (n2 /n1)

Using Snell’s Law, some dot product calculations, and some trigonometry, I was able to calculate the cosines of both the angle of incidence and the angle of refraction, and the refraction vector “T”. I then used this refraction vector to calculate what the incident ray would hit as it goes through the water at a point on the water surface “O”; if it hit a sphere, wall, or the pool floor, I would save an appropriate “refractColor” of the object the refracted incident ray would hit.

Before I go any further, I must briefly talk about a somewhat subtle difference in the calculation of what exactly the “incident” ray is. There is a difference between what a person “sees” and where light from a “light source” hits in a scene, because unless the light source is at the exact same place as the point of view (the camera, as is were), the incident vector/ray from those two points starts in a different spot, and hits the refraction boundary at a different spot, leading to different incident rays, different refraction/transmission vectors, and different places that these rays hit a surface/plane. My calculations, therefore, all make use of an incident ray that comes from the point of view/camera; every time the camera in the WebGL scene looks into the water, the water will show what an incident ray coming from the point of view hits as it passes through the surface, not the point lit up by the light source of the scene. As it were, the scene’s light source only factors into the color at a certain point of a surface being reflected/refracted; for the purposes of this scene, the color at the reflected/refracted point is a Phong-illumination-modelled calculation of the color given that current light source and the position of that point and the triangular face it lies on.

The last element of the calculations involve the Fresnel equations as it relates to the ratio of reflected and refracted light/color at a point on the surface of the water. The angles of reflection, refraction, and indexes of refraction of air and water determine how much the water will show what it's reflecting and how much it's refracting. Given the refractive indices of the two mediums (let "n1" be the refractive index of the medium the incident ray is coming from, and "n2" be the refractive index of the medium said incident ray is being refracted through), the cosine of the incident/reflection angle (call it "c1") and the cosine of the refraction angle (call it "c2"), the Fresnel equations give two values/components which, averaged together, give the actual ratio of reflected light "FR", to total light (FR / 1.0). The two components are calculated as follows:

fPar = (((n1 * c1) - (n2 * c2)) / ((n1 * c1) + (n2 * c2))) ^ 2

fPerp = (((n1 * c2) - (n2 * c1)) / ((n1 * c2) + (n2 * c1))) ^ 2

Then, FR = (0.5) * (fPar + fPerp).

And due to the conservation of energy, the ratio of refracted light "FT" is simply (1.0 – FR).

Once these values "FR" and "FT" are calculated, the total color value of a particular point in the water mesh is simply:

Color = (FR * reflectColor.rgb) + (FT * refractColor.rgb)

The colors of the non-water surfaces were calculated based on the Phong illumination model I utilized in the 112 Final Project (which I also revisited in our "Project 1" in CS114). For the sake of completeness, the following equations are the calculations used in the implementation of the Phong illumination model for that project (and re-used for this one):

The diffuse component:

color += (0.8 * max(dot1,0.0)) * diffuseColor.rgb;

The specular component:

color += (0.4 * pow(max(dot2,0.0),specularExponent)) * specularColor;

where dot1 = dot(L,N) and dot2 = dot(R,V). In this case, L is the incident vector (which I specified as "I" in equations before when it was coming from the eye, but here the incident vector is coming from a specular light source) pointing away from the point "O". And as specified previously, the "reflectColor" and the "refractColor" values of the surfaces under the water or reflected by the water were the colors calculated in the Phong illumination model as if the light source was unblocked/unaffected by the water itself.

I also implemented some additional animation functionality to the scene which moved the various spheres around in different ways, which served to illustrate the reflection and refraction properties of the water. The first and most notable of these was the fact that I arranged for the red and yellow balls to "float" on the water mesh in a circular fashion; while the x and y coordinates simply used a sine function to change their "x" and "y" values, their "z" values instead used a sum of sine function and a cosine function to determine their fluctuations in height from the surface of the water. The second thing I did related to the blue ball that was permanently under the water; I implemented a sine function to vary its height such that it would be periodically moving up and down under the water, with the water surface showing the refracted view of that periodically moving underwater ball. The last of these animations involved the orange ball initially placed at the top of the northernmost wall; I implemented a sine function to change the "x" and both a cosine function and an absolute value function to change the "y", such that the ball ended up swinging horizontally from the east wall to the west wall in a half-circle motion. This was done to showcase the reflective properties I implemented in the water mesh.

As for functionality that I attempted, but failed to implement, I have two areas to mention in particular: water mesh movement/animation, and light caustics. My attempts at implementing a moving water mesh were centered on ideas I had from out "Project 3" (the moving cloth project), in that I wanted to implement a height map that was built in such a way that the individual vertices could move periodically and create a new mesh shape with each "tick" and draw associated with it. To that end, I built the "water mesh" in far greater detail using far more vertices, triangles, and normal than any of the shapes already present in the "models.js" file I was using to implement pre-built shapes; my intention was to have a mesh in which each individual vector's height could vary/be changed based on the passage of a "tick" timer or a running counter total, and this value would influence what value gets plugged into the sine and cosine functions whose sum would then simulate the height of the mesh. I had even further plans to replace this sum of sines and cosines with the wave function to simulate the water even more accurately. Unfortunately, I could not figure out a way to get the mesh's vertex locations to change periodically; I tried storing the vertex values in a buffer that would be updated with each draw and referenced in the next draw, I tried to pass in a running counter to the shape, but I could not get it to actually move. I feel like I am on the right track, however, and further study of WebGL capabilities and the "Project 3" code would likely give me insight as to a satisfactory solution (due to implementation and library incompatibilities, I cannot use the Project 3 code in my current project; any functionality I ultimately implemented in this final project related to that "Project 3" is functionality I had to rebuild and implement myself using my chosen environment and library as it related to the 112 project code base/library I was using as this project's foundation). I would ultimately like to find a way to implement this moving water mesh and the spherical flotation that depends on this movement to render a more physically accurate model, but I was not able to successfully do that in time for this project.

The other area I initially attempted to implement work on was water light caustics. However, there were two obstacles I encountered that compelled me to abandon this area of work for my project. First of all, the actual mathematical calculations involved in implementing this feature were challenging for me to understand, let alone understand how to implement as it related to my WebGL environment. I found these equations in NVIDIA's GPU Gems website, and I tried to utilize further research on the internet to understand this area further. Unfortunately, I was not successful, and in fact almost all of the resources I found on the subject as it related to OpenGL and WebGL (mainly, Evan's Water Simulation and the GDC 2008 conference slides by Matthias Muller-Fischer) recommended a method utilizing pre-drawn textures to "map" the caustic effects onto the surface as opposed to calculating them in real-time, which they all stated was both difficult and too expensive in terms of time and resources to implement in WebGL. But what sealed the fate of the caustics idea was the discussion I had with Professor (Zheng) in which he mentioned that caustics were only applicable in moving/curved water, and not in still water. Given that I was not able to successfully implement moving water to begin with, and that my final project ultimately treated the water as if it was lying still, implementing caustics in this case would be both unnecessary and nonsensical. Thus, I abandoned this idea.

Ultimately, my project involved the simulation of a scene featuring a pool of water that includes various spherical objects floating on said water, in the water, and moving above the water. The project also features a simulation of the visual properties of water, namely reflection and refraction, with the Fresnel coefficients determining the ratio of each element to more accurately depict the water itself. Using a height map, I was able to simulate the water and what the water would show if there were 2 spheres floating in it, one sphere floating inside it, and one placed above it moving around, along with how it would show the reflection and the refraction of the walls of the pool said water is filling. Though I was not able to get the water itself to move nor was I able to implement any light caustics through the water itself, I did gain an understanding on why we see what we see through water, how it affects light passing through it, and why it reflects some things, doesn't reflect others, and shows certain things at certain angles that happen to be placed on the other side of it. Perhaps with further work and study, I can add to this model and implement the moving mesh I was not able to implement, and maybe better understand the equations behind water caustics.