# Singapore University of Technology and Design

# 50.017 : Graphics & Visualisation

## Final Term Project Report

Han Zikang 1000436 | Jermaine Cheng 1000502 | Ng Ping 1000513

Contents

# 1. Introduction

Highly uniform objects such as sand and grains have been modelled within multiple simulations. However, the simulation of the 3D dynamic motion and interaction between different materials presents a challenging problem because of added physical phenomenon between vastly different objects. For example, adding a collision between a heavy glass orb and a rubberized cube cannot be simply modeled by the physical forces of friction,strain and shear stress. We have to also consider optical properties and the deformative properties of the objects to correctly render a visually accurate simulation.  By incorporating techniques from given physical models, our approach hopes to describe solid-solid and solid-liquid based interactions more faithfully than previous methods by incorporating ray tracing and solid deformation.
(For instructions & code structure overview, please see README.markdown inside the root folder of Final)

# 2. Initial Idea

### 2.1 Overview

In this simulation, we intend to simulate the interactions, such as collisions and interactions of both rigid and deformable components(such as blobs) as well as reflection and refraction of objects of different shapes and objects.

The final deliverable will be an interactive interface featuring multiple objects of different materials and physical properties contained in a glass aquarium.

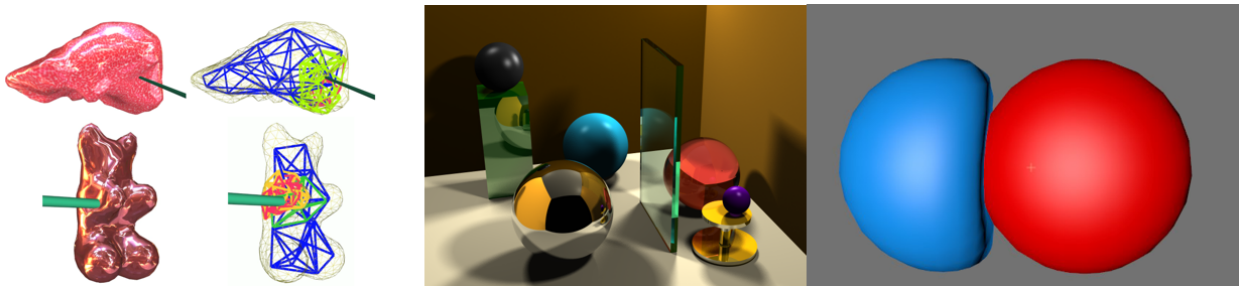

Fig 2.1 From left to right: Deformation, Reflection and refraction, Collision detection

In our early development phase, we started out by using C++ and OpenGL interface to develop our simulations. We also used the GLUT graphics library to develop our initial stage.

### 2.2 Difficulties faced

Our development was halted when we faced issues with the OpenGL and the GLUT library specifically. Such issues included Vega FEM 2.2 being unable to compile on Xcode which led to us being unable to implement deformable objects effectively. Furthermore, modelling of live elastic deformation also consumes large amounts of computing power.

On top of that, the use of the GLUT library is not optimal for this project due to the need for various shader, renderer and model/picture loader libraries which runs on the newer GLEW or GLFW3 libraries instead. After many attempts at resolving compilation errors and library issues, we decided that using the GLUT library in OpenGL will not be able to allow us to simulate the environment we proposed

above.

# 3. New Development

### 3.1. Overview

A suitable substitute for a graphical interface we found is WebGL which runs in the browser. Three.js is the 3D library used for implementation.

### 3.2. Basic Collision Detection

For our project, we implemented a collision model which includes a system of balls and walls. So the collisions which occurs will be in either ball-ball collisions or ball-wall collisions. Gravity is constantly applied with a constant on the y direction of all the balls.

For the basic model, we will need to determine all possible ball-ball pairs and ball-wall pairs. So every ball created will have a ball-ball (BB) pair with all other balls in the environment and a ball-wall (BW) pair with the 6 sides of the box.

The BB pairs will then be tested if they are colliding by checking if they are moving away or towards each other and if their radius intersects one another. The BW pairs will be tested for collision in a similar way, where the BW pair will be tested if ball is moving towards each other and if there is intersection.

If they are colliding, the speeds of the ball will be 'reflected' just like a beam of light by reversing the velocity of each ball direction.

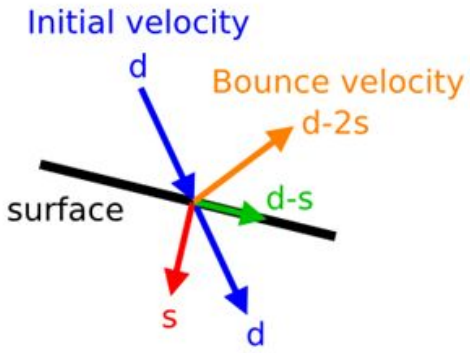


Fig 3.2.1 Collision reaction

### 3.3. Optimized Collision Detection: Octree

However the method above of having each ball calculate the relative velocity and radius intersection of each and every ball in the system is extremely taxing on the computing power resulting in much lower FPS as more balls are added in. To allow more balls to be thrown into the system, we have made use of the Octree data structure to optimize the collision detection of our simulation.

What the octree accomplishes is simply by subdividing the volume of the box into compartments, where the compartments will be further divided if there are more than a certain number of balls inside of it. If there are no balls in the subdivided compartment, it will be destroyed.

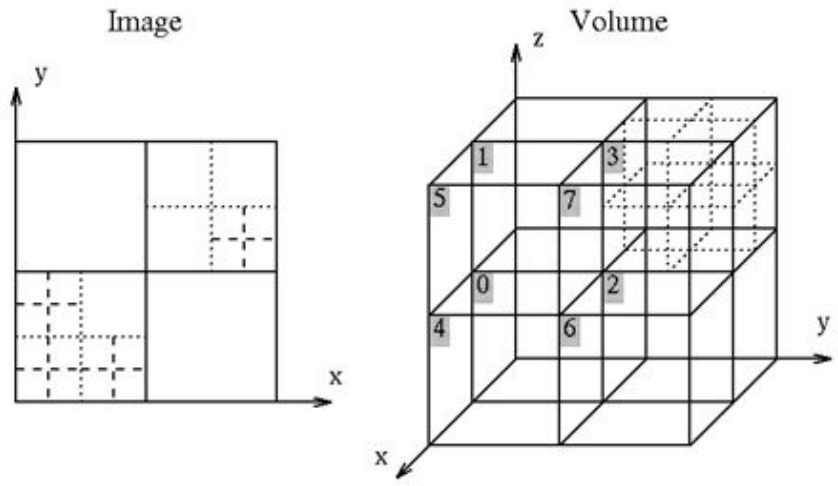


Fig 3.3.1 Octree visualization

So instead of having each ball calculating distance and direction for every other ball, we would ‘file’ the BB and BW collisions into the octree class.

The octree would organize the balls into subdivided compartments, where collision calculations will only be done for the balls within that compartment. Effectively reducing the number of calculations made in the environment. The time step/ refresh rate will be determined as well to tell the code to refresh the octree, so a new set of calculations can be done for the moving balls.

When the code determine which BB and BW pairs will collide, the interaction will then be the same as above, where the ball will bounce off ‘reflectively’ against other ball or wall objects.

```
class Octree { ... };

Declarations
void potentialBallBallCollisions(vector<BallPair> &potentialCollisions,
                                 vector<Ball*> &balls, Octree* octree) { ... }
//Puts potential ball-wall collisions in potentialCollisions.  It must return
//all actual collisions, but it need not return only actual collisions.
void potentialBallWallCollisions(vector<BallWallPair> &potentialCollisions,
                                 vector<Ball*> &balls, Octree* octree) { ... }
//Moves all of the balls by their velocity times dt
void moveBalls(vector<Ball*> &balls, Octree* octree, float dt) { ... }
//Decreases the y coordinate of the velocity of each ball by GRAVITY *
//TIME_BETWEEN_UPDATES
void applyGravity(vector<Ball*> &balls) { ... }
//Returns whether two balls are colliding
bool testBallBallCollision(Ball* b1, Ball* b2) { ... }

//Handles all ball-ball collisions
void handleBallBallCollisions(vector<Ball*> &balls, Octree* octree) {
    vector<BallPair> bps;
    potentialBallBallCollisions(bps, balls, octree);
    for(unsigned int i = 0; i < bps.size(); i++) {
        BallPair bp = bps[i];

        Ball* b1 = bp.ball1;
        Ball* b2 = bp.ball2;
        if (testBallBallCollision(b1, b2)) {
            //Make the balls reflect off of each other
            Vec3f displacement = (b1->pos - b2->pos).normalize();
            b1->v -= frictionCoeff*2 * displacement * b1->v.dot(displacement);
            //b1->v -= frictionCoeff*2 * displacement * Vec3f::dot(b1->v,displacement);
            b2->v -= frictionCoeff*2 * displacement * b2->v.dot(displacement);
            //b2->v -= frictionCoeff*2 * displacement * Vec3f::dot(b2->v,displacement);
        }
    }
```

Figure 3.3.2: Code snippet for collision (discarded C++ skeleton)

Ball-ball is only implemented in the first version of our project in C++. It is not implemented in the WebGL version.

### 3.4. Cube mapping

In this project and computer graphics, cube mapping is a method of environment mapping that uses the six faces of a cube as part of the map shape. The environment is projected onto the sides of a cube and stored as six square textures, or unfolded into six regions of a single texture.  With this technique, we can effectively create realistic looking environments, reflection and refraction effects.

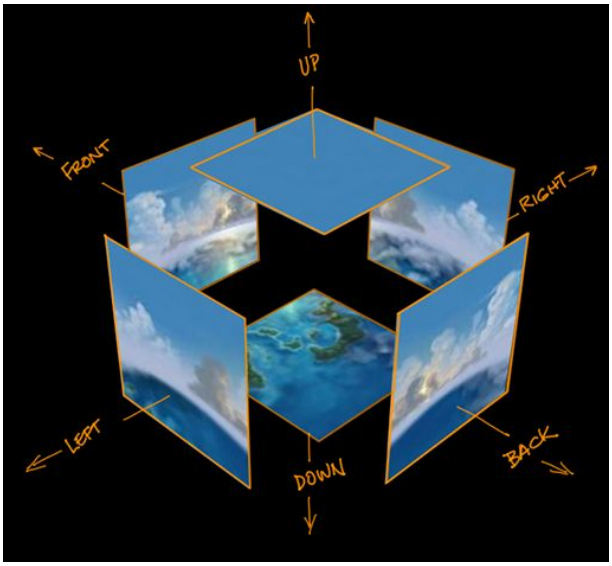

Figure 3.4: Illustration to show how cube mapping is done

### 3.5. Sky boxes

The idea with the skybox technique is to have a big box encase the camera as it moves around. If the box appears to be the same distance even when the camera is moving, then we get a parallax effect where we receive the illusion of it being relatively far away. Similarly, if we fix the skybox as the camera rotates, we can create a pan around effect.

However, only if the initial textures were well constructed, the distortions of the created environment will not appear distorted. In a non-ideal construction of the skybox, it is not difficult to notice the seams of the box.


Figure 3.5.1 : Ideal skybox rendering example

To create the skybox in webGL via Three.js, we have to first define a cube texture and load a t-shaped texture file we want to use as the skybox. After which, we define the material via shaders of the box and add it to a predefined box geometry mesh.

```
var cubeMap = new THREE.CubeTexture( [] );
cubeMap.format = THREE.RGBAFormat;

var loader = new THREE.ImageLoader();
loader.load( 'textures/skyboxsun25degtest.png', function ( image ) {

    var getSide = function ( x, y ) {

        var size = 1024;

        var canvas = document.createElement( 'canvas' );
        canvas.width = size;
        canvas.height = size;

        var context = canvas.getContext( '2d' );
        context.drawImage( image, - x * size, - y * size );

        return canvas;

    };

    cubeMap.images[ 0 ] = getSide( 2, 1 ); // px
    cubeMap.images[ 1 ] = getSide( 0, 1 ); // nx
    cubeMap.images[ 2 ] = getSide( 1, 0 ); // py
    cubeMap.images[ 3 ] = getSide( 1, 2 ); // ny
    cubeMap.images[ 4 ] = getSide( 1, 1 ); // pz
    cubeMap.images[ 5 ] = getSide( 3, 1 ); // nz
    cubeMap.needsUpdate = true;

} );

//references the shader library for cube mapping
uniforms = {
    tCube: {type: "t", value: cubeMap},
    tFlip: {type: "f", value: -1},
    darkness: {type: "v4", value: new THREE.Vector4( 0.7, 0.7, 0.7, 0.0 ) }
}

var cubeShader = THREE.ShaderLib[ 'cube' ];
cubeShader.fragmentShader = document.getElementById( 'fragmentShader' ).textContent;
cubeShader.uniforms = uniforms;
cubeShader.uniforms[ 'tCube' ].value = cubeMap;
cubeShader.uniforms[ 'darkness' ].value = sunColor ;
```

```
//apply shaders for cube mapping
var skyBoxMaterial = new THREE.ShaderMaterial( {
    fragmentShader: cubeShader.fragmentShader,
    vertexShader: cubeShader.vertexShader,
    uniforms: cubeShader.uniforms,
    depthWrite: false,
    side: THREE.BackSide,
} );

//draws skyboxes on the box
var skyBox = new THREE.Mesh(
    new THREE.BoxGeometry( 1000000, 1000000, 1000000),
    skyBoxMaterial
);

//adds the skybox to the scene
scene.add( skyBox );
```

Figure 3.5.2: Code snippet to define skyboxes

### 3.6. Reflection and refraction

Using cube mapping to create the skybox, we can similarly texture objects within the scene with the same textures as the skybox to produce a relatively convincing reflective effect.

The principle is very simple. We work out a direction vector from the viewpoint (camera position) to the surface. We then reflect that direction off the surface based on the surface normal. We use the reflected direction vector to sample the cube map. This gives us a new specific texture mapping for the object to simulate the reflective effect.

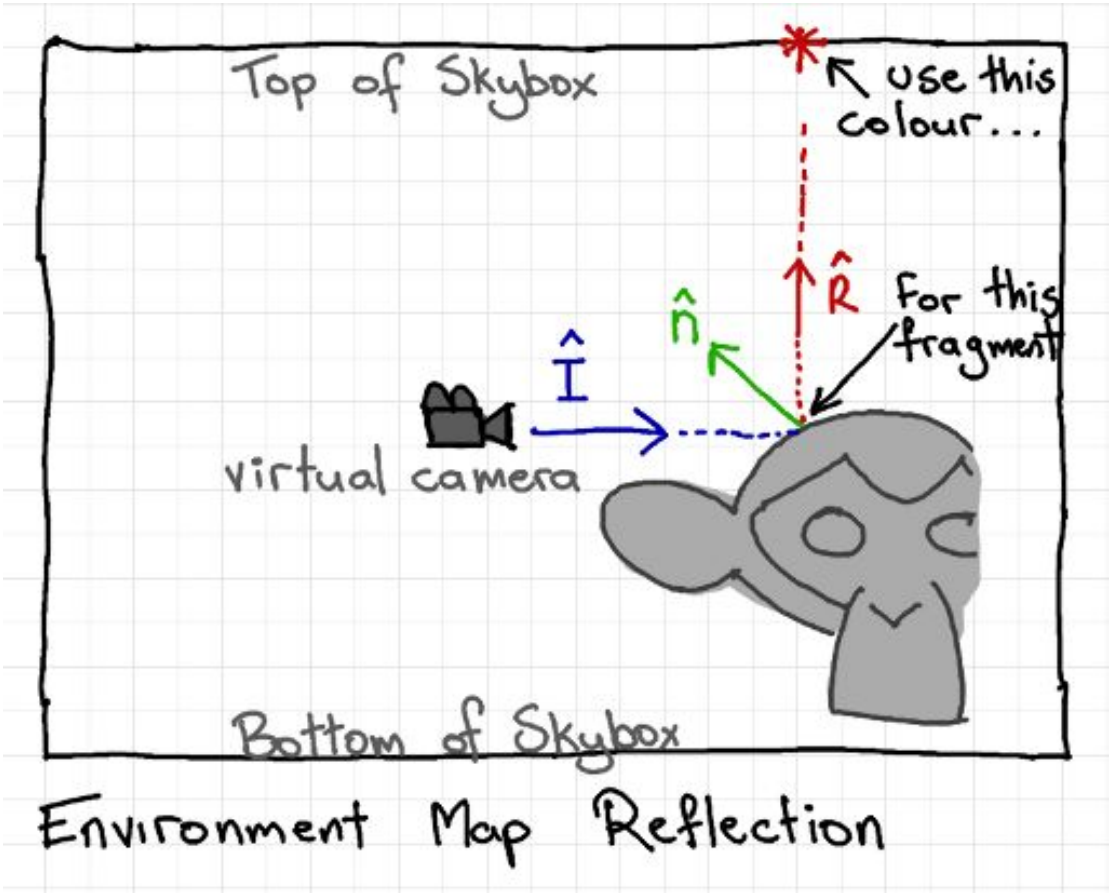


Figure 3.6.1: Visual example of mapping the skybox texture onto the reflective object

Similarly for refraction, we can specify a dynamic texture mapping such that it will refract light convincingly by bending the light rays. More specifically, we perturb the eye-to-surface vector about the surface normal, instead of reflecting it. The actual change in direction is governed by Snell's Law that gives us new texture mapping. This is defined for us in Three.js with simply "THREE.CubeRefractionMapping".

```
sphereTarget.mapping = THREE.CubeRefractionMapping;
```

Figure 3.6.2: Code snippet to reconstruct a refractive effect via texture mapping

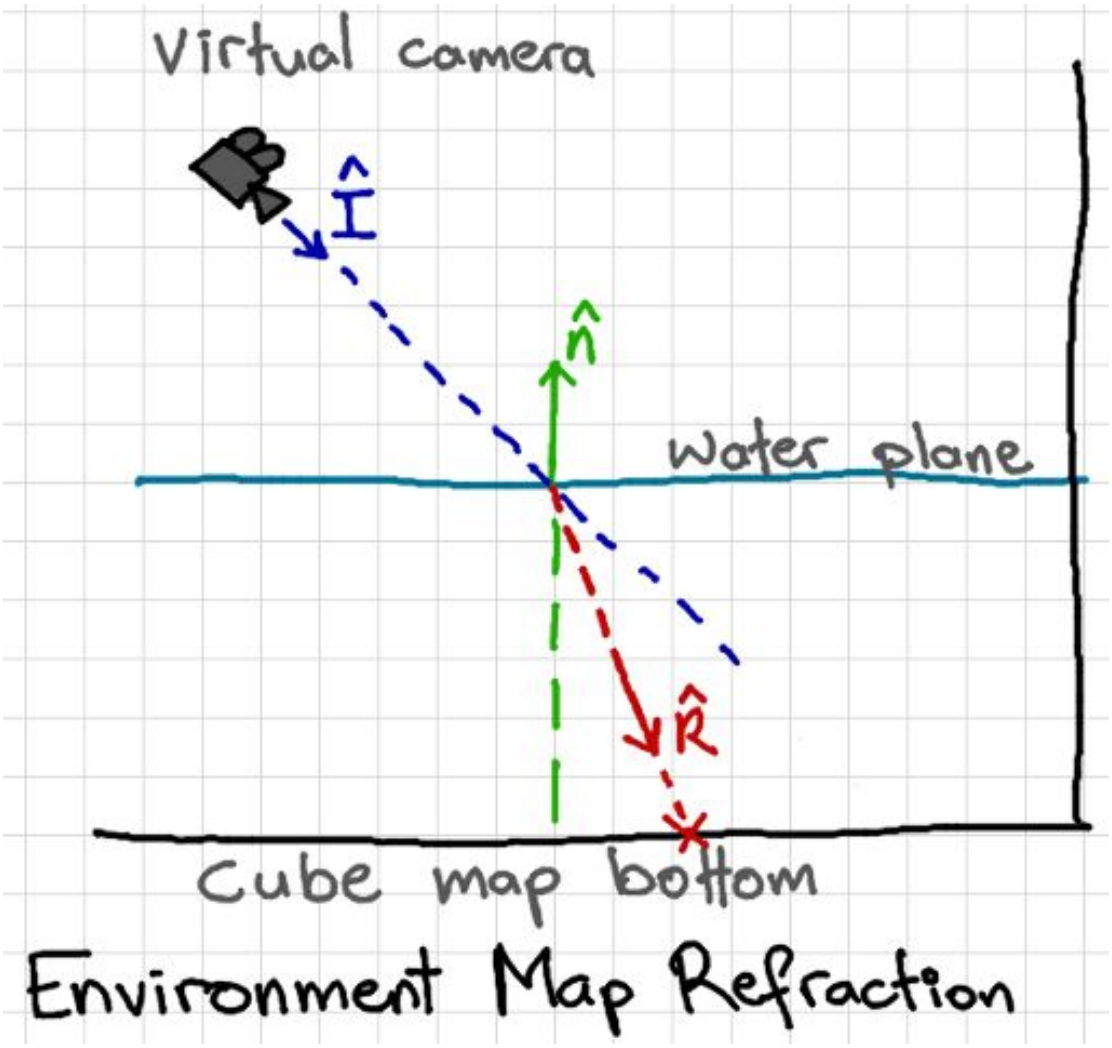


Figure 3.6.3: Visual example of mapping the skybox texture onto the refractive surface

However, after implementing these two effects, one major flaw of this implementation can be quickly noticed. Because we are using only the texture of the skybox to map out the reflective and refractive objects, we will no longer be able to see other object in the scene in the reflection or refraction.

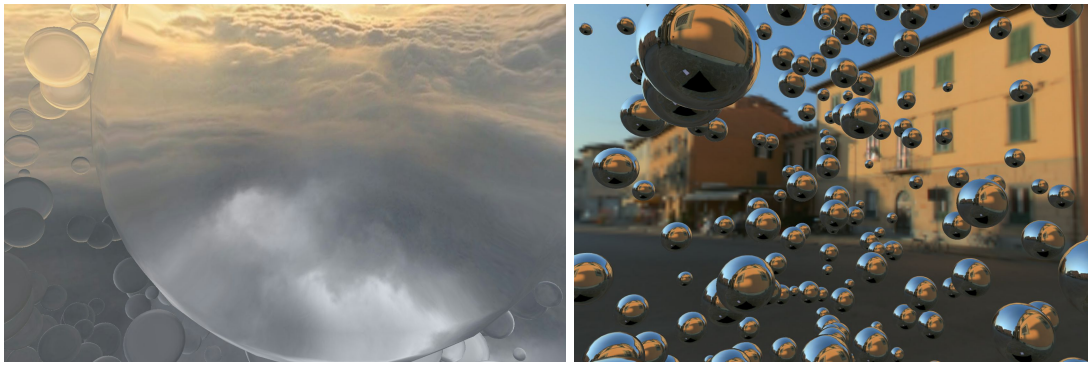

Figure 3.6.4: Visual limitation of doing a modified texture map to mimic the reflection and refraction

This demeans the quality of the refraction and refraction as this is amplified with multiple objects in the scene. To correct this effect, we can implement something called a “render to texture”. In short, this technique allows you to drop in a camera within the created scene and capture it’s current input as a texture. With 6 similar cameras, we can recreate a new cube map that renders all objects in the scene around the camera in real time.

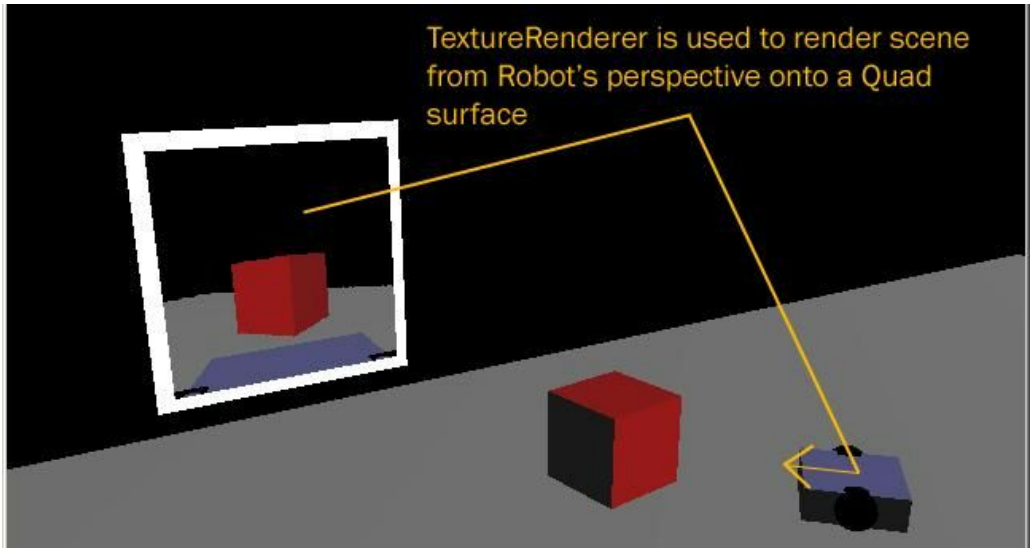


Figure 3.6.5: Illustration of how a “render to texture” can be used and works

With this added workaround, we can simulate reflections and refractions much more convincingly and ensure that it seems more physically accurate too.

```
//refraction
var sphereTarget = cameraSphere.renderTarget;
sphereTarget.mapping = THREE.CubeRefractionMapping;
var sphereMaterial = new THREE.MeshBasicMaterial( {
    color: 0xffffff,
    envMap: sphereTarget,
    refractionRatio: 0.1
} );

var torusTarget = cameraTorus.renderTarget;
torusTarget.mapping = THREE.CubeRefractionMapping;
var torusMaterial = new THREE.MeshBasicMaterial( {
    color: 0xffffff,
    envMap: torusTarget,
    refractionRatio: 0.1
} );

var coneTarget = cameraCone.renderTarget;
coneTarget.mapping = THREE.CubeRefractionMapping;
var coneMaterial = new THREE.MeshBasicMaterial( {
    color: 0xffffff,
    envMap: coneTarget,
    refractionRatio: 0.1
} );

sphere = new THREE.Mesh( new THREE.SphereGeometry(400,400,400), sphereMaterial );
cone = new THREE.Mesh( new THREE.CylinderGeometry( 1, 300, 500, 32 ), coneMaterial );
torus = new THREE.Mesh( new THREE.TorusKnotGeometry(200,50,200), torusMaterial);
scene.add( sphere );
scene.add( cone );
scene.add( torus );
```

Figure 3.6.6: Code snippet to use render to textures with refraction mapping

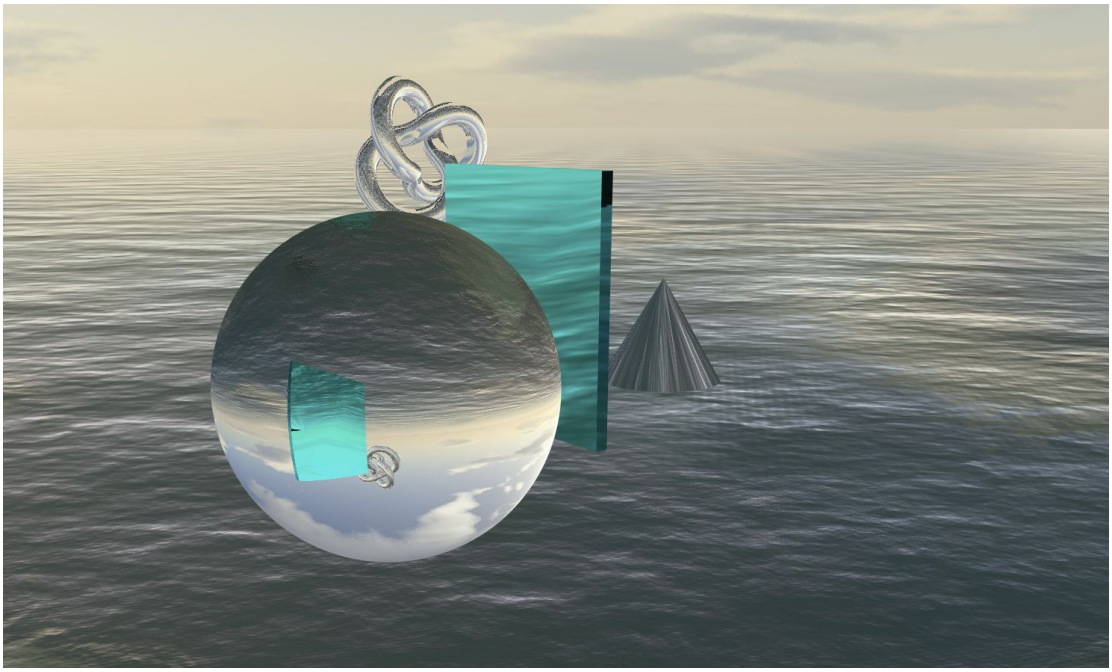

Figure 3.6.7:  Final output after implementing render to textures and reflection and refraction mapping

### 3.7. Shaders

Learning to write graphics shaders is learning to leverage the power of the GPU, with its thousands of cores all running in parallel. It's a kind of programming that requires a different mindset, but unlocking its potential is worth the initial trouble.
Virtually every modern graphics simulation you see is powered in some way by code written for the GPU, from the realistic lighting effects in cutting edge AAA games to 2D post-processing effects and fluid simulations.

A shader is simply a program that runs in the graphics pipeline and tells the computer how to render each pixel. These programs are called shaders because they're often used to control pixel and shading effects, but there's no reason they can't handle other special effects. Shaders are written in a special shading language and in this project GSLS was used due to it’s compatibility with Three.js and webGL.


Figure 3.7.1: Demonstration of the effect of shaders onto rendering

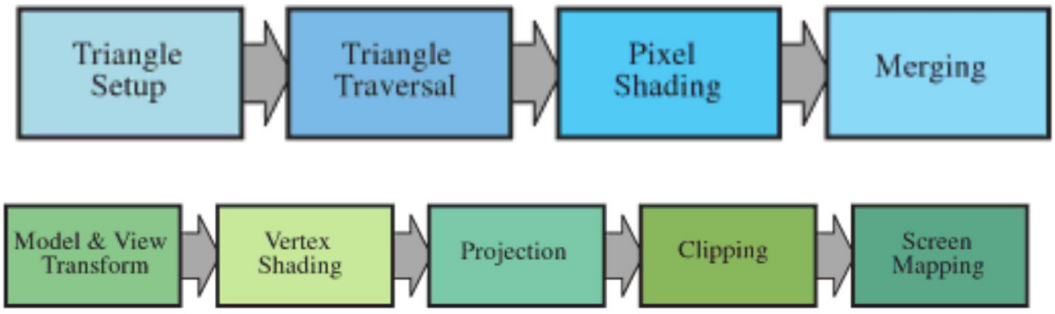


Figure 3.7..2: Pipeline of the two types of shaders (ie, fragment/pixel and vertex)

### 3.8. Water Shaders

In modeling water, we have to take note of multiple factors; specifically when we visually observe water, we always notice the water waves, surface refraction, reflection, reflectance, light absorption, light scattering and caustics.

In Three.js, these effects are handled for us and we only have to provide a number of inputs. Namely, these are the position of the point light in the scene, color of the point light, the color of the water, the normals of the waves to animate as a texture and the given texture height and width. These variables are enough for the engine to recognize and reconstruct the wave materials and provide it with the necessary effects.

```
//loading the one tile water texture
waterNormals = new THREE.TextureLoader().load( 'textures/waternormals.jpg' );
waterNormals.wrapS = waterNormals.wrapT = THREE.RepeatWrapping;

//create the water shaders and material
water = new THREE.Water( renderer, camera, scene, {
    textureWidth: 512,
    textureHeight: 512,
    waterNormals: waterNormals,
    alpha:  1.0,
    sunDirection: light.position.clone().normalize(),
    sunColor: 0xffffff,
    waterColor: 0x001e0f,
    distortionScale: 100.0,
} );

//applying the material onto a plane that simply looks like terrain
mirrorMesh = new THREE.Mesh(
    new THREE.PlaneBufferGeometry( parameters.width * 500, parameters.height * 500 ),
    water.material
);

//adding the reflection and refraction to the water
mirrorMesh.add( water );
mirrorMesh.rotation.x = - Math.PI * 0.5;
scene.add( mirrorMesh );
```

Figure 3.8.1: Code snippet defining the context of water in the project

### 3.9. Lens Flares

Using multiple textures, we can simulate a lens flare. Basically, we cant to calculate two conditions. If the camera is within the cone of the lens flare effect and where to place the added lens flare textures when the camera is considered to be in the camera's range.

We can do that by first checking where the position of the light source and add the brightest texture there. After which, we calculate the position of the light source causing the lens flare and generate a line that projects the textures/ghost features accordingly at runtime.

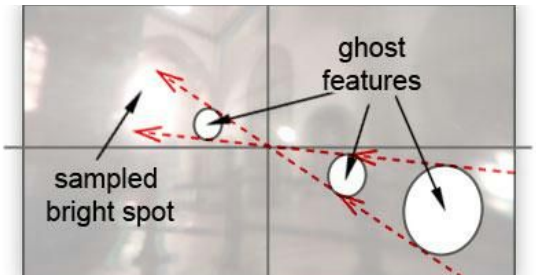


Figure 3.9.1: Illustration to show how to calculate where the lens flare features are

```
//lens flare function
function addLight( h, s, l, x, y, z ) {
    var light = new THREE.PointLight( 0xffffff, 1.5, 2000 );
    light.color.setHSL( h, s, l );
    light.position.set( x, y, z );
    scene.add( light );
    var flareColor = new THREE.Color( 0xffffff );
    flareColor.setHSL( h, s, l + 0.5 );
    var lensFlare = new THREE.LensFlare( textureFlare0, 700, 0.0, THREE.AdditiveBlending, flareColor );
    lensFlare.add( textureFlare2, 512, 0.0, THREE.AdditiveBlending );
    lensFlare.add( textureFlare2, 512, 0.0, THREE.AdditiveBlending );
    lensFlare.add( textureFlare2, 512, 0.0, THREE.AdditiveBlending );
    lensFlare.add( textureFlare3, 60, 0.6, THREE.AdditiveBlending );
    lensFlare.add( textureFlare3, 70, 0.7, THREE.AdditiveBlending );
    lensFlare.add( textureFlare3, 120, 0.9, THREE.AdditiveBlending );
    lensFlare.add( textureFlare3, 70, 1.0, THREE.AdditiveBlending );
    lensFlare.customUpdateCallback = lensFlareUpdateCallback;
    lensFlare.position.copy( sunPos );
    scene.add( lensFlare );
}

//update function for the lens flare so it changes as the camera changes view
function lensFlareUpdateCallback( object ) {
    object.position.copy(sunPos);
    var f, fl = object.lensFlares.length;
    var flare;
    var vecX = -object.positionScreen.x * 2;
    var vecY = -object.positionScreen.y * 2;
    for( f = 0; f < fl; f++ ) {
        flare = object.lensFlares[ f ];
        flare.x = object.positionScreen.x + vecX * flare.distance;
        flare.y = object.positionScreen.y + vecY * flare.distance;
        flare.rotation = 0;
        //updates the flare colour if needed
        flare.color.set(0xffffff);
    }

    object.lensFlares[ 2 ].y += 0.025;
    object.lensFlares[ 3 ].rotation = object.positionScreen.x * 0.5 + THREE.Math.degToRad( 45 );
}
```

Figure 3.9.2: Code snippet for the implementation for lens flares

### 3.10. Sunset effect

After implementing the water and lens flare effects, we started to apply run time shading to the cube map too. The intention was to give the ability to transit the scene from a daytime setting to an evening one (ie, simulating sunsets). To do so, we added shaders for the cube map and added in variable on key controls to change the position of the predefined sun.

This allows us the change the effect dramatically as seen below. To interact with the scene simply press down/s to set the sun and up/w to cause a sun rise.

Figure 3.10.1: Demonstration of before and after the sunset

```
<script type="x-shader/x-fragment" id="fragmentShader">
    //shader script for the sky box cube texturing
    uniform samplerCube tCube;
    uniform float tFlip;
    uniform vec4 darkness;

    varying vec3 vWorldPosition;


    #include <common>
    #include <logdepthbuf_pars_fragment>

    void main() {
        gl_FragColor = textureCube( tCube, vec3( tFlip * vWorldPosition.x, vWorldPosition.yz ) ) - darkness;
        #include <logdepthbuf_fragment>
    }

</script>
```

```
//pressing the up button causes a sunrise and conversely for the down button
function onKeyDown ( event ) {
// the keycode of the key pressed
switch( event.keyCode ) {
    case 40: // down
    case 83: // s

        if (sunColor.x < 0.7){
            sunColor.x += 0.05;
            sunColor.y += 0.05;
            sunColor.z += 0.05;
        }

        if (sunPos.y >1000){
            sunPos.y -= 75;
        }

        break;

    case 38: // up
    case 87: // w

        if (sunColor.x > 0.0) {
            sunColor.x -= 0.05;
            sunColor.y -= 0.05;
            sunColor.z -= 0.05;
        };

        if (sunPos.y <3000){
            sunPos.y += 75;
        }

        break;
    }
}
```

Figure 3.10.2: Code snippet in triggering the sunset interactive feature

### 3.11 Solid Interactions

Since we weren't going to do elastic deformations, we explored other ball-interaction forms including metaballs - squishy objects having an organic appearance that can "merge" in and out of each other.

#### 3.11.1. Metaballs & Isosurfaces

Metaball is a subset of isosurfaces, representing isosurfaces in 3-D. An isosurface is created using a threshold value and a function which translates every point in a multi-dimensional space to a scalar form. The Isosurface is composed of every point in the space where the function value equals to the threshold value.

So how do we render a meta-ball in 3D space? Here's an algorithm:

1. Divide the 3-D space into small cubes;
2. For every pixel in 2D, cast a ray from the camera; at each cube the ray goes through, we evaluate the function value by doing the following:
3. Iterate through every meta-ball, calculate the function values (using one of the corners of the cube), and add them together;
4. If we find a cube (point) along the ray that is above the threshold, we know we have hit the meta-shape;
5. We then determine the color, plot it and move on to the next pixel.

**Equations for a meta-shapes in 3D space:**

The equations are based on the equation for calculating strength of an electrical field in physics.

**1. Meta-spheres**

$$f(x,y,z) = \frac{1.0}{(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2}$$

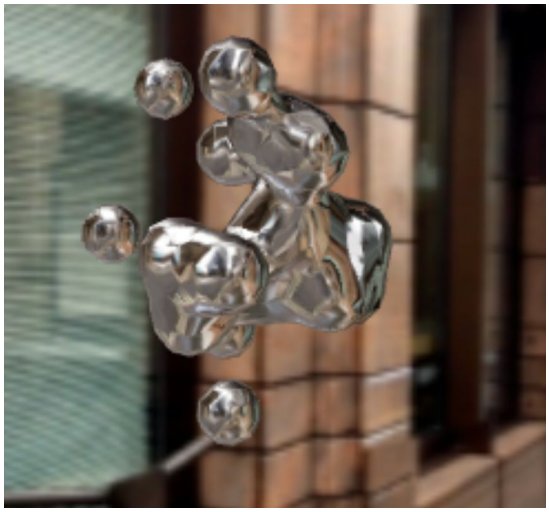

Figure 3.11.1: Meta-spheres

**2. Meta-ellipses**

$$f(x,y,z) = \frac{1.0}{a*(x-x_0)^2 + b*(y-y_0)^2 + c*(z-z_0)^2}$$

The factors a, b and c are scalars.

Figure 3.11.2: Meta-ellipses

**3. Meta-diamonds**

$$f(x,y,z) = \frac{1.0}{|x-x_0| + |y-y_0| + |z-z_0|}$$

Figure 3.11.3: Meta-diamonds

### 3.11.2. Optimizations

Rendering isosurfaces is slow if we use the naive algorithm of iterating through every small cube and every meta-shape to compute the values. Here we will list down a few optimization tricks we implemented.

**Changing the fall-off function:**

For every point in the 3D space, we have to compute a quadratic function for every meta-shape; and the division operation of the function is quite costly. So we replaced the electrical field of the point charge function:

$$f(x, y, z) = \frac{1.0}{(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2}$$

with another polynomial, say:

$$f(x, y, z) = (1 - ((x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2))^2$$

When the distance between (x, y, z) and (x0, y0, z0) is smaller than 1, this equation decreases non-linearly similar to our electric field equation, thus preserving the “blobbiness” of the meta-shapes. This equation also has the nice property of evaluating to 1 (instead of infinity in the case of our original equation) where (x, y, z) is at (x0, y0, z0).

**Clamping the value**

Also we are clamping the value of the equation to 0 when the distance between (x, y, z) and (x0, y0, z0) is larger than 0.9. The electric field contribution due to a charge no longer covers the entire space, we approximates it to zero when the point is too far away from the electric charge. This means that each charge has a limited sphere of influence. So as we move along our rays, we only have to add the values of the meta-shapes for which the current point on the ray is inside the sphere of influence. This saves a lot of computations.

### 3.11.2. Marching Cubes algorithm

We now have an efficient way of calculating the values for every small cube in space, but this will only create a discrete rendering. To smoothen it out, we implemented the marching cubes algorithm.

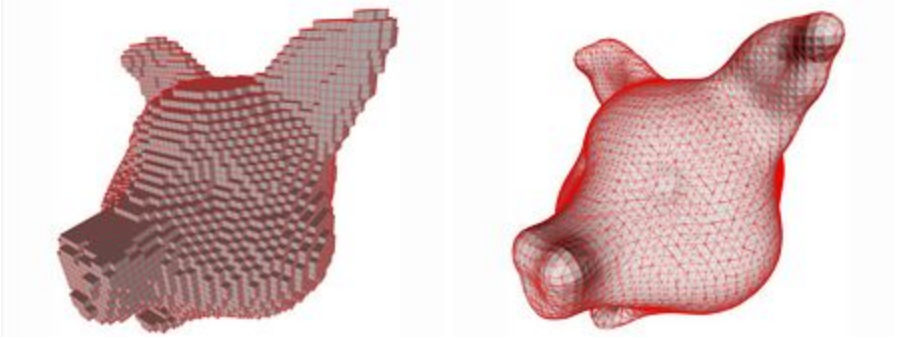

Figure 3.11.4: Visualization of how the marching cubes algorithm works.

For every cube where the value is evaluated, if any of its vertices is above above a certain threshold, we know that the surface must have crossed the cube in some manner. The total number of ways a surface can cut a cube is 256. The following diagram presents 14 main cases; where rotation or symmetrical operations on the cube will result in the said 256 scenarios.

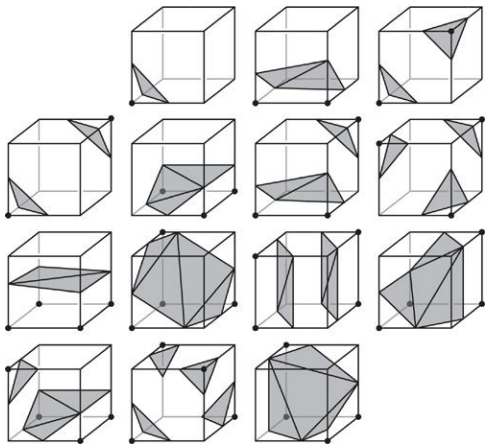

Figure 3.11.4: 14 main cases a vertice can slice the cube.

After choosing one case out of the 256 scenarios, we then estimate the position of the point on the edge by comparing the value of the isosurface function evaluated at the two end points of the edge. It is  implemented using this code snippet:

```
function gen_point(v1, v2, p1, p2){
  var percent= (threshold - v1) / (v2 - v1);
  return p1.clone().lerp(p2, percent);
}
```

Figure 3.11.5: Code snippet of point estimation.

We obtained the  triTable(translating points on edges to triangles) and the edgeTable(an array indexing the 256 scenarios of surface cutting across cubes) from @greggman on github and implemented rest of the algorithm in our code.

## 4. Conclusion

The WebGL application will show how we combined these features together in our project. Though tedious with much unexpected obstacles along the way, we have learned many other computer graphics algorithms and techniques outside of lessons from this final project!

## 5. References


1. http://nehe.gamedev.net/tutorial/collision_detection/17005/
2. http://antongerdelan.net/opengl/cubemaps.html
3. http://www.rastertek.com/dx11tut29.html
4. http://john-chapman-graphics.blogspot.sg/2013/02/pseudo-lens-flare.html
5. Exploring Metaballs and Isosurfaces in 2D by Stephen Whitmore
6. Metaballs by Ryan Geiss