Rendering objects with effect of refraction, reflection and lighting

Abdullah bin Abdullah,, Louis Bennette, Leyla Mammadova

Github repository: https://github.com/Maths-Assignment2/LightRendering
Video link: https://youtu.be/ny_T3JI6_gQ

Table of contents

Introduction

The aim of this project was to simulate light reflection and refraction on objects in real time. Originally we started building our code using the octet opengl system of scene's and meshes, But quickly we changed system as we narrowed down our task and decided to implement our own opengl code from scratch. This gave us more freedom over the mesh data structures and the shader code, which was what needed. We built the project in a custom rendering pipeline built with parts of octet.
The jobs were split mostly as follows although much collaboration took place as the code base is not very large:
Abdullah; octet rendering, opengl rendering, shader work, phong lighting shader.
Louis; opengl rendering, obj reader, opengl data structure, shader work and GL projection code.
Leyla; skybox, cubemap textures, reflective & refractive calculations for shaders, shader work.

Abdullah Bin Abdullah

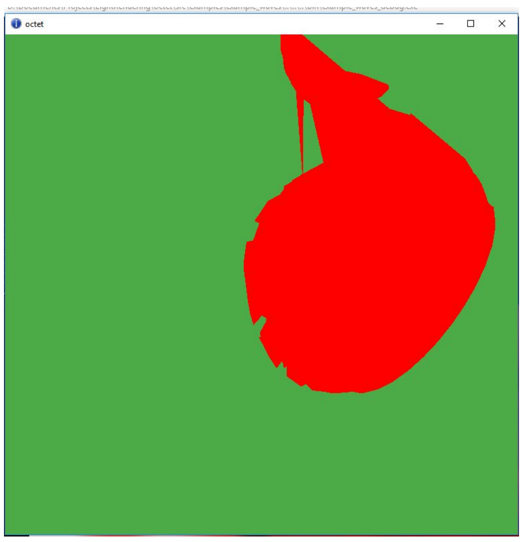
Inspiration:

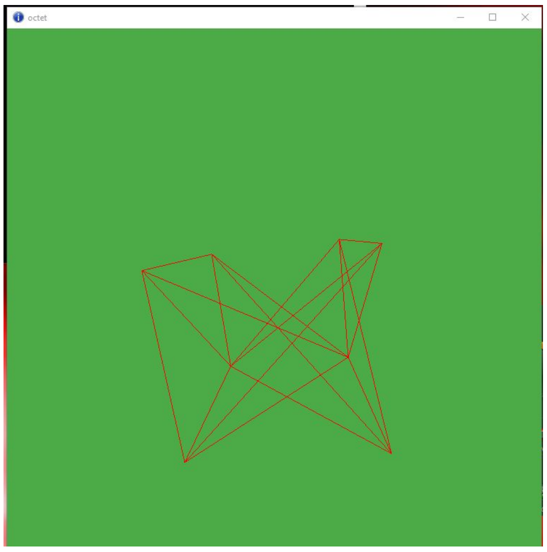
The project is inspired by Evan Wallace's WebGL Water demo (Wallace, 2011). We wanted to emulate the interesting light effects that Evan's demo featured using octet to achieve it.

Implementation:

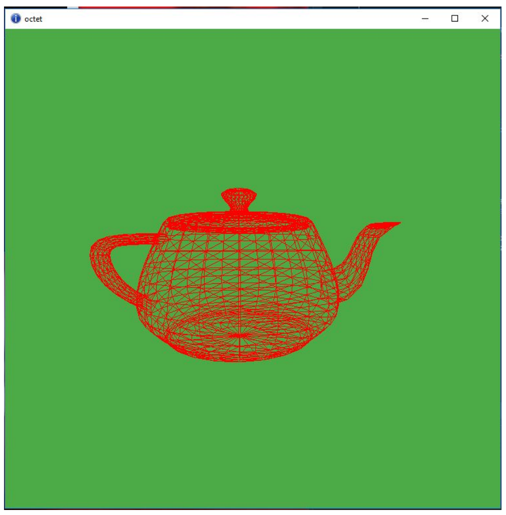
We started the project by using my previous waves demo as a base. As the project went on we encountered problems with using octet due to a lack of understanding. It was decided then that we would use OpenGL calls in order to make things simpler.

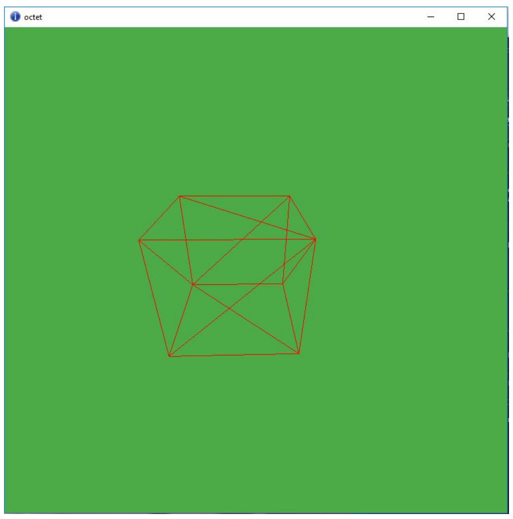
While trying to implement Louis' file loader, we encountered bugs when OpenGL started drawing the models.





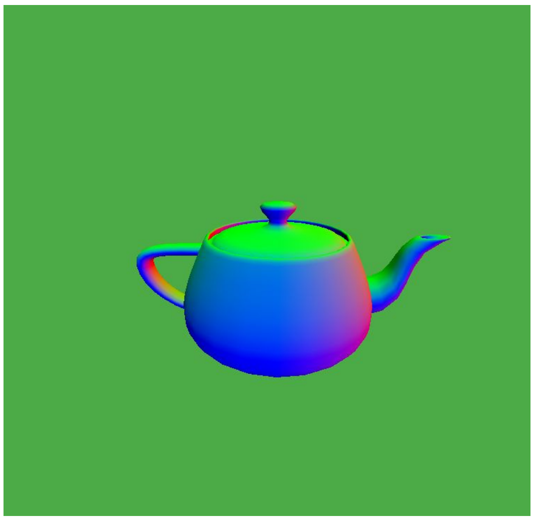
These were caused by the .obj file starting their indices count at one for the faces. We had to either modify the .obj file or subtract one from the data each time we store it for our own uses. The following screenshots show the files successfully loaded into the program.



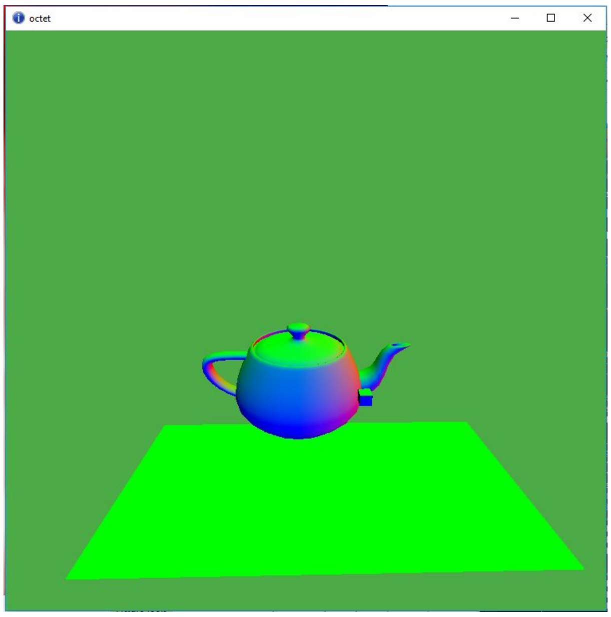


Implementing Normals into the Vertex Data:

The next step was to implement the normals supplied by the .obj files. The way the .obj file is arranged, only unique data was stored for the vertices, texture coordinates and normals. The faces section of the .obj files contains the indices to how the data in the files are meant to be read. Initially it was thought that we could read the data from .obj file as-is but we had to process it into a format that OpenGL uses. Louis implemented a parser to handle this information and convert it to a format that OpenGL uses. Once the files were loaded in correctly again, we needed to find out if the normals were loaded in correctly. We verified it by assigning normal positions to be read as colour by the fragment shader.

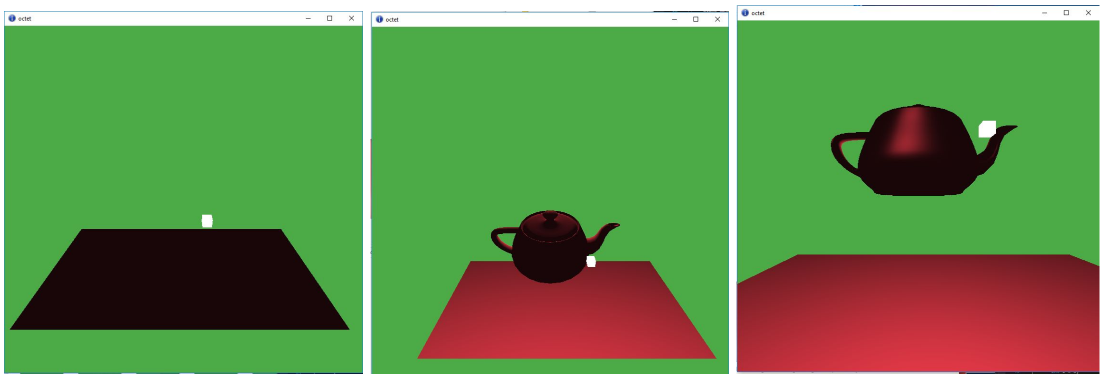


After cleaning up the entity class so that it can function separately from the obj_file_io class, I started working on my plane_mesh to test loading an object that doesn't use the obj_file_io class.
The plane_mesh class' only function is to create the data necessary to pass into the entity class so that it would draw it.



I separated reading in the shader source code from the entity class as well so each object could have it's own set of custom shaders if needed. Although this is inefficiant because each entity object will have its own shader program linked to it. Once I was done reorganizing how the entity reads in data and separating it from the obj_file_io, I started to work on implementing lighting to the scene through shaders.

BASIC LIGHTING:



All of the lighting code for this part of the project is stored in the fragment shader because all we needed to do is output the final colour of the fragment. I assigned some uniforms in the shader to get important data, such as the light position and colour, object colour and the camera position.

Ambient lighting was the first to be implemented. I simply created a variable called "ambient" and stored the light colour multiplied by the ambient intensity. The next to be implemented is the diffuse lighting. This is achieved by taking the dot product of the normal and light direction vectors and then multiplying it by light colour. Specular lighting was the last to be implemented for the plane shader.

First we set the specular intensity, view direction and reflect direction. The view direction is the normalized value between the camera and the fragment position. The reflection direction is the result of the built-in reflect function in GLSL that takes in the inverse light direction and the normal. We then get the specular value by calculating the dot product of the view direction and the reflect direction and then raise it by the power of the shininess value. In this case, it is 32. The specular value is then multiplied by the specular intensity and light colour. When the ambient, diffuse and specular lighting are calculated, they are added together and multiplied by the object colour (de Vries, n.d.). This data is then sent to gl_Fragcolor to be processed.

Louis Bennette

Scene Projection:

This initial part of the project required us to set up a 3D scene. We needed a scene as we intended to be able to move the camera around. To view the object from different angle to set how the lighting changed. To do this we took what we had learn't from our maths lectures and implemented our own vertex to project matrix chain. The scene entities each have their own transform, as well as the camera. We use a perspective projection matrix to render the object's on screen.

gl_Position = (cameraToProjection * worldToCamera * modelToWorld) * vec4(pos,1.0);

We do this calculation in the vertex shader as we need the worldToCamera and modelToWorld transforms for lighting calculations as well.
The projection is built using the octet frustum matrix, which creates a projection matrix for use with desired near and far clipping planes as well as field of view etc.

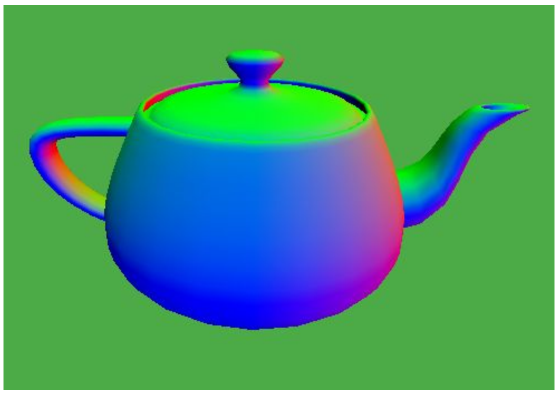
Entities:

The entity class encompasses most of the rendering openGL calls. It's job it to hold the entity's transform data, load the entities data into the GPU as well as load the right shaders and finally render the object using the shaders.
The entity class is a handle to enable the other programmers to very simply load an obj file with a shader and output the result with any scene translation.

The rendering was done using part of Abdullah's work on rendering a moving sin wave plane, which used a similar principle of compounded vertex data. Specifically position and colour. This code was modified and changed to use normals instead of a colour and was a great help to render more complex models thanks to it's index based rendering calls.
The objects are rendered using specular and diffuse lighting, the class also has become responsible for providing the correct variables to the shader so that different objects can appear to have different materials

Obj File Reader and Parser:

In order to render large polygons opengl expects mesh data to be loaded into VRAM a certain way, every vertex has extra data associated with it, position, texture coords and normals. Additionally to construct the mesh openGL requires a list of vertex indices, to know which vertices to combine to make a surface.
The flat face rendering technique expects a surface to have one normal, however openGL expects the vertices to store the normal data. This means that when a vertex shares a connection with more than one face, it has different normals for each face. This is handled by the parser by duplicating vertices with different normals and re assigning the indices to point to the correct vertices. Using the model's vertices with normals we tested that the models were loaded correctly by setting to output colour to the normal value of the nearby vertices.



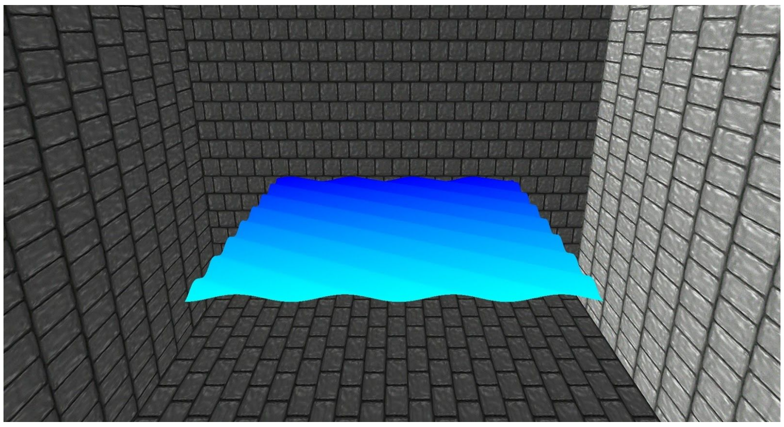
The .obj parser uses a custom algorithm which is very inefficient, this is because of time shortage during the implementation. The algorithm needs to re check it's entire set for every vertex. The algorithm works on an order n^n. In future this part of the project will need work. I will continue to work in this new environment for my research project.
The parser does still do it's job (although very slowly) and multiple models can be loaded in. See fig bellow of a stanford bunny rendered with a simple phong lighting shader.

I am pleased with our result as a group, we created our own 3D rendering engine and handled model imports as well as custom shader code, reflective and refractive calculations. We had good daily communication and worked well as a team we debugged each other's errors and cooperated .

Leyla Mammadova

Skybox and cubemaps:

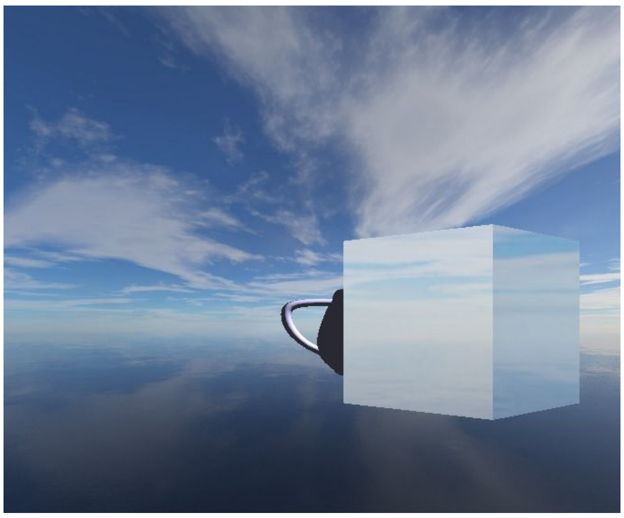
The idea that we had for this project required an advanced knowledge of OpenGL. We needed to implement things like skybox, cubemaps, lights and custom shaders so everything would work together, and the hardest part for me was to implement it in octet. I've spent decent amount of time reading how OpenGL works and learned to work with use of glsl. First of things that I've learned and tried to implement is a simple OpenGL lighting and trying to link it with octets mesh_box. The result wasn't the one that we expected. Light was distributed evenly on whole surface of the mesh. After my teammates made a custom .obj file loader I've started to search for a way to make our object transparent and have a glass effect with a bit of reflection of the surroundings. In order to implement that we needed to load a skybox and use a cubemap of that texture and apply it on the surface of the object. A cubemap is basically a texture that contains 6 individual 2D textures that each form one side of a cube: a textured cube. As seen on picture above, I applied texture with use of material function in octet. But problem that I had was that it was applying the same texture on all six cube faces. I also had problems with jpeg_decoder that is build in octet, it didn't allowed me to use the jpeg format. After we switched from default visual_scene that build in octet to a custom one, I wasn't able to use the same technique.



It required for me to build a custom skybox. I used the same pattern that was used in last year coursework of "Interactive Terrain Generation and Effective Water Simulation" made by Wanganning Wu, Alex Lux for creating a skybox. It helped me to fix the bug I had with jpeg_decoder and use the built in texture_shader. The skybox class has a construction script that draws six vertices of a cube, create cubemap() function draws the textures on the cube surfaces separately on each side and binds the textures to a uniform variable that will be used in our shaders for effects. Init() function initialize texture_shader and render() renders the skybox.

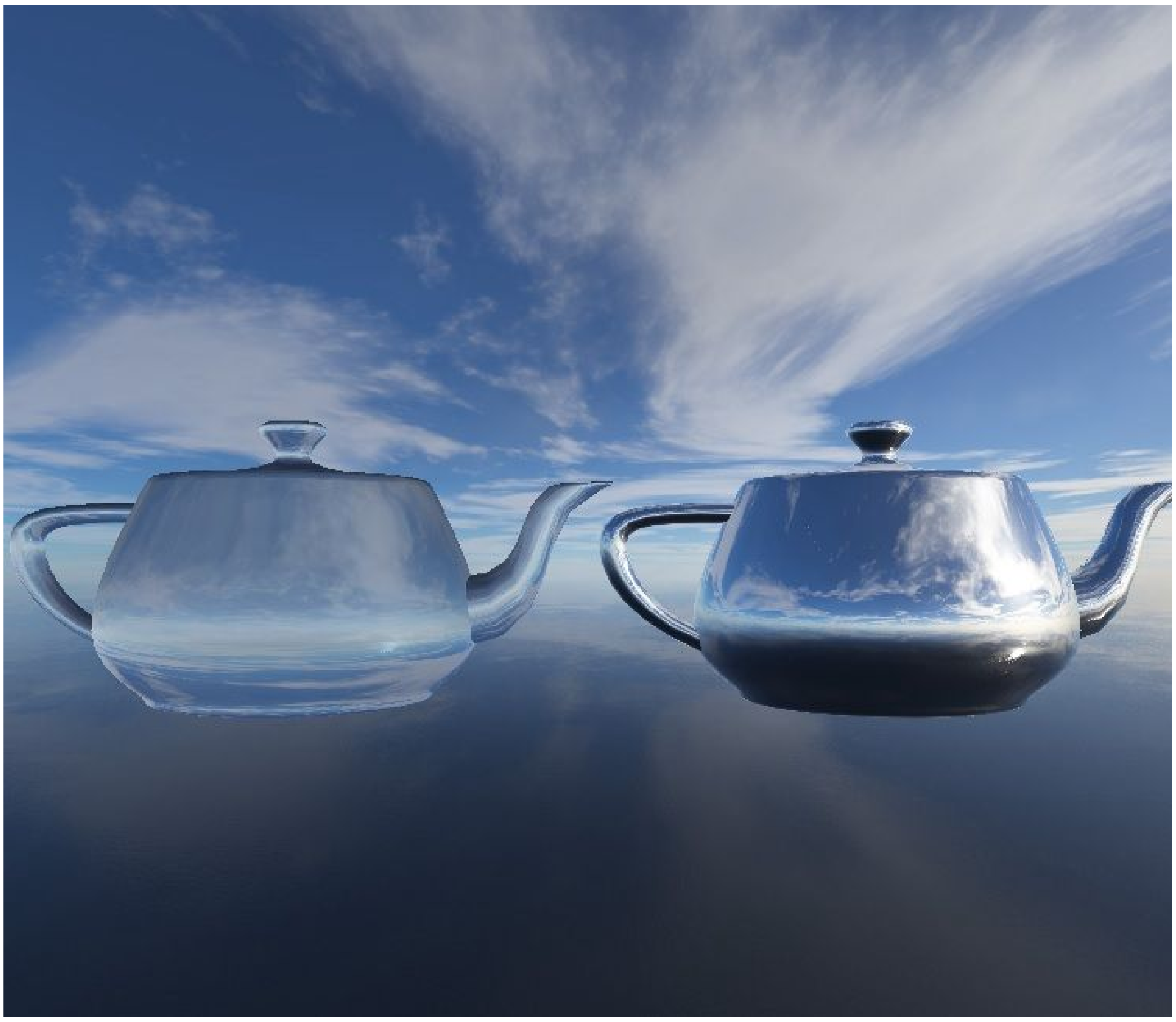
After skybox was placed we needed to implement the shaders to do the work for us. Then to test the texture application on the objects i first applied it to the box with use of code in fragment shader.

I started with a simple shader that was created by my teammate and added texture calculations.
I added the uniform value of cube_texture.
Uniform samplerCube cube_texture;



And in main function I added the calculations:

Float ratio = 1.00/1.52 // ratio is the index that determines the amount light distorts/bends of a material where each material has its own refractive index. 1.52 is the glass index
Vec3 unitNormal= normalize(rel_normal_);
we normalize the normali vector
Vec3 viewVector = normalize(vert_to_camera_); vert to camera is the distance between world position and the view position.
Then I calculated the refracted vector:
refractedVector = refract(viewVector, unitNormal,1.00/1.52);
And calculated the final color with textures:
vec4 color = textureCube(cube_texture,refractedVector);
Result wasn't as transparent and glass looking as I expected. After some manipulations with shaders I decided to put the refracted vector calculations into the vertex shader. And also calculate the reflected vector. And mix them both together to get a nice glass effect. Then Abdullah separated the reflection calculation and we applied our shaders to the teapots.



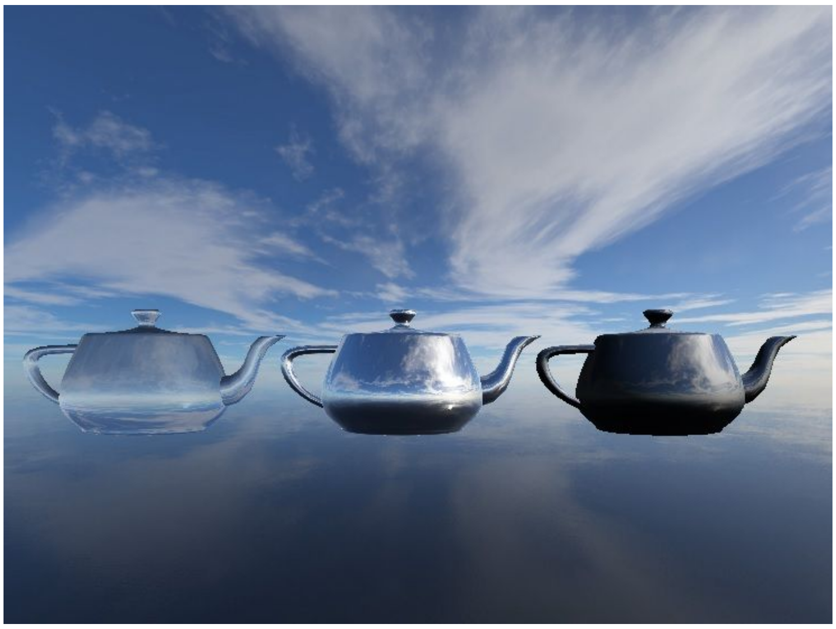
The teapot on the left has transparent shader applied to it , and the teapot on the left has a mirror shader applied to it. They use the same vertex shader but have separate fragment shaders. I will put now the whole source code.

Vertex shader:

```
uniform mat4 modelToWorld;
uniform mat4 worldToCamera;
uniform mat4 cameraToProjection;

uniform vec3 light_pos_;
uniform vec3 view_pos_;

// attributes from vertex buffer
attribute vec3 pos;
attribute vec3 normal;
attribute vec2 tex_coord;

// outputs
varying vec3 rel_normal_;
varying vec3 vert_to_light_;
varying vec3 vert_to_camera_;
varying vec3 world_pos;
varying vec3 reflectedVector;
varying vec3 refractedVector;

void make_lighting() {
  vec4 world_pos = modelToWorld * vec4(pos, 1.0);
  rel_normal_ = (modelToWorld * vec4(normal,0.0)).xyz;
  vert_to_light_ = light_pos_ - world_pos.xyz;
  vert_to_camera_ = world_pos.xyz - view_pos_;
}

void main() {
  make_lighting();

  gl_Position = (cameraToProjection * worldToCamera * modelToWorld) * vec4(pos,1.0);

  vec3 unitNormal = normalize(rel_normal_);
  vec3 viewVector = normalize(vert_to_camera_);
  reflectedVector = reflect(viewVector, unitNormal);
  refractedVector = refract(viewVector, unitNormal,1.00/1.52);
}
```
Transparent fragment shader:
```
uniform vec3 object_colour_;
uniform vec3 light_colour_;
uniform float shineDamper;
uniform float reflectivity;
uniform samplerCube cube_texture;
varying vec3 rel_normal_;
varying vec3 vert_to_light_;
varying vec3 vert_to_camera_;
varying vec3 world_pos;
varying vec3 reflectedVector;
varying vec3 refractedVector;

void main()
{

  vec4 color1 = textureCube(cube_texture,reflectedVector);
  vec4 color2 = textureCube(cube_texture,refractedVector);
  vec4 color = mix(color1,color2, 0.8); //mixing two of them so it would be transparent yet
reflect the cubemap
  gl_FragColor = color;
}
```
Mirror fragment shader:
//same variables
```
void main()
{
  gl_FragColor = textureCube(cube_texture,reflectedVector);
}
```

Fragment shader with diffuse light:
Louis has made a shader with diffuse light calculations. I took his shader implementation and mix it with the mirror fragment shader.

```
uniform vec3 object_colour_;
uniform vec3 light_colour_;

uniform float shineDamper;
uniform float reflectivity;
uniform samplerCube cube_texture;

varying vec3 rel_normal_;
varying vec3 vert_to_light_;
varying vec3 vert_to_camera_;
varying vec3 world_pos;
varying vec3 reflectedVector;
varying vec3 refractedVector;

void main()
{
        float brightness = max(dot(normalize(rel_normal_), normalize(vert_to_light_)), 0.1);
        vec3 diffuse = brightness * light_colour_;

         vec4 result = vec4(diffuse,1.0) * textureCube(cube_texture,reflectedVector);

         gl_FragColor = result;
}
```



I applied to the third teapot and this is result that I had. It means that our shaders are working with any light that can be calculated.

Bibliography

de Vries, J. (n.d.). *Basic Lighting*. [online] Learnopengl.com. Available at: https://learnopengl.com/#!Lighting/Basic-Lighting [Accessed 22 May 2017].

de Vries, J. (n.d.). *Cubemaps*. [online] Learnopengl.com. Available at: https://learnopengl.com/#!Advanced-OpenGL/Cubemaps [Accessed 22 May 2017].

de Vries, J. (n.d.). *Framebuffers*. [online] Learnopengl.com. Available at: https://learnopengl.com/#!Advanced-OpenGL/Framebuffers [Accessed 22 May 2017].

Lighthouse3d.com. (2012). *Directional Lights I*. [online] Available at: http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/directional-lights-i/ [Accessed 22 May 2017].

Blog.csdn.net. (2017). *Modelling of glass surfaces （1） - EricXu1990-CSDN博客 - 博客频道 - CSDN.NET*. [online] Available at: http://blog.csdn.net/u011417605/article/details/70173433 [Accessed 22 May 2017].

Wallace, E. (n.d.). *WebGL Water*. [online] Madebyevan.com. Available at: http://madebyevan.com/webgl-water/ [Accessed 22 May 2017].