

# **Computer Graphics Final Report**

## **Team 8**

**20171105** 문장원

**20191170** 양우림

**20191332** 홍현기

## Introduction/Motivation

Our team embarked on a quest to explore and visualize common but interesting natural phenomena that are easily observable in our daily lives. By collaboratively brainstorming and sharing different perspectives, we found some interesting examples. First, we observed a straw bend in the coffee cup where the coffee was served. Second, when the laser was shot into the fluid, we noticed a bend in the direction of the laser's progress. Third, when we examined the gamak pond, we observed a clear difference between the depth of the water we perceived and the actual depth indicated by the actual sign.

These observations have led us to delve deeper into the principles of refraction. These events are routine, but the principles contained in them are complex and profound. It involves the phenomena of reflection as well as refraction, and we realize that a more detailed understanding is needed. Our goal is to explore how these natural phenomena can be expressed through computers, beyond simply understanding them. This provides an opportunity to shed light on the incredible complexity of light-material interactions from a broader perspective.

## Specifications

Before we could begin the actual implementation of our project, it was essential to establish a suitable environment. To accomplish this, we utilized a combination of Python and several critical packages, namely pyopengl, numpy, and math, each playing a pivotal role in our setup. The foundational step involved creating a representation of the water tank, which necessitated an .obj file composed of a mesh structure. This requirement was aptly addressed by employing Fusion360, a decision driven by its simplicity and effectiveness.

For the actual output of the water tank model, we reused the rabbit output function previously utilized in assignment 3. The design of the implemented model was deliberately kept straightforward, opting for a basic rectangular parallelepiped shape. While a more intricate representation was certainly possible, we selected simplicity to reduce computational demands. However, to enhance the visual appeal and realism, we added a transparent and blue texture to the model(water tank), simulating the appearance of a natural water block. These textured outcomes are evident in the results section of our project.

In preparing for the model's implementation, we faced several considerations, particularly regarding the use of ray tracing. At the program's commencement, rays are emitted, with their subsequent interactions being contingent on whether they intersect with the water tank, the straw, or nothing at all. When a ray penetrates the water tank, Snell's law is applied to refract the light, and the point of refraction serves as a new starting point for the ray's continued journey. If a ray encounters the straw, it is allowed to proceed to its intended destination. Conversely, if it does not intersect with any object, it travels in a straight path, obviating the need for further calculations. This planning and setup were instrumental in the successful execution of our project, allowing us to accurately simulate and visualize the interplay of light, objects, and refraction.

## Approach

We decided to utilize the raycasting mechanism to implement refraction. At the camera's coordinates, a ray is shot for each pixel, it moves through refraction, and when it encounters an object, it is displayed on the original destination pixel. Below is our pseudocode.

```

set each rays' value for each pixel
while rays:
    x = closest point
    if (x == refracting area):
        apply snell's law
    else if (x == object):
        display to original destination
    else:
        break

```

## Implementation

To implement our project, we used the code from the end of assignment 2. Since our goal was to implement refraction through OpenGL, we started with a basic OpenGL environment. Additionally, the implementation of the raycasting method we will use requires a single starting viewpoint, so we assumed that situation by setting the fov as 90. In this state, we implemented it in three steps.

First, create rays corresponding to all pixels. The starting point is at the camera, and the end point corresponds to each pixel. Since our window's width and height range from 0 to 400, we need to create 401\*401 rays. Since the camera's viewpoint is (0, 0, 1) at the first viewpoint and the fov is set to 90, the rays corresponding to each pixel are as follows.

```

rays = np.array([[[i / 100, j / 100, -2, 0] for j in range(-200, 201)] for i in range(-200, 201)])

```

We also need to adjust the direction of the light rays so that it works even when the viewpoint is different. Therefore, we adjusted the direction through the code below.

```

t = SubWindow.windows[3].viewMat.T
camera = t[:3, 3] + t[:3, 2]
x = np.matmul(t, rays.reshape(160801, 4).T)[:3].T

```

The next step is to find all the intersection points between the starting point and the ending point and find the closest point among them. There are two cases of intersection. First, it is a situation where the object passes through a space with a different refractive index. In this case, the intersection point corresponding to the plane of the space must be found. We used the code below to check all refractive space objects for a ray. Check that it passes through each side of the refractive space and check whether the point is inside the plane. Here, the return value x contains information about the position of the intersection, the refractive index of the refractive space, and the normal vector.

```

for obj in SubWindow.obj_list:
    t = obj.checkSujo(start, dest)
    if t:
        x = t[1:]
        x.insert(0, t[0][1])
        x[2] = t[2][0]

```

```

points.append(x)
if len(t[0]) == 3:
    x = t[1:]
    x.insert(0, t[0][2])
    x[2] = t[2][1]
    points.append(x)

```

Checking function inside the refracting area class

```

def checkSujo(self, start, dest):
    t = self.mat
    transv = np.matmul(t, np.append(np.array(vsuj).T, np.array([1 for _ in
range(len(vsuj))])).reshape(4, len(vsuj)))
    mx, Mx, my, My, mz, Mz = min(transv[0]), max(transv[0]), min(transv[1]), max(transv[1]),
min(transv[2]), max(transv[2])
    x, y, z = start[0], start[1], start[2]
    nx, ny, nz = dest[0], dest[1], dest[2]
    v = dest - start
    coor = [1234]
    normal = []
    if x < mx < nx or x > mx > nx:
        d = abs((mx - x) / v[0])
        if my < y + v[1] * d < My and mz < z + v[2] * d < Mz:
            coor.append(np.array([mx, y + v[1] * d, z + v[2] * d]))
            normal.append(np.array([-1, 0, 0]))
    if x > Mx > nx or x < Mx < nx:
        ...
    if len(coor) > 1:
        return [coor, self.index, normal]
    else:
        return False

```

The following is a situation where the target object is passing by. We tried to implement a situation where we could pass through a mesh object, but we failed. Instead, we thought that the object was composed of points and found the intersection with these points. To do this, the cross product of the vector connecting the starting point and the end point and the vector connecting the starting point and the object point must be 0. However, the situation where it was completely 0 was rarely a situation for us who assumed the object was made up of multiple points, and we used approximation to calculate points that were close to 0. Calculating the cross product alone also applies to points in the opposite direction. Therefore, by performing the dot product calculation, only the dots in the same direction are calculated. Overall, we were able to find the intersection with the target object with the code below.

```

y = np.cross(stickpoints - start, dest - start)
y = list(map(np.linalg.norm, y))
j = y.index(min(y))
if min(y) < 0.01:
    if np.dot(stickpoints[j] - start, dest - start) > 0:

```

```
points.append([stickpoints[j], 0, j])
```

Among these points, find the point closest to the starting point.

The next step depends on the closest point. First, this is the case where there is no intersection. This means that there are no objects to display, so we exit with a break and examine the next ray. The next case is the plane of the refractive space. In this case, the starting and ending points of the ray must be adjusted to create a situation of refraction. Below is a code implemented using vector operations using OpenGL and numpy according to Snell's law.

First, check whether the direction is going out within the refraction space, and if so, change the direction of the normal vector. Afterwards, calculations are made according to Snell's law. It is also divided into two operations based on the conditions under which total reflection occurs. After this calculation, the intersection that touches the refraction surface is changed to a new viewpoint. Also, the new end point is changed according to the changed angle. Afterwards, go back to the step of finding the closest point to the new ray and repeat.

```
original_direction = (dest - start)
normal = closest[1][2]
new_index = closest[1][1]
if np.dot(normal, original_direction) > 0:
    normal = -normal
    new_index = 1
v, vnorm = -original_direction, np.linalg.norm(-original_direction)
w, wnorm = normal, np.linalg.norm(normal)
theta1 = np.degrees(np.arccos(np.dot(v, w) / (vnorm * wnorm)))
if (current_index * np.sin(np.radians(theta1)) / new_index) > 1:
    theta_diff = 2 * theta1 - 180
    new_index = current_index
else:
    theta2 = np.degrees(np.arcsin(current_index * np.sin(np.radians(theta1)) / new_index))
    theta_diff = theta1 - theta2

axis = np.cross(original_direction, -normal)

glMatrixMode(GL_MODELVIEW)
glPushMatrix()
glLoadIdentity()
glRotatef(theta_diff, axis[0], axis[1], axis[2])
dest = np.matmul(np.append(dest - closest[1][0], [0]), glGetDoublev(GL_MODELVIEW_MATRIX))[:3]
+ closest[1][0]
glPopMatrix()

start = closest[1][0]
current_index = new_index
```

The last case is when the target object point is reached. In this case, it means that the path of the ray we shot meets the object. Therefore, it displays the original destination corresponding to the pixel

of the original ray. Finally, when the process is completed for all rays, the points that need to be marked are displayed.

## Results

Our result consists of 4 screens, xy plane, yz plane, xz plane and 3D space. When you press right click on the 2D spaces, you can place the water cube. By clicking the object and dragging the object to translate the water cube's position. However these three 2D spaces are only for displaying and changing the position of the cube, not showing the rendered result of the object; which is black line.

In the 3D space when dragging the mouse will rotate the camera, mouse wheel to scale up or down, and dragging right mouse to translate the camera position. However, since it's not a real time rendering process, you might have to click once to see the result of the changed process. You can press 'X' and 'Z' to set the starting point and ending point of the line in the console by inputting the three position argument. After setting the object and camera position press 'Q' and click the 3D space once and only once to start rendering. It will take about 2-3 minutes to finish rendering and the green line will be the object with the refraction and total reflection. We do not recommend changing the camera position when 'Q' is enabled (press 'Q' once more to disable) because when you try to change the camera rotation, the rendering process will start again and it will take time. The result will look different where the camera position and where the camera is looking at the screen. Additionally, you can change the refraction rate by changing the Sujo's index in the add Object function.

```
def addObject(self, x, y):  
    if self.id != 5:  
        #sujo refraction rate  
        sujo = Sujo(index)
```

Here are some figure examples to show the result.

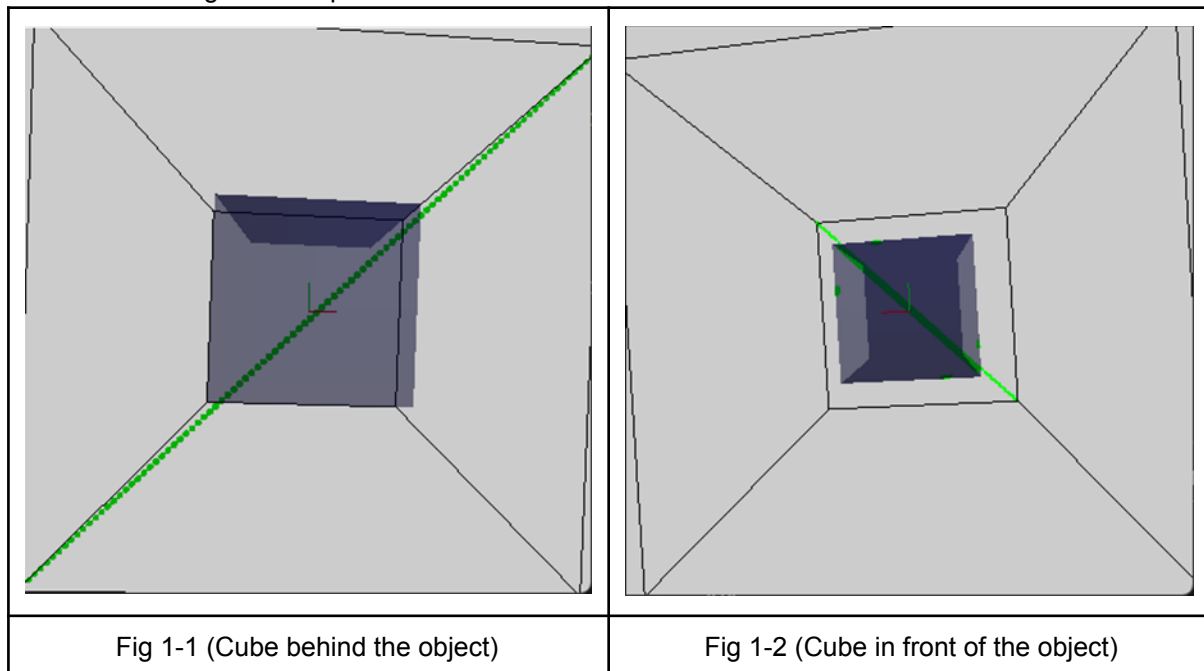


Fig 1-1 and 1-2. First image shows that when the water cube behind the actual object (green line from [0.5, 0.5, 0.5] to [-0.5, -0.5, 0.5]) refraction doesn't happen. Meanwhile in the second image when the water cube is ahead of the actual object, we can see refraction.

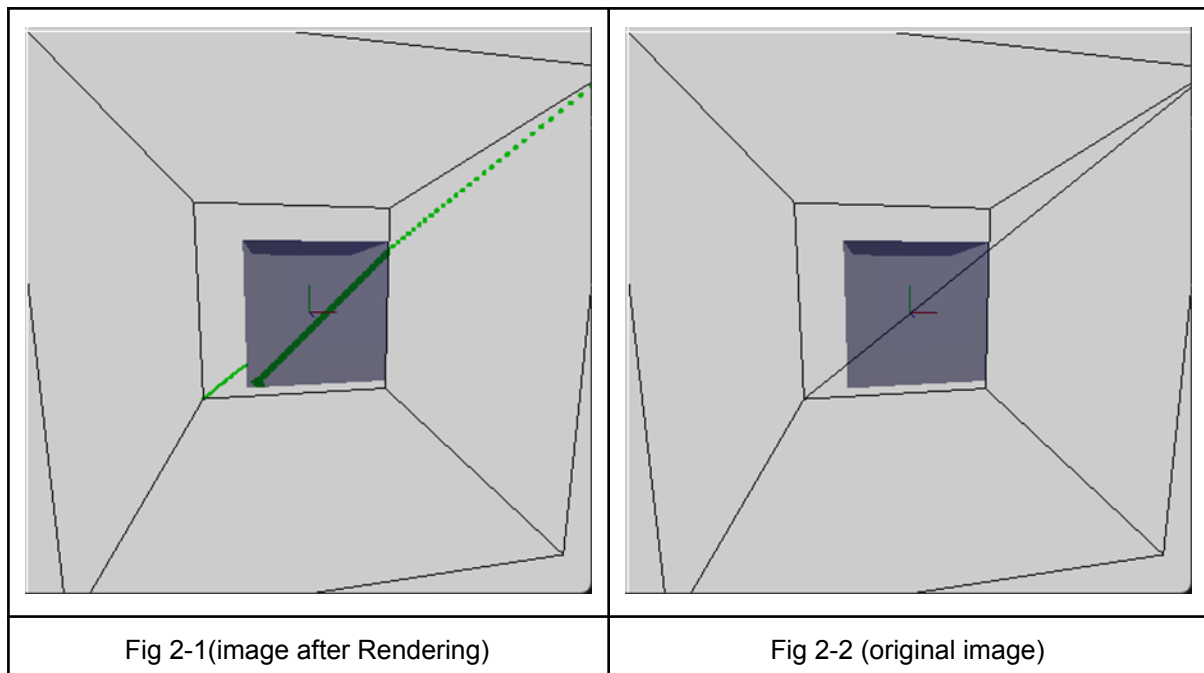
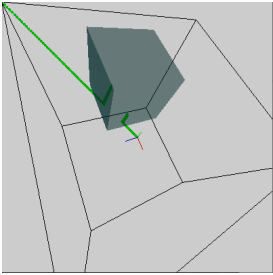
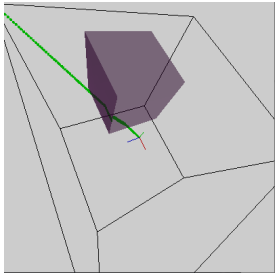
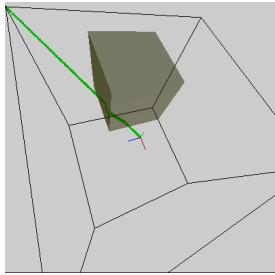
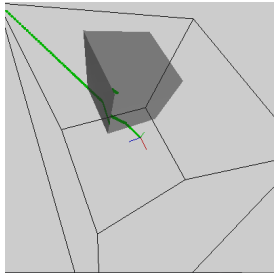
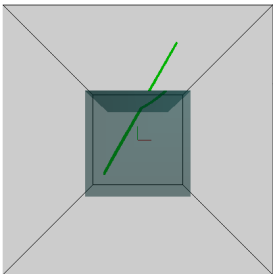
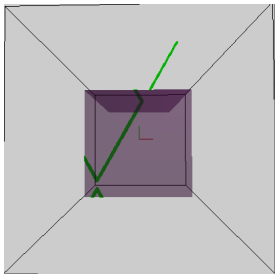
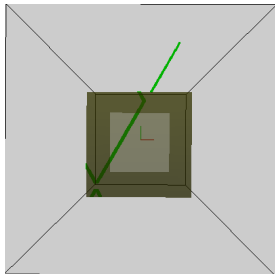
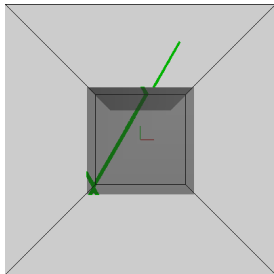
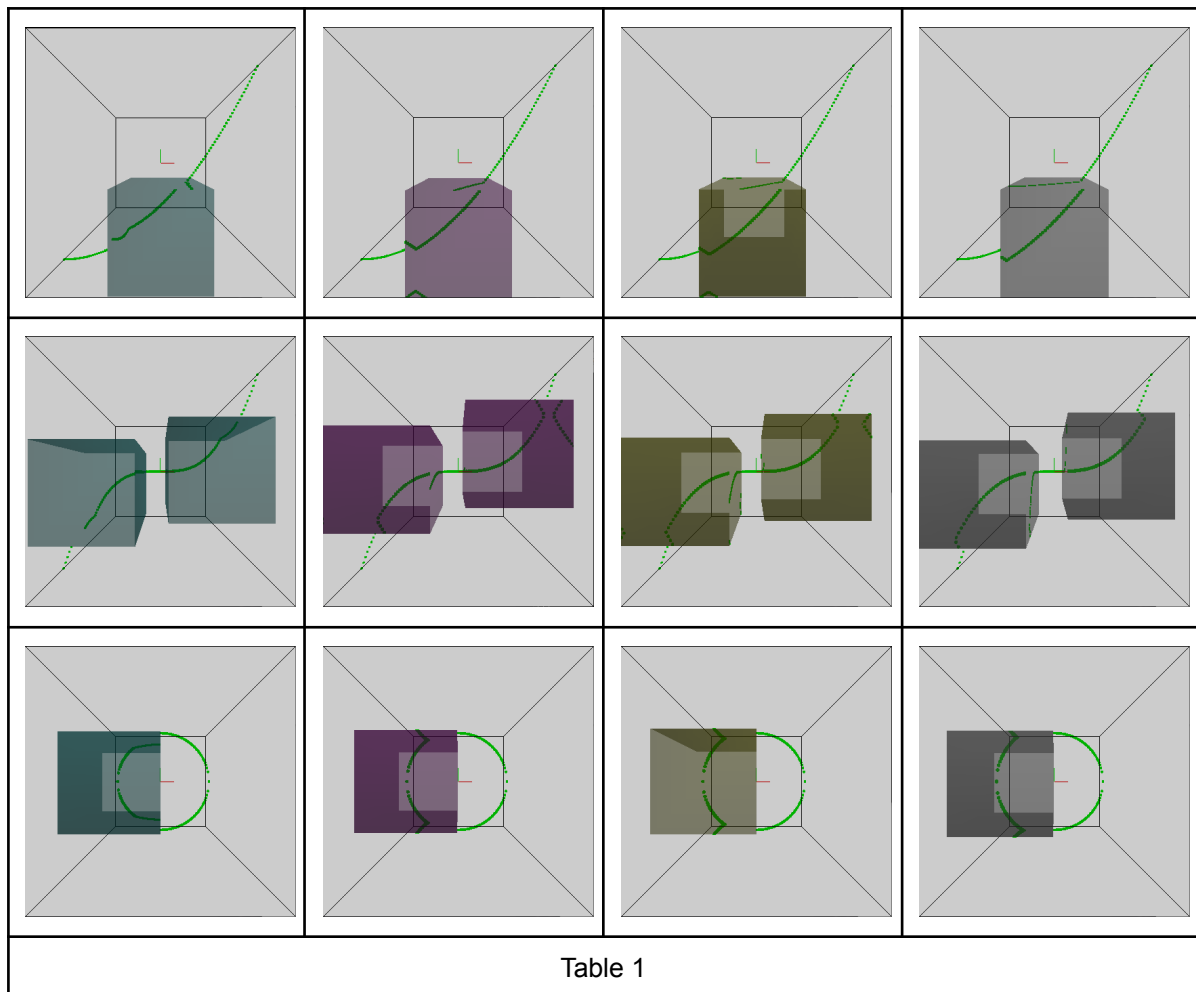


Fig 2-1(3D space after render) and Fig 2-2(3D space before rendering). When an object (from  $[0.5, 0.5, 0.5]$  to  $[-0.5, -0.5, -0.5]$ ) passes through the water cube where refraction rate is 10. When entering the cube, we can see the refraction because the original line vents to the outside. Also where the object ends, we can see the object is back to its original position, Straight line at the outer point is from the bottom square from the cube.

Table 1 is the table of images shown when different indices are used for the same situation.

index = 0.7	index = 1.3	index = 1.5	index = 2.4
			
			



## Discussion

In the course of the project, we failed to solve three major challenges. First, we had difficulty implementing using mesh objects instead of using dots (straw part). An understanding of the structure and characteristics of the mesh was essential to achieve the desired visual effect, but we did not acquire the knowledge necessary to realize it.

Secondly, we failed to effectively reproduce the color change and loss of the object when the refraction phenomenon occurs. The color change or loss that occurs when the light passes through the object is a very important factor visually, but we do not have the necessary technical ability to express it realistically.

Finally, we encountered significant difficulties during the real-time rendering process. This was a major obstacle to reflecting the current state of the simulation in real-time, and did not provide smooth visualization during the user's interaction. Speed issues in the rendering process were a major performance constraint, and we discussed various ways to solve them. For example, when calculations can be performed on non-camera objects, the processing time of 160,000 iterations can be reduced to the number of pixels corresponding to the object repetition length. However, this method has not provided a solution in finding a way to track the light rays from the object, and there is a risk of a significant decrease in accuracy at locations where cube faces overlap.

These issues have been a significant challenge in the progress of the project, and it will be important to address them through future research and development.