CSE167 FA22 Final Project - Ray Tracing

Chuning Liu chl155@ucsd.edu
Nan Jiang n3jiang@ucsd.edu

Topic

Ray Tracing: Ray tracing is a rendering technique that creates highly realistic images by simulating how light behaves in the real world. It involves tracing the path of light as it bounces off of objects in a scene, taking into account factors such as reflection, refraction, and shadow. This allows for the creation of images with more realistic lighting and shadows, as well as more accurate reflections and other optical effects. Ray tracing is becoming increasingly popular in computer graphics and is used in various applications, including movies, video games, and architectural visualization.
In order to do this, a number of mathematical concepts and techniques are used, including vector algebra, matrix operations, and calculus. The physics involved in ray tracing includes the principles of geometric optics, which describe how light behaves as it travels through different materials and media. More precisely, a collection of light will be used to generate the final color for a certain “hit”(which is the intersection of light and the entity).

Components

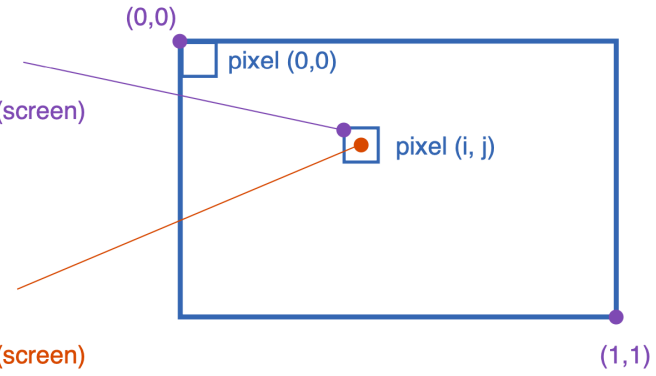
RayThruPixel. This is used to determine the starting point of a ray and its direction.
Since ray tracing requires the position of the pixel on the image, we need to get the corner of pixel(i, j) and center of pixel (i,j) if the screen ranges from (0,0) to (1,0) from top-left to bottom-right.

- The corner of pixel (i, j)

$$\left(\frac{i}{\text{width}}, \frac{j}{\text{height}}\right)$$ (screen)

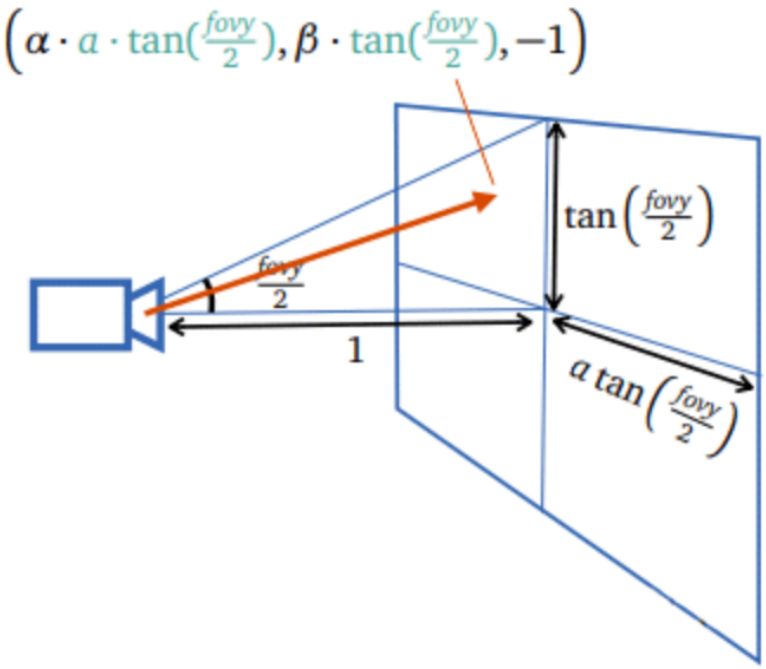
- The center of pixel (i, j)

$$\left(\frac{i+\frac{1}{2}}{\text{width}}, \frac{j+\frac{1}{2}}{\text{height}}\right)$$ (screen)



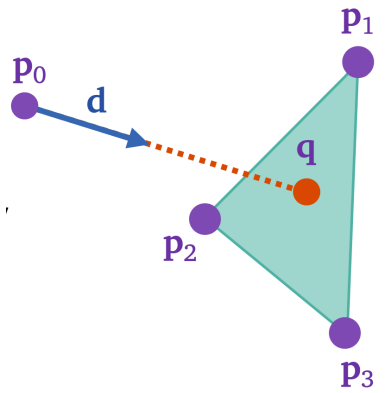
If the screen ranges from (-1,1) to (1,1) from bottom-left to top-right. The center of pixel will be $(2 \cdot \frac{i+1/2}{width} - 1, 1 - 2 \cdot \frac{j+1/2}{height})$. Therefore, we can define $\alpha = 2 \cdot \frac{i+1/2}{width} - 1$, $\beta = 1 - 2 \cdot \frac{j+1/2}{height}$.
For this function, we have four parameter: Camera, i, j, width and height.

IN camera coordinate, we have source of ray which is (0,0,0) and we need to calculate the position that red arrow point to the image, by the trigonometric function, the position of red arrow point to the image is $(\alpha * a * tan(\frac{fovy}{2}), \beta * tan(\frac{fovy}{2}), -1)$, $which\ a = \frac{width}{height}$



In the world, the ray pass the position $(\alpha * a * tan(\frac{fovy}{2}), \beta * tan(\frac{fovy}{2}), -1)$, which indicates that the direction of the ray is the vector
$d = normalize(\alpha * a * tan(\frac{fovy}{2}), \beta * tan(\frac{fovy}{2}), -1)$.

Intersect. This is used to find the intersection of a ray and an entity
When we get that ray from rayThrupixel, there is a chance that it will hit an object placed in the scene. Likewise, there will be light hits between objects and objects. Among them, the nearest hits are the ones that may scatter from other light sources.
So, to find such hits, we need to define an "object" in the ray tracing algorithm. These "objects" can be replaced by triangles. We build a function called buildtriangleSoup. It converts the position and normals of each triangle from model coordinates to camera coordinates and stores them in a set. This way we get the set of triangles for each model. So, when we want to get hits, we just need to iterate through each triangle in the triangle soup. After that we are checking if any ray intersects with it.
Next, we have to find the intersection between the ray and the triangle.
Given ray(p_0, d) and triangle's vertices p_1, p_2, p_3.



So, we can get any point along the ray takes the form $q = p_0 + td$,t means the distance between hit and camera center. we also get any point on the plane spanned by the triangle takes the form: $q = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 p_3$ and $\lambda_1 + \lambda_2 + \lambda_3 = 1$Therefore, bace on those three equation,

$$\begin{bmatrix} | & | & | & | \\ p_1 & p_2 & p_3 & p_4 \\ | & | & | & | \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{bmatrix} = \begin{bmatrix} | \\ p_0 \\ | \\ 1 \end{bmatrix}$$

For all $\lambda_1 + \lambda_2 + \lambda_3$ and t are ≥ 0 . Therefore, we have an intersection

Fornow, we need to use the barycentric coordinate to interpolate position and vertex attributes, such as $q = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 p_3$ $and\, n = normalize(\lambda_1 n_1 + \lambda_2 n_2 + \lambda_3 n_3)$, which n means the normal for three points.

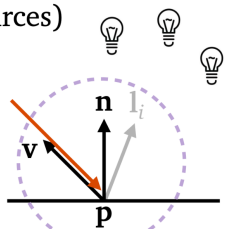
For now, we find part of the object which was hit by light.

FindColor. This is used to find the color given the ray/light and the hit intersection

With step 2, we can find the hit intersection and our goal is to generate the color in this intersection point. Since we have multiple light sources, the color for certain intersections is determined by the intersection's material(i.e ambient, diffuse, specula, and emission) and the aggregation of those colors. The following figure explains the process of generating the corresponding color. We need to aggregate diffuse from other objects(which is done by recursive step by setting recursive depth) and add specular at last. The way to get ambient, diffuse, specular and emission is shown below. The goal is to generate the color and the reflection by other objects and the graph below.

- Direct shading model we did in OpenGL

$$\mathbf{L}_{\text{seen}} = \sum_{i \in \text{lights}} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{\substack{\text{light} \\ \text{source}_i}} \max(\mathbf{n} \cdot \mathbf{l}_i, 0) + \mathbf{C}_{\text{specular}} \text{BlinnPhong}(\mathbf{v}, \mathbf{n}, \text{LightSources})$$



$$\mathbf{R}_{\text{ambient}} = \mathbf{C}_{\text{ambient}} \mathbf{L}$$

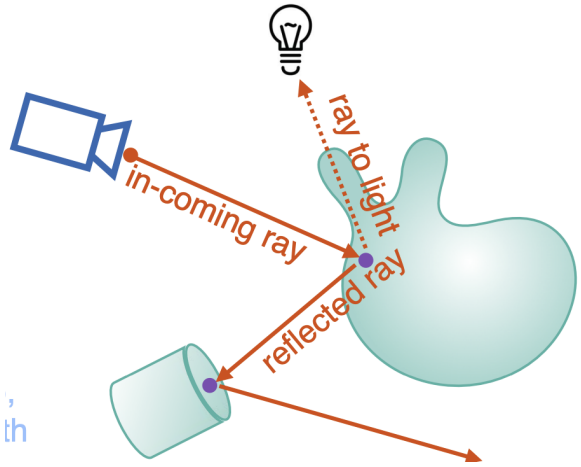
$$\mathbf{R}_{\text{diffuse}} = \mathbf{C}_{\text{diffuse}} \mathbf{L} \max(\mathbf{n} \cdot \mathbf{l}, 0)$$

zero out negative cosines

componentwise multiplication

cosine

$$\mathbf{R}_{\substack{\text{BlinnPhong} \\ \text{specular}}} = \mathbf{C}_{\text{specular}} \mathbf{L} [\max(\mathbf{n} \cdot \mathbf{h}, 0)]^{\sigma}$$



Procedure

1. We determine all positions and colors of light
2. For each pixel in the screen, we output the color for this pixel. To achieve this, we will do the following
    a. We determine the ray passed in this pixel using **RayThruPixel**
    b. We use this ray to find the intersection of this ray and the entity it shot with **Intersect**

    c. With this intersection and the material given, we iterate over all light sources and aggregate all colors reflected. Finally, we add specular for the final output. This is the detail of **FindColor**
    d. If there is no intersection, we just output the color with the default value(0)
3. We get the rendered image

Implementation

Image:
This image.h file Its main function is to store ray-traced colors. In the draw function, buffer object and buffertexture are stored, which mainly allows the image to be rendered in the image.

Triangle:
Triangle.h file is a struct class, it only contains three points, three normals and the material pointer.

RTGeometry:
RTGeometry class is a parent class which contains a triangle vector and two virtual init () functions.

RTScene:
RTScene class is to build triangleSoup and store and transform the triangles to the camera coordinate and add material to it.

RTModel:
RTModel is a struct class to store geometry pointers and material pointers.

RTCube and RTObj:
Those two classes have init() function, which fills the triangle to it.
Intersection:
Intersection contains position of the intersection, surface normal, direction to incoming ray, and triangle pointer point to geometric primitive and a distance to the source of ray.

Ray:
Ray class contains the basepoint and direction.

RayTracer:
RayTracer has four functions to help us render the image.

- **RayThruPixel** is to calculate the ray under camera coordinate that we talk about above.
- **Intersect with parameter triangle:** Find the center of barycentric coordinate lambda and then calculate the distance between the ray source and the intersection point. Note that the sum of lambda and t should be greater than zero according to the description of intersection we talk about above.

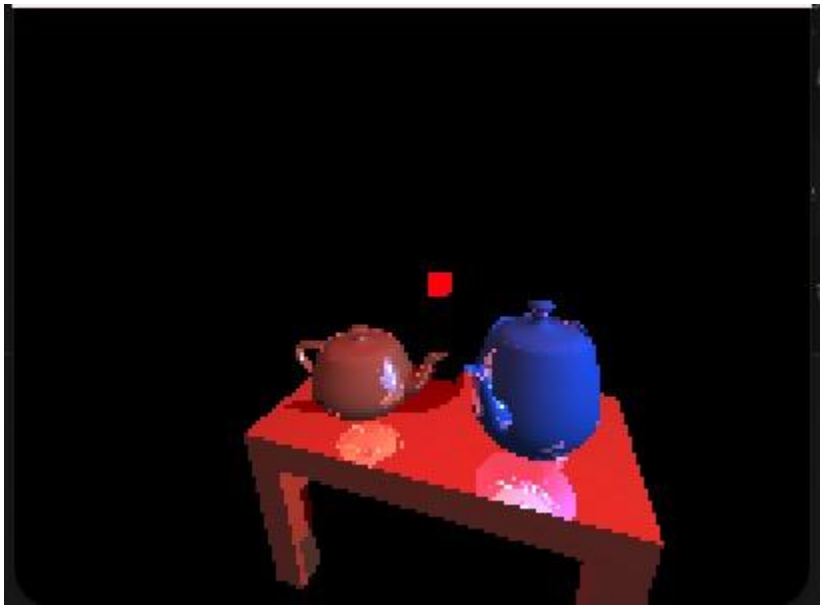
- **Intersection with parameter scene:**It mainly traverses the triangle soup and then finds the intersection closest to the ray.
- **FindColor:** Its main purpose is to find the color of a given intersection by calculation. This is mainly achieved by recursive shading models and specular reflection.

Summary

In this project, we implement a very simple ray tracing rendering with OpenGL. This report includes basic math/physics related to this topic. Also, the implementation detail is included.

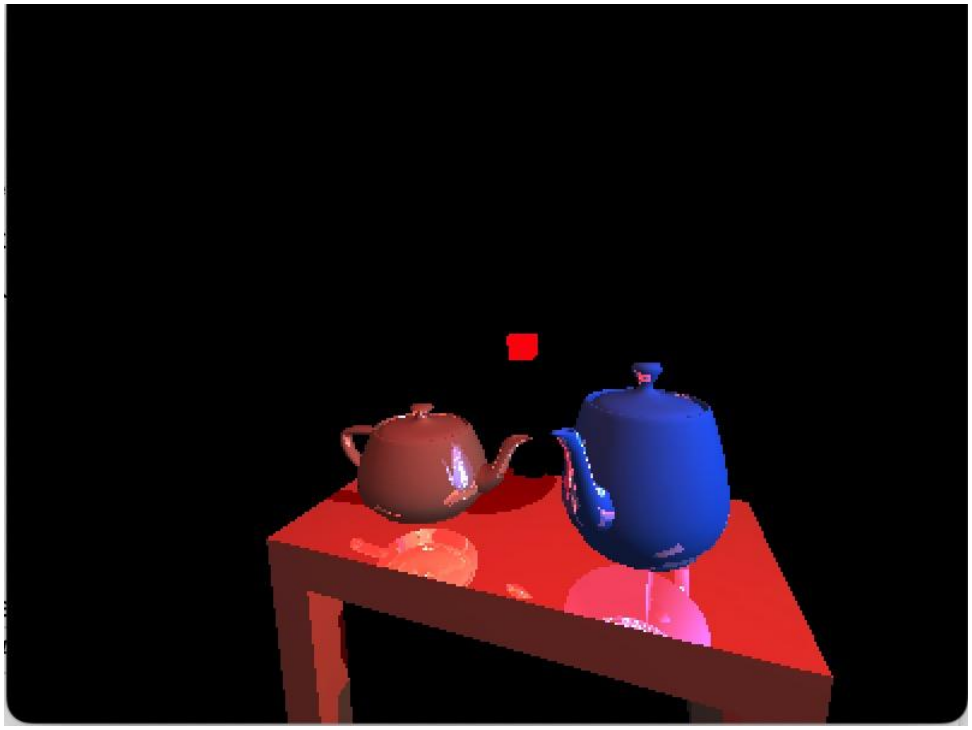
Demonstration

The scene we decided to render is from the HW3 object, a table with two teapots.
Here is when the image is 200 X 150, The recursion death is 3.
We zoom the camera with 0.19f, rotateRight the camera with 10.0f and rotateUp the camera with 5.0f.



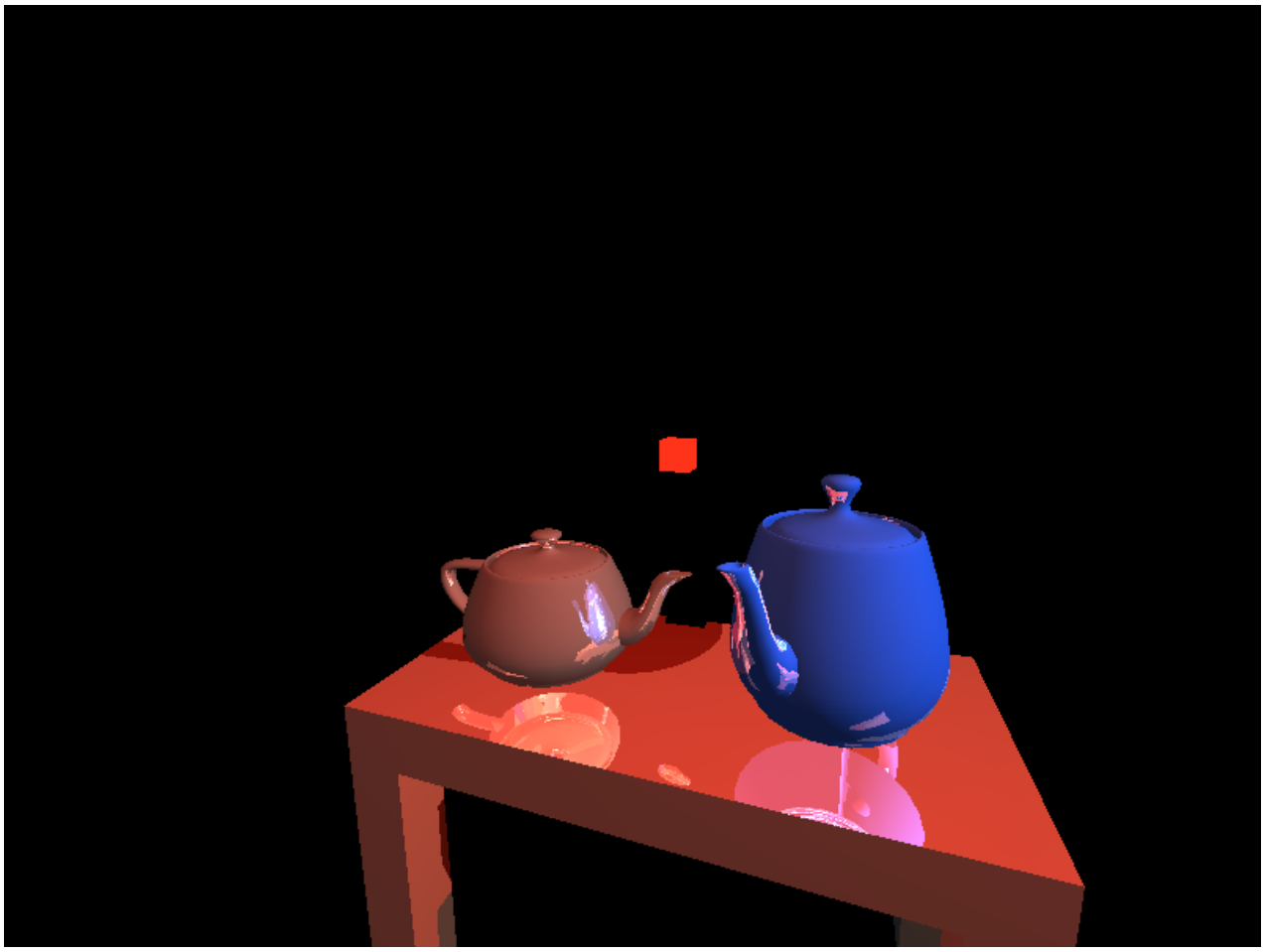
As we can see, the brown teapot already has shades of the blue teapot on it. And we can see that there is already a clear reflection on the table. However, the image is too small, resulting in a relatively blurry picture.

Here is when the image is 400 X 300, The recursion death is 60.
We zoom the camera with 0.19f, rotateRight the camera with 10.0f and rotateUp the camera with 5.0f.



We can see that by enlarging the size of the image, the rendered image becomes clearer. The shadows and specular reflections also become clearer.

And here is the Image with 800 X 600 with the same recursion death.



We see that the final original size image has been rendered to a much sharper image.

Acknowledgements:

Office Hour with Peter Wu
Office Hour with Professor Albert Chern

Reference:

- Lecture (Light) and (Ray Tracing) ppt
- Ray Tracing Write up.
- Piazza Post