Ray Tracing

CS 452 - Computer Graphics
Assignment 4

We start our exploration of “Rendering"- the process of converting a high-level object-based description of scene into an image. We will do this by building a raytracer - a program that converts 3D geometry into 2D pixels by tracing light rays backwards through a scene. Raytracing simulates light rays moving through a scene, and can produce physically accurate representations, as explored in detail in Lectures.

Required Reading

Please read Chapters 2 (Miscellaneous Math), 9 (Surface shading), 10 (Ray Tracing), of Shirley Fundamentals of Computer Graphics", 2nd ed., for this assignment.

Skeleton Code

Download the skeleton code provided with the assignment from the LMS. The skeleton code project folder also contains 7 scene files namely Scene1.scd, Scene2.scd, Scene 3.scd, Scene4.scd, Scene5.scd, Scene4r.scd and Scene5r.scd that you have to render.

Results

The desired results are also uploaded on the LMS along with this assignment. These results will help you track your progress in this assignment.

Submission

The assignment has been broken down into two parts. The details and division of both parts of the assignment is given in this handout. The first part is due on **2nd of November**, whereas the second part will be due approx. **one week** after (exact date to be announced later).

1. Tracing rays

Raytracing models the rendering process as shown in this illustration:

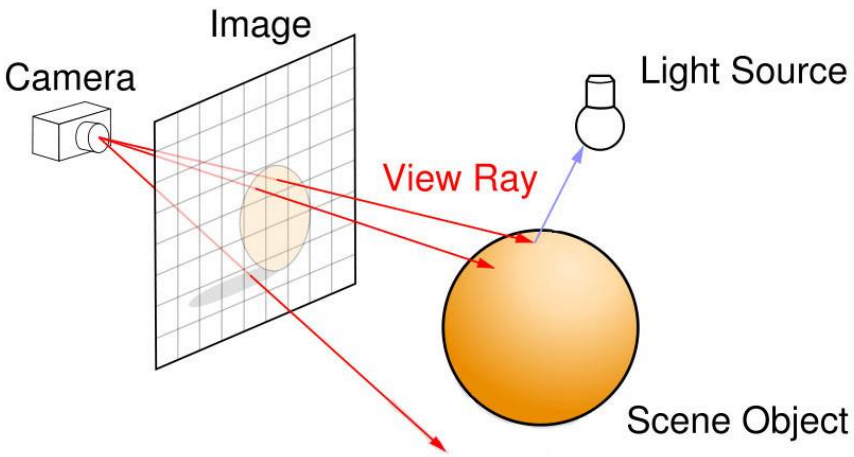

Figure 2: Raytracing

1a. Overall Design

To do this, our Raytracer will follow the design shown in the flow diagram given below,

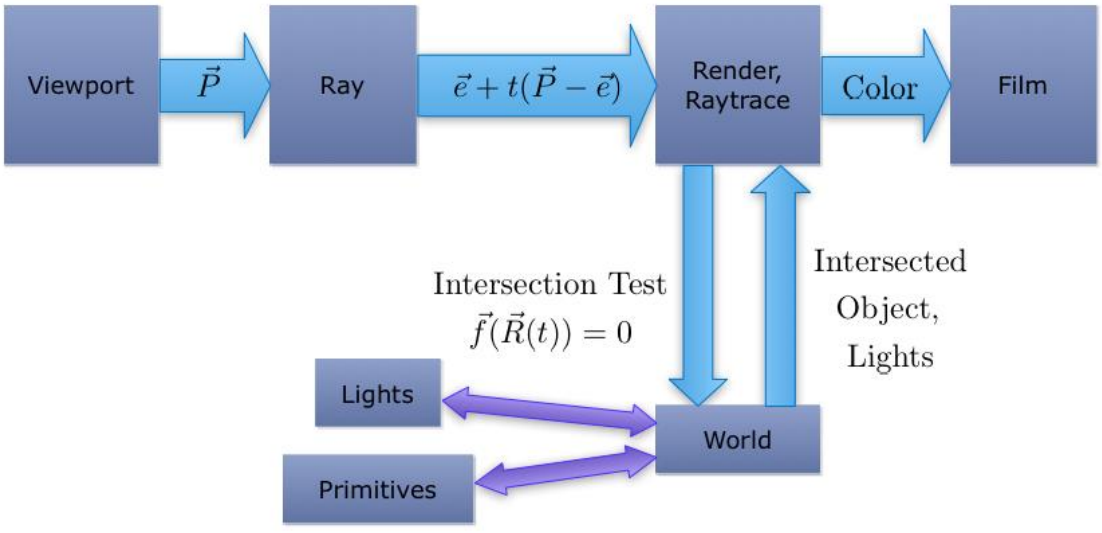

Figure 3: Suggested design

1b.Implementation

Before beginning implementation, it is important to note that classes for **Color, Sampled Point, Ray, Material** are provided in algebra3.h. Please make sure you have read these classes, before moving on.

Next, move step by step through the following guideline and wherever possible, test your code before moving on to the next step to ensure correctness and later help in debugging,

ASSIGNMENT PART ONE

i. Sampling Viewport

This step requires you to construct a viewing ray, as shown in Figure 2.Implementation for this step is to be carried out in **Viewport.cpp**

Our viewport is a rectangle defined by its 4 coordinates in world space, LL, LR, UL, UR .
Write the function **GetSample()** to update the pixel position(x,y) such that the next call to **getSample()** will give world coordinates of the next pixel position by incrementing x and y. This increment will be a value of 1 with aliasing and less than 1 with anti-aliasing.

Anti-aliasing means shooting multiple rays per pixel. Shooting only a single ray per pixel can result in objectionable aliasing effects; these can be reduced by shooting multiple rays per pixels and averaging the returned (r,g,b) intensities. Write your getSample() function in Viewport class to allow shooting multiple rays per pixel. Each pixel should be sampled according to a regular grid (i.e. 2, 3, or 4 samples per pixel edge), the size set by the raysPerPixel parameter in the Viewport class.

Next, write **CreateViewingRay()** to construct the ray from the camera to the image. Generate the point in 3d world space from sample coordinates by using bilinear interpolation across the viewport. This is done by varying u and v from 0 to 1 in appropriately sized steps for each pixel or sample to find the specific point on the rectangle in world space by using the equation,

$$\vec{P}(u,v) = (1-u)\left[(1-v)\mathbf{L}\vec{\mathbf{L}} + (v)\mathbf{U}\vec{\mathbf{L}}\right] + (u)\left[(1-v)\mathbf{L}\vec{\mathbf{R}} + (v)\mathbf{U}\vec{\mathbf{R}}\right]$$

Where u = x/width and v= y/Height, and x and y are sample coordinates.

As a ray can be completely defined by its starting position, e, and direction, d, both of which are vectors, construct the viewing ray by using the equation,

$$\vec{r}(t) = \vec{e} + t\vec{d} = \vec{e} + t(\vec{p} - \vec{e})$$

ii. Save and Display

After this, write the **expose()** and **bakeandSave()** functions in **Film.cpp** to allow the Film aggregator to write the color at each pixel at the end of the raytracing cycle and display the image using FreeImage library functions.

At this point, you may test your Viewport and Film functions by commenting the line c+=traceray(ray,0) out in your renderwithRayCasting() function in main.cpp. Set up a dummy value for the color, c, and you should be able to see a window of that color when you execute your code.

iii. Intersection Tests

Next, we want to find the intersection between the primitives and the viewing ray. Implementation of this step is to be done in **primitives.cpp.** Given a sphere of radius r at position c and a ray from position e in direction d, the sphere can be represented as an implicit surface of the form f(p(t)) = 0. To find the parameter t value at which the intersection occurs means solving for that equation. From Shirley page 77, we find the following formula for t, where d and e describe the ray and c and R describe the sphere,

$$t = \left(\frac{1}{\vec{d}\cdot\vec{d}}\right)\left[(-\vec{d})\cdot(\vec{e}-\vec{c}) \pm \sqrt{(\vec{d}\cdot(\vec{e}-\vec{c}))^2 - (\vec{d}\cdot\vec{d})((\vec{e}-\vec{c})\cdot(\vec{e}-\vec{c}) - R^2)}\right]$$

You can implement this directly using operators on our matrix library's objects, but you want to check for the value of the discriminant (the contents of the square root) before you complete the calculation. If it is negative, the square root will be imaginary, and no intersection occurred. Otherwise, find the value of t for the closest point of intersection.

Note: Sphere centre is the translation applied on a unit sphere at origin. This transformation is stored within the transformation matrix in the sphere class. Remember that in homogenous coordinates, transformation is stored as the 4th column of the transformation matrix.

Next, calculate sphere's normal at the point of intersection. A sphere's outward normal, given its center and a point on its surface, is trivial to find. The normal is the vector from the center to the point on the surface, normalized.

$$\hat{\mathbf{n}} = \text{Normalize}(\vec{p}-\vec{c}) = (\vec{p}-\vec{c})/R$$

After this, write **intersect()** in **World.cpp.** This function calls the intersect function on all the list of primitives intersecting with the ray iteratively to find the closest non-negative value of the point of intersection of the ray with the primitive.

*At this point of Assignment, you may test your code by assigning a white pixel value if a ray hits the primitive and black otherwise, to obtain a binary image of the scene and check the output of your intersect functions in **primitives.cpp** and **world.cpp**.*

ASSIGNMENT PART TWO

iv. Lights

In **Lights.cpp**, write functions for,

Directional Lights – Directional Lights are light sources 'infinitely' far away with light rays moving in a specific direction. Model lights as emitting a color c = (r,g,b) with light rays moving in a direction d = (x,y,z), allowing you to calculate the angles needed for shading.

Point Light – Point lights are emitted from a specific position, p. Model lights as emitting color c=(r,g,b) with light rays moving in all directions from point p=(x,y,z) allowing you to calculate the angles needed for shading.

Point light is affected by Fall-off distance, which means that we want to model lights as dimming with distance, and be able to adjust the factor with which it occurs. This factor is already included in the specification of each light and is read into the light class. When calculating the color of Light, apply this falloff according to the absolute distance from the light to the current location. Assume that all the light intensity values are measured at a distance of one unit from the light. This should make it easy to apply falloff by measuring the Pythagorean distance $x^2 + y^2 + z^2 = (dist)^2$ and scaling the light's intensity appropriately.

To calculate the color of a point light, use the scaling factor of $\frac{1}{(dist+deaddistance)^{falloff}}$ and scale the illumination of the light by this amount.

2. Shading

Now that the basic components have been implemented, the next phase is to implement shading models to calculate the color at each pixel.

All shading models are to be implemented in **Main.cpp**

Given the formulation of Phong Model below, implement ray tracing using the pseudo code given in the slides of **Lecture 14** of class. For reflection and refraction terminate the recursive nature of ray tracing after some specified depth. Set the default depth to be 3.

For this assignment, you will use a slightly different formulation of the Phong reflectance Model to calculate the color of a pixel. Our formula for the Phong Reflectance Model, where ρ is the (r, g, b) intensity of light sent towards the eye/camera, is

$$\rho = m_a CA + \sum_{Lights} m_l CI \max(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}, 0) + m_s SI \max(-\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}, 0)^{m_{sp}}$$

Note: This formulation separates the "color" of the object from the components in the Phong formulation. The equivalent parameters in the lecture slides are:

$$\begin{aligned} \boldsymbol{k_A} &= m_a \boldsymbol{C} \\ \boldsymbol{k_L} &= m_l \boldsymbol{C} \\ \boldsymbol{k_S} &= m_s \boldsymbol{S} \\ \sigma &= m_{sp} \end{aligned}$$

$\boldsymbol{I} = (r, g, b)$ is the intensity of the infalling light.
$\boldsymbol{A} = (r, g, b)$ is the intensity of the ambient lightsource in the scene.
$\boldsymbol{C} = (r, g, b)$ is the "color" of the object.
$\boldsymbol{S}$ is the specular highlight color
$\hat{\mathbf{l}}$ is the incidence vector from the intersection point to the light.
$\hat{\mathbf{n}}$ is the surface normal vector.
$\hat{\mathbf{u}}$ is the vector along the (backwards) viewing ray, from the viewer to the intersection point.
$\hat{\mathbf{r}}$ is the reflectance vector, supplying the angle of reflected light.
m_a, m_l and m_s are the ambient, lambertian and specular surface reflection properties of the material.
m_{sm} is the metalness of the material.
Colors multiply component-wise, thus $\boldsymbol{CI} = \{\boldsymbol{C}_r \boldsymbol{I}_r, \boldsymbol{C}_g \boldsymbol{I}_g, \boldsymbol{C}_b \boldsymbol{I}_b\}$
Notice that all the vectors in this equation are **unit length vectors**. You need to normalize your vectors to ensure this is true.

The material properties are:

- **Ambient reflectance** - m_a is always visible, regardless of lights in a scene. It multiplies directly with the color c to calculate the ambient color of an object
- **Lambertian reflectance** - m_l is matte reflection directly related to light falling onto the object from a light source.
- **Specular term** - term m_s is the mirror-like reflection of light off an object to the eye.
- **Reflection term** - m_r is the mirror property of the material, which attenuates the bounce ray.
- **Metalness** - m_{sm} controls the color of the specular highlights. $m_{sm} = 0$ means the highlight is the color of the lightsource, $m_{sm} = 1$ means the highlight is the color of the object.
- **Specular Exponent (or Phong exponent)** - m_{sp} characterizes the smoothness (i.e., the sharpness of the highlight spot) of a material, and forms an exponent in the calculation of the specular term.

2.1 Ambient Lighting/Shading

Light reflects around a room, illuminating objects uniformly from all sides. This ambient light mixes diffusely, component by component, with the inherent color of the object.

2.2 Lambertian Lighting/Shading

We assume that surfaces are **Lambertian**, thus they obey *Lambert's Cosine law,*

➔ Absorbed and re-emitted light energy $\mathbf{c} \propto \cos(\theta_{\text{incidence}})$. In other words $\boldsymbol{c} \propto \hat{\mathbf{n}} \cdot \hat{\mathbf{l}}$

This states that the color of a point on a surface is independent of the viewer, and depends only on the angle between the surface normal and the incidence vector (the direction from which light falls on the point). We

want the actual color to depend on both the color of the light source $\boldsymbol{I} = (r, g, b)$ and the material's color and lambertian reflectance, specified by $m_I\boldsymbol{C}$:

$$\rho_{\text{lambert}} = m_I\boldsymbol{C}\boldsymbol{I}\max(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}, 0)$$

Where ρ_{lambert} is an (r,g,b) intensity value.

$\hat{\mathbf{l}}$ is the vector defining the light's direction(pointing at the light)

$\hat{\mathbf{n}}$ needs to be calculated for the surface itself. This is easy for spheres as the normal vector points away from the centre.

2.3 Specular Lighting/Shading

The Phong illumination model states that there may be a bright highlight caused by the light source on the surface. This effect depends on where the viewer is. The effect is the strongest when the viewer vector and reflectance vector are parallel.

$$\rho_{\text{specular}} = m_S\boldsymbol{S}\boldsymbol{I}\max(-\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}, 0)^{m_{sp}}$$

The color, $\boldsymbol{S}$, of this highlight is calculated by linearly interpolating between $\boldsymbol{C}$ and (1, 1, 1) according to m_{sm}, the **metalness**. A metalness of 1 means that the specular component takes the color of the object, and a metalness of 0 means that the specular component takes the color of the infalling light.

m_{sp} is the **smoothness** of the material - it affects how small and concentrated the specular highlight is.

$\hat{\mathbf{u}}$ is calculated along the backwards viewing ray from the viewere to the surface point

$\hat{\mathbf{r}}$, the reflectance vector, is calculated using $\hat{\mathbf{l}}$ and $\hat{\mathbf{n}}$

$$\hat{\mathbf{r}} = -\hat{\mathbf{l}} + 2(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$

3.Design Ideas

The following documentation will help you in this project:

- Chapters 3 and 14 in Shirley's textbook (+ 2 and 6 for math background).
- Sid's "How to raytrace" guide http://fuzzyphoton.tripod.com/howtowrt.htm. See the Links and Reference pages on this site as well.
- Raytracer Design http://inst.eecs.berkeley.edu/~cs184/sp09/resources/raytracing.htm
- Intersection Tests http://www.realtimerendering.com/intersections.html

Submission Details

- DUE: Mon November 2nd, 11:55 p.m (Part 1)
- Submission will be done on LMS.
- You must submit only source files that you have modified namely:
    - Film.cpp
    - Lights.cpp
    - main.cpp
    - Primitives.cpp
    - Viewport.cpp
    - World.cpp
- Submit snapshots of your intersection result (for part 1). See Results folder for comparison.
- Submit snapshots for each of the 7 provided scenes and any other additional scene that you may have generated (for part 2)
- Late submission policy: 10% deduction for each day after the due date exempting the first 1 hour after the due date and time.

Bonus

- The scene files are very simple text files and can be changed easily to generate a different scene. Modify the scene file and submit results on your own scene (10%)
- Implement ray – triangle intersection (15%)

Acknowledgements

The inspiration for the structure of this assignment, and skeleton source code, came from Siddhartha Chaudhuri framework for Stanford's CS148 Intro to Graphics and Imaging course which is inspired from Niels Joubert and James Andrews' framework for Berkeley's CS184 Intro to Graphics course.

References

Stanford CS148, Siddhartha Chaudhuri, http://graphics.stanford.edu/courses/cs148-10-summer/as3/as3.html
Berkley CS184, Niels Joubert and James Andrews, http://inst.eecs.berkeley.edu/~cs184/sp09/assignments/AS5.html