Project 3A Guide

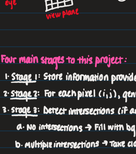
Ray Tracing: what we're doing at a high level

(Feat: Shachi's terrible drawing skills)

lights

1. For each pixel, shoot a ray that goes through the pixel on the view plane
2. Check to see if that ray hits any of the objects in the scene (i.e. iterate through list of spheres)
3. If the ray doesn't hit anything, color it with the background color

Else, use diffuse shading equation to color the pixel based on the material properties of the closest hit object (and lights in the scene)



Four main stages to this project:

1. Stage 1: Store information provided in .CLI files → interpreter function
2. Stage 2: For each pixel (i, j), generate an eye ray → render_scene() function
3. Stage 3: Detect intersections (if any) between the eye ray and all of the spheres in the scene → Still in the double for-loop
   a. No intersections → Fill with bg color
   b. Multiple intersections → Take closest hit sphere
   c. Tip: To check if you're detecting intersections correctly, fill the pixel with a dummy color if you successfully hit something
4. Stage 4: Implement diffuse shading
   ↳ If intersection was detected, color the pixel based on material properties of closest hit sphere

Stage 1: Store information from .CLI files

↳ Global variables, data structures, and classes (recommended) !!

- Lists: Lights, Spheres, and UVW coordinate frame vectors
- Global variables: background color, FOV, current material, eye position
- Classes: HIGHLY RECOMMENDED!
  - Light:
    - Attributes: position (x,y,z), color (r,g,b)
  - Sphere:
    - Attributes: center (x,y,z), radius, material
  - Material: ← Your "current material" global variable would be an instance of a "Material" object
    - Attributes: Diffuse r,g,b; Ambient r,g,b; Specular r,g,b; Specular power, n; refl

(global variable)

↳ "Surface" is parsed before "SPHERE", so you would store the current material within your sphere constructor

Note: The interpreter() function already parses the file for you! You just have to store the information in the respective if-else blocks

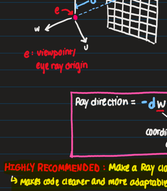
Stage 2: Generate/calculate eye rays

↳ For each pixel, we're creating an eye ray that originates from the eye position and passes through the view plane → i.e. within the double for-loop in render_scene()

- Recall, a ray has
  - origin → eye position for 3A
  - direction → Based on coordinate frame vectors U, V, W

How to calculate eye ray direction:

- The eye is a focal distance d away from the view plane



d: focal length (distance from e to view plane) $= \frac{1}{\tan(FOV/2)}$

$U = \frac{2i}{width} - 1$

$V = \frac{2j}{height} - 1$

screen coordinates (i.e. i and j)

Make sure to normalize after sum!

$$\text{Ray direction} = -d\,w + V\,v + U\,u$$

coordinate frame vectors given to you

HIGHLY RECOMMENDED: Make a Ray class that stores origin and direction

↳ Makes code cleaner and more adaptable for 3B!

Stage 3: Detecting intersections between eye ray and scene objects (MOST IMP. STAGE!!)

↳ Still within that double for loop in render_scene(), after you've created an Eye ray, see if it intersects with any of the spheres in your scene

- Recall that we want to take the closest hit sphere. → How do we do this?
  ↳ Maintain a $minT$ variable. This will store the smallest, valid root (t-value) out of all the spheres and thus correspond to the closest hit sphere

Overall Process

1. Initialize $minT$ to a big number
2. Loop through your list of spheres → For each sphere, do the following:
   a) Check if the ray intersects with the sphere by plugging ray eq. into sphere eq. and solving for t → More on this below
   b) If the t-value is valid (positive/non-imaginary) AND it's less than $minT$
      i) Store information about that sphere + intersection pt with ray (plug t into ray equation), material, normal vector (intersection pt. − center of sphere)
      ii) Set $minT$ to the sphere's t-value
3. Color the pixel:
   - No intersections → bg color
   - ≥1 intersections → Diffuse shading → Just to see if you passed Stage 3. Fill the pixel with black to make sure you're getting the right "outlines" of the spheres in the scene

Detecting Intersections Between a Ray and a Sphere

↳ Recall from lecture, the equation for a ray and implicit equation for a sphere:

- Ray equation: $R = \underbrace{O}_{\text{origin}} + t\underbrace{d}_{\text{direction}}$

  ↳ By components: $x(t) = O_x + t \cdot d_x$, $y(t) = O_y + t \cdot d_y$, $z(t) = O_z + t \cdot d_z$

- Implicit Sphere Equation: $\langle C_x, C_y, C_z \rangle$: center of sphere

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - r^2 = 0$$

(center, radius)

- Key Idea: Plug ray equation into sphere equation, solve for t
  If t-values are valid, compare to $minT$

① Plugging Ray Eq. into Sphere Eq.: Replacing x, y, z in the implicit equation with respective ray components

$$(d_x t + O_x - C_x)^2 + (d_y t + O_y - C_y)^2 + (d_z t + O_z - C_z)^2 - r^2 = 0$$

(x, y, z)

(O_x, O_y, O_z) → origin of ray

(d_x, d_y, d_z) → direction of ray

Let $U_x = O_x - C_x$; $U_y = O_y - C_y$; $U_z = O_z - C_z$

$$(d_x t + U_x)^2 + (d_y t + U_y)^2 + (d_z t + U_z)^2 - r^2 = 0$$

$$= (d_x t)^2 + 2 d_x U_x t + U_x^2) + (d_y t)^2 + 2 d_y U_y t + U_y^2) + (d_z t)^2 + 2 d_z U_z t + U_z^2) - r^2 = 0$$

$$= \underbrace{(d_x^2 + d_y^2 + d_z^2)}_{a} t^2 + \underbrace{(2 d_x U_x + 2 d_y U_y + 2 d_z U_z)}_{b} t + \underbrace{(U_x^2 + U_y^2 + U_z^2 - r^2)}_{c} = 0$$

② Solve for t using quadratic equation:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Make sure $b^2 - 4ac$ is not negative before proceeding, otherwise, you'll end up with imaginary values!

a) If the discriminant is negative, ray did not hit sphere

b) Else, calculate roots (t-values)

↳ Two roots? → Take the smallest positive out of the two

* Root value MUST be positive!

Recommended Approach for Intersection Logic: This will make your code more organized and adaptable for part B!

- Maintain one function to loop through all of your shapes, and then a function within your sphere class to detect ray intersection with a single sphere

```
renderScene()
  for (i,j)
    create Eye ray
    rayIntersectScene(Eye ray)
    ...
```

rayIntersectScene(Eye ray) → Insert "closest hit" sphere logic here ($minT$, closest hit sphere)

```
  for each Sphere in your list
    call intersect(Eye ray) on Sphere
    ...
```

Sphere class:

```
  ...
  intersect(ray)
```

→ Plug ray equation into implicit sphere eq. + calculate t

2. Create a "Hit" class to store information about closest hit sphere
   ↳ Attributes: Sphere itself, normal vector, t-value, intersection point
   - intersection point: Plug calculated t-value back into ray equation
   - normal vector: intersection point − center of sphere, normalized

* You can pass this hit object as a parameter in your diffuse shading function (more on that below)

Stage 4: Diffuse Shading

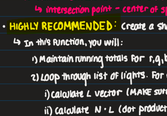
- Recall that if you've successfully hit something, that you're taking the closest hit sphere
- If your ray hits nothing, you fill that pixel in with the background color
- For part 3A, you're just implementing the diffuse shading equation:

$$C = \sum_{\text{all lights}} (\text{hit object's diffuse material}) * (\text{light color}) * \frac{\max(0, N \cdot L)}{\text{diffuse coefficient}}$$

(sum across all lights; current light)

L vector = light position − hit point, normalized

Ex:



N vector: surface normal of the closest hit sphere
↳ intersection point − center of sphere

- HIGHLY RECOMMENDED: Create a shade function that takes in a Hit object, call in render_scene()
  ↳ In this function, you will:
  1) Maintain running totals for r,g,b color components
  2) Loop through list of lights. For each light, do the following:
     i) Calculate L vector (MAKE SURE TO NORMALIZE!!)
     ii) Calculate $N \cdot L$ (dot product)
     iii) Add the following to each color component's running total:
     running total += (hit object's material) * (light color) * max(0, N·L)
  3) Return the total color for each color component → Fill the pixel with the returned color

A Few Additional Recommendations:

1) Use helper functions!
   ↳ If you find yourself copying and pasting segments of code, make helper functions!
2) Create classes as outlined above!
   ↳ Trust us, this will make your code more organized (and easier to debug)!!
3) Inverting the y-coordinate → Important!!
   ↳ Recall that Processing's coordinate system is flipped, so to make sure to account for this

When calculating eye ray direction → multiply the "V" scalar by −1