 # CSE167

Done by: Jiayi Guan, Yuheng Ge

Code based on HW3-CSE167-FA22.

Ray tracing:

Ray tracing is about tracing the path of light from the view camera , through the 2D viewing plane, out into the 3D scene, and back to the light sources. Imagine there is an object in front of you that is illuminated by some rays, ray tracing is just about following these rays backwards to the objects that interacts with the lights.

Workflow:

When a scene is built by Hierarchical Modeling, for each light, we use triangles to replace buffers to make the storage more efficient. We create a ray that has a hit on the scene, we get that intersection by calling the helper function which returns a ray-triangle intersection. If there is a hit, we follow the concepts of ray-triangle intersection and uses triangle's positions and base point of the ray along with its direction to calculate the three lambdas and one t, if the four values are all greater or equal to zero, we know there is a hit, and t is the distance. We manipulate vectors and matrices to transform everything into world coordinate. FindColor runs like the lighting.frag for hw3, after getting all the vectors correct, just simply applying the formula of Phong model. We have this recurrence for FindColor to make the image more realistic by doing the reflected lights, using specular to add onto color and when we reach 0 recursion depth, we add the specular part to color which is our final result. The color is stored in each pixel, which is initialize when creating Image object.

Image:

We created this Image class to store the RGB colors for each pixel in arrays of pixels. Calling Raytraced will assign pixel values to the image. This class includes properties of an Image object, which is width, height, and the array to store colors; default constructor and another constructor that takes in the width and height, initializes corresponding variables, and the pixels array needs to contain colors for all pixels, so the size of it is width * height. To show an image on screen, we store it as a texture and transfer it to the frame buffer. Then we created the draw function to display the image.

Triangle:

We created the triangle class to store the position and color of each pixels. The reason why we used triangle instead of buffers is that triangles are more memory efficient and also can be sorted. In this class, each triangle contains three vertices, three normal vectors and material pointer which represents the material of the area. One more reason why we used triangles instead of buffers is that triangle is the smallest two dimensional primitives, so that triangles are easier to render.

Ray:

A ray is described by a point and a direction. Ray class contains two variables, basepoint and a direction. As basepoint is where the ray is shotting out from in dir direction.

Intersection:

Intersection class contains the three variables, "P" the position of the intersection, "N" the surface normal vector, "V" the direction to the incoming ray, "triangle" that contains the material information and "dist" which is the distance to the source of ray . The class basically represents the intersection point of a ray with a triangle or a scene. We will be finding the color of each pixel with intersect and lights.

RTGeometry:

This class was modified from geometry class that we replaced the buffers with triangles to make the memory more efficient and make it easier to render.

RTCube:

In this class, we assign/initialize the 3 positions and 3 normals for each triangle with given datas, and insert each triangle into elements defined in RTGeometry.

RTObj:

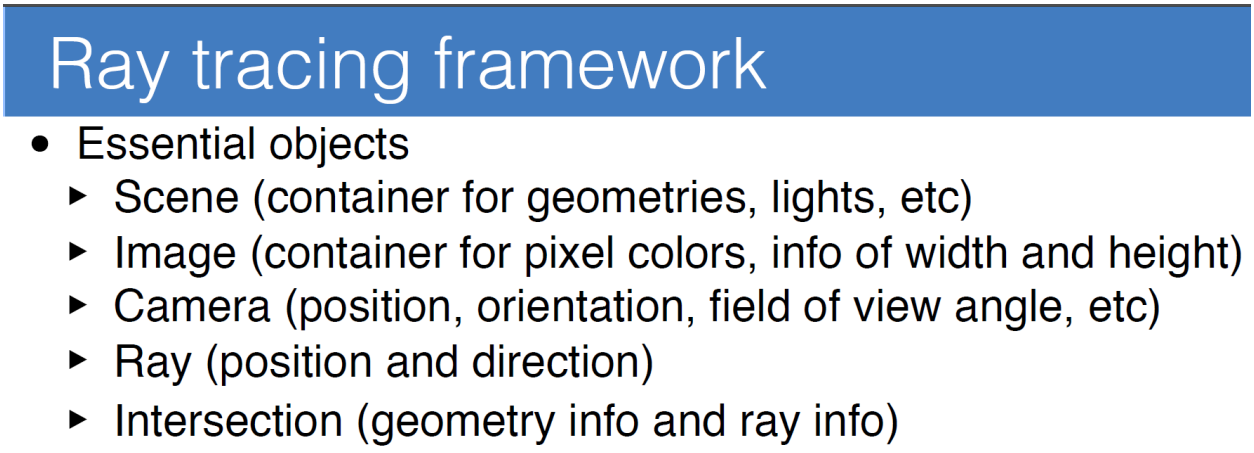
RTObj.h is modified as RTGeometry is created.

In RTObj.cpp, we wrote the post processing of triangles. Similar to RTCube.h, given datas, we store the positions and normals at corresponding index into each triangle and insert it into elements list.

RTScene:

In RTScene, we traverse through a scene in camera view using stack and dfs algorithm which is same as HW3. When drawing all the models, we are working on world coordinate, using the modelview matrix * position and inverse transpose * normal to transform the positions and normals correctly, as discussed in Lighting lecture.

RayTracer:

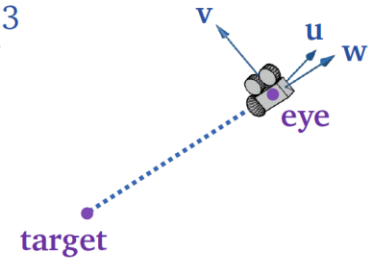


In Raytracer, we have a basic function call that calls:

RayThruPixel - generates a ray originated from the camera position through the center of the (i,j) pixel into the world;

- A **camera** has position and orientation described by

$$\mathbf{eye} \in \mathbb{R}^3 \quad \mathbf{u} \in \mathbb{R}^3 \quad \mathbf{v} \in \mathbb{R}^3 \quad \mathbf{w} \in \mathbb{R}^3$$



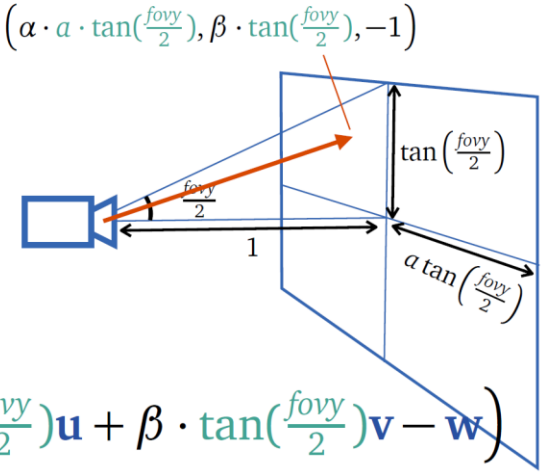
- Recall that the camera matrix is

$$\mathbf{C} = \begin{bmatrix} | & | & | & | \\ \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{eye} \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Given $\mathbf{eye} \in \mathbb{R}^3$ $\mathbf{target} \in \mathbb{R}^3$ $\mathbf{up} \in \mathbb{R}^3$

$$\mathbf{w} = \frac{\mathbf{eye} - \mathbf{target}}{|\mathbf{eye} - \mathbf{target}|} \qquad \mathbf{u} = \frac{\mathbf{up} \times \mathbf{w}}{|\mathbf{up} \times \mathbf{w}|} \qquad \mathbf{v} = \mathbf{w} \times \mathbf{u}$$

- Given camera $\mathbf{eye} \in \mathbb{R}^3$ $\mathbf{u} \in \mathbb{R}^3$ $\mathbf{v} \in \mathbb{R}^3$ $\mathbf{w} \in \mathbb{R}^3$ $a = \frac{\text{width}}{\text{height}}$ $fovy$
- Given pixel (i, j)
- In camera coordinate,
  - Source of ray = (0,0,0)
  - Ray passes through $\left(\alpha \cdot a \cdot \tan(\frac{fovy}{2}), \beta \cdot \tan(\frac{fovy}{2}), -1\right)$



- In world, the ray is given by

$$\mathbf{p}_0 = \mathbf{eye}$$

$$\mathbf{d} = \text{NORMALIZE}\left(\alpha \cdot a \cdot \tan(\frac{fovy}{2})\mathbf{u} + \beta \cdot \tan(\frac{fovy}{2})\mathbf{v} - \mathbf{w}\right)$$

Intersect(ray, scene) - searches over all geometries in the scene and returns the closest hit, it calls Intersect(ray, triangle) that performs the Ray-triangle intersection discussed in class; RayTracing ppt p31

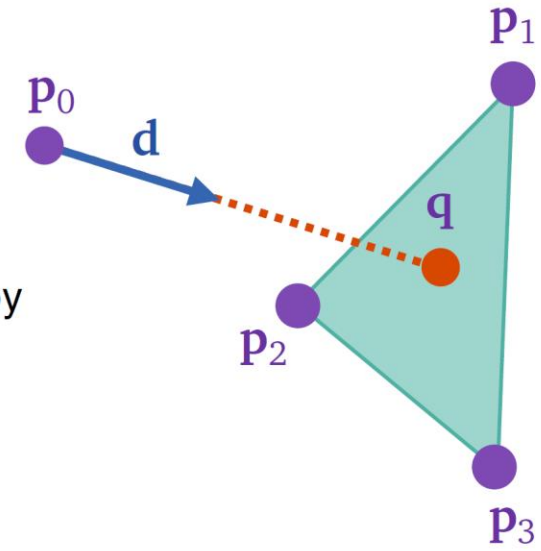
Ray-triangle intersection

- Given ray $(\mathbf{p}_0, \mathbf{d})$
- Given triangle $\mathbf{p}_1\ \mathbf{p}_2\ \mathbf{p}_3$
- Any point along the ray takes the form

$$\mathbf{q} = \mathbf{p}_0 + t\mathbf{d}$$

- Any point on the plane spanned by the triangle takes the form

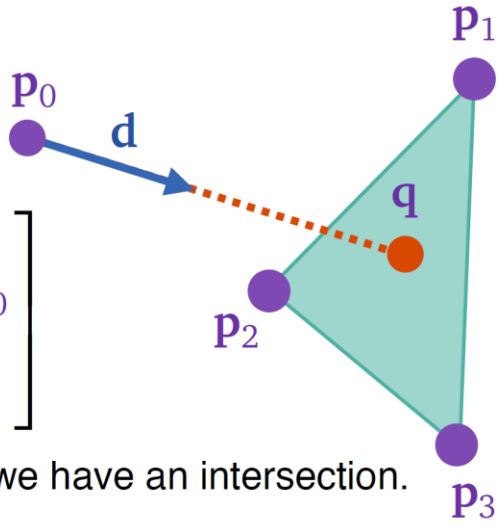
$$\mathbf{q} = \lambda_1\mathbf{p}_1 + \lambda_2\mathbf{p}_2 + \lambda_3\mathbf{p}_3$$
$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$



$$\begin{cases} \lambda_1\mathbf{p}_1 + \lambda_2\mathbf{p}_2 + \lambda_3\mathbf{p}_3 - t\mathbf{d} = \mathbf{p}_0 \\ \lambda_1 + \lambda_2 + \lambda_3 = 1 \end{cases}$$

Solve

$$\begin{bmatrix} | & | & | & | \\ \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 & -\mathbf{d} \\ | & | & | & | \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ t \end{bmatrix} = \begin{bmatrix} | \\ \mathbf{p}_0 \\ | \\ 1 \end{bmatrix}$$



If all $\lambda_1, \lambda_2, \lambda_3$ and t are ≥ 0 then we have an intersection.

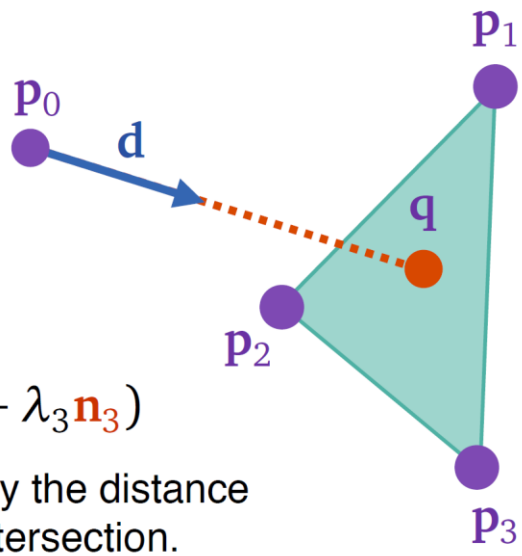
- If we have an intersection, use the barycentric coordinate (what we just solved)

  $$\lambda_1, \lambda_2, \lambda_3$$

  to interpolate position and vertex attributes, such as normals

  $$\mathbf{q} = \lambda_1 \mathbf{p}_1 + \lambda_2 \mathbf{p}_2 + \lambda_3 \mathbf{p}_3$$

  $$\mathbf{n} = \text{normalize}(\lambda_1 \mathbf{n}_1 + \lambda_2 \mathbf{n}_2 + \lambda_3 \mathbf{n}_3)$$



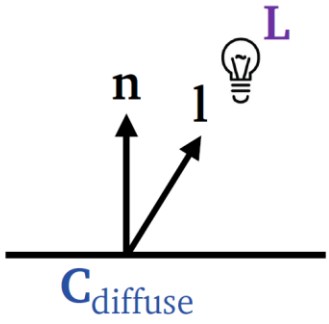
- The variable t we solved is exactly the distance between the ray source and the intersection.

FindColor - shade the light color seen by the in-coming ray which is basically the same as HW3's lighting fragment, but in world coordinate, for each light, we generate its color, also for specular we loop on recursion_depth that is the number of reflected lights, adding up the color in the end will give us the right result.

- Given
  - Unit (normalized) surface normal $\mathbf{n}$
  - Unit (normalized) light direction $\mathbf{l}$
  - Material diffuse reflectance (material color) $\mathbf{C}_{\text{diffuse}}$
  - Light color (intensity) $\mathbf{L}$
- The reflected diffuse color (intensity) is

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

componentwise multiplication; zero out negative cosines; cosine

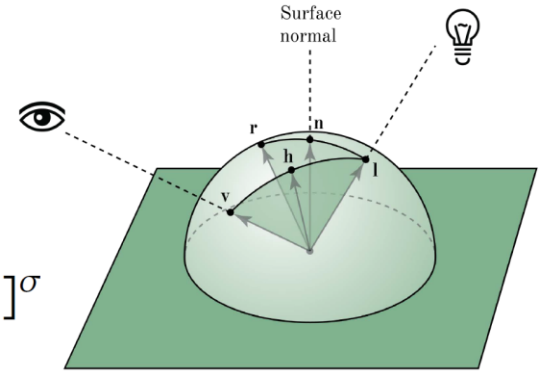


$$\mathbf{R}_{\text{Phong specular}} = \mathbf{C}_{\text{specular}}\mathbf{L}[\max(\mathbf{v}\cdot\mathbf{r}, 0)]^{p}$$

- Compute the half-way vector

$$\mathbf{h} = \frac{\mathbf{v}+\mathbf{l}}{|\mathbf{v}+\mathbf{l}|}$$

- Replace $(\mathbf{v}\cdot\mathbf{r})$ by $(\mathbf{n}\cdot\mathbf{h})$



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

- For distant light and camera, $\mathbf{h}$ is constant. This can speed up the rendering.

- Phong model supports multiple light sources

$$\mathbf{R} = \mathbf{E} + \sum_j \mathbf{L}_j \left(\mathbf{C}_{\text{ambient}} + \mathbf{C}_{\text{diffuse}} \max(\mathbf{n} \cdot \mathbf{l}_j, 0) + \mathbf{C}_{\text{specular}} [\max(\mathbf{n} \cdot \mathbf{h}_j, 0)]^{\sigma} \right)$$

self-emission

Getting **v**, **n**, **l** correctly

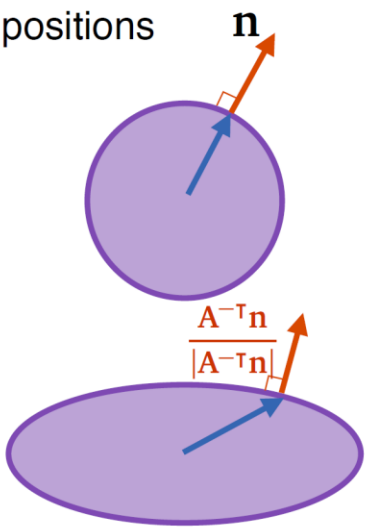
- Suppose we have an affine transformation on positions

$$\begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \mapsto \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$$

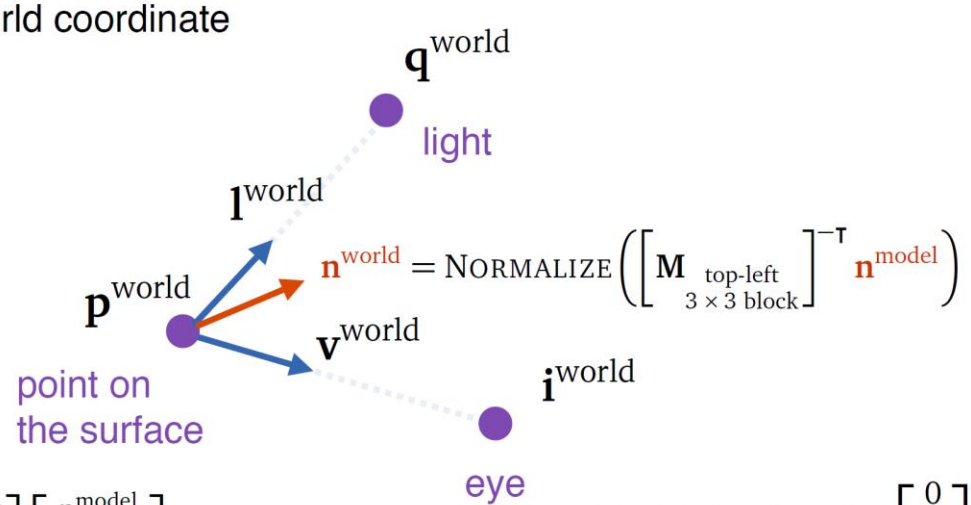
- and normal vectors transform according to

$$\begin{bmatrix} \mathbf{n} \\ \end{bmatrix} \mapsto \begin{bmatrix} \mathbf{A}^{-\mathsf{T}} \\ \end{bmatrix} \begin{bmatrix} \mathbf{n} \\ \end{bmatrix}$$

(followed by a normalization)



- In the world coordinate



$$\begin{bmatrix} \mathbf{p}^{\text{world}} \\ 1 \end{bmatrix} = [\mathbf{M}] \begin{bmatrix} \mathbf{p}^{\text{model}} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{i}^{\text{world}} \\ 1 \end{bmatrix} = [\mathbf{V}^{-1}] \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$