CMSC 427
Computer Graphics[1]

David M. Mount
Department of Computer Science
University of Maryland
Fall 2017

Lecture 1: Ray Tracing and Picking

Ray Tracing: Ray tracing is among the conceptually simplest methods for synthesizing highly realistic images. Unlike the simple polygon rendering methods used by OpenGL, ray tracing can easily produce shadows, and it can model reflective and transparent objects. (Fig. **??** shows an image generated by Andre Kirschner.)

Fig. 1: A ray traced image.

Because it is relatively slow, ray tracing is typically used for generating highly realistic images offline (as opposed to interactively). For example, it is used when interactivity is not needed, as in producing high quality animated films. It is also useful for generating realistic texture maps and environment maps that could later be used in interactive rendering. Ray tracing also forms the basis of many approaches to more producing highly realistic complex types of shading and lighting.

Ray tracing can also be used in the context of interactive computer graphics for a process called *picking*. In picking, we imagine that we have rendered a 3-dimensional scene, and the user has placed the cursor over one of the objects that has been drawn, and we want to know which object the user is referring to. OpenGL provides a number of methods to perform picking. Raytracing can be used for picking. If you imagine a ray shot from the viewer's eye through the screen pixel associated with the mouse coordinates, the first object this ray hits in the scene is the visible object that the user is referring to.

In spite of its conceptual simplicity, ray tracing can be computationally quite intensive (particularly when applied to the generation of complex high resolution images). We will discuss the basic elements of ray tracing, revisit the Phong lighting model in this context, and discuss some of the details of generating rays and handling ray intersections.

Ray Tracing for Image Synthesis: Imagine that the viewing window is replaced with a fine mesh of horizontal and vertical grid lines, so that each grid square corresponds to a pixel in the final image. We shoot rays out from the eye through the center of each grid square in an attempt to trace the path of light backwards toward the light sources. Consider the first

object that such a ray hits. (In order to avoid problems with jagged lines, called *aliasing*, it is more common to shoot a number of rays per pixel and average their results.) We want to know the intensity of reflected light at this surface point. This depends on a number of things, principally the reflective and color properties of the surface, and the amount of light reaching this point from the various light sources. The amount of light reaching this surface point is the hard to compute accurately. This is because light from the various light sources might be blocked by other objects in the environment and it may be reflected off of others.

A purely local approach to this question would be to use the model we discussed in the Phong model, namely that a point is illuminated if the angle between the normal vector and light vector is acute. In ray tracing it is common to use a somewhat more global approximation. We will assume that the light sources are points. For each light source L_i, we shoot a ray R_{L_i} from the surface point to each of the light sources (see Fig. **??**(a)). For each of these rays that succeeds in reaching a light source before being blocked another object, we infer that this point is illuminated by this source (as for L_1 in Fig. **??**(a)), and otherwise we assume that it is not illuminated, and hence we are in the shadow of the blocking object (as for L_2 in Fig. **??**(a)). (Can you imagine a situation in which this model will fail to correctly determine whether a point is illuminated?)

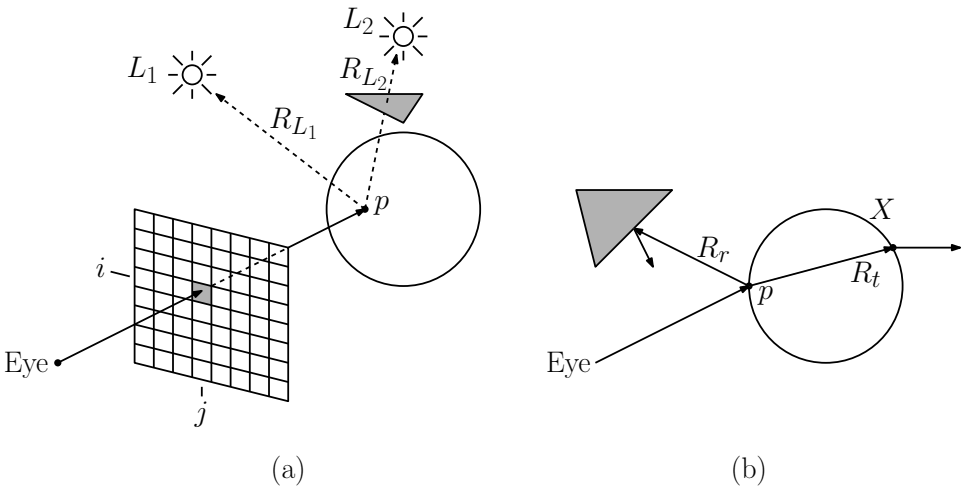

Fig. 2: Ray Tracing.

Given the direction to the light source and the direction to the viewer, and the surface normal (which we can compute because we know the object that the ray struck), we have all the information that we need to compute the reflected intensity of the light at this point, say, by using the Phong model and information about the ambient, diffuse, and specular reflection properties of the object. We use this model to assign a color to the pixel. We simply repeat this operation on all the pixels in the grid, and we have our final image.

Even this simple ray tracing model is already better than what OpenGL supports, because, for example, OpenGL's local lighting model does not compute shadows. The ray tracing model can easily be extended to deal with reflective objects (such as mirrors and shiny spheres) and transparent objects (glass balls and rain drops). For example, when the ray hits a reflective object, we compute the reflection ray and shoot it into the environment. We invoke the ray

tracing algorithm *recursively* (see Fig **??**(b)). When we get the associated color, we blend it with the local surface color and return the result. The generic algorithm is outlined below.

rayTrace() : Given the camera setup and the image size, generate a ray R_{ij} from the eye passing through the center of each pixel (i, j) of your image window (See Fig. **??**.) Call trace(R) and assign the color returned to this pixel.

trace(R) : Shoot R into the scene and let X be the first object hit and p be the point of contact with this object.

(a) If X is reflective, then compute the reflection ray R_r of R at p. Let $C_r \leftarrow \text{trace}(R_r)$.

(b) If X is transparent, then compute the transmission (refraction) ray R_t of R at p. Let $C_t \leftarrow \text{trace}(R_t)$.

(c) For each light source L,

(i) Shoot a ray R_L from p to L.

(ii) If R_L does not hit any object until reaching L, then apply the lighting model to determine the shading at this point.

(d) Combine the colors C_r and C_t due to reflection and transmission (if any) along with the combined shading from (c) to determine the final color C. Return C.

Ray Tracing for Picking: Another application of ray tracing is for the process of identifying the object associated with a particular pixel of the image. Suppose that the user places the cursor over the window at row i and column j. In order to determine the object of the scene that lies under this pixel, we shoot a ray through this pixel, and determine the first object that is hit by the ray.

Ray Representation: Let us consider how rays are represented, generated, and how intersections are determined. First off, how is a ray represented? An obvious method is to represent it by its origin point p and a directional vector $\vec{u}$. Points on the ray can be described *parametrically* using a scalar t:

$$R = \{p + t\vec{u} \mid t > 0\}.$$

Notice that our ray is *open*, in the sense that it does not include its endpoint. This is done because in many instances (e.g., reflection) we are shooting a ray from the surface of some object. We do not want to consider the surface itself as an intersection. (As a practical matter, it is good to require that t is larger than some very small value, e.g. $t \geq 10^{-3}$. This is done because of floating point errors.)

In implementing a ray tracer, it is also common to store some additional information as part of a *ray object*. For example, you might want to store the value t_0 at which the ray hits its first object (initially, $t_0 = \infty$) and perhaps a pointer to the object that it hits.

Ray Generation: Let us consider the question of how to generate rays. Let us assume that we are given essentially the same information that we use in gluLookAt and gluPerspective. In particular, let *eye* denote the eye point, *at* denote the center point at which the camera is looking, and let $\overrightarrow{up}$ denote the "up vector" for gluLookAt. Let $\theta_y = \pi \cdot \text{fovy}/180$ denote the y-field of view in radians. Let n_{rows} and n_{cols} denote the number of rows and columns in the final image, and let $\alpha = n_{cols}/n_{rows}$ denote the window's aspect ratio.

In gluPerspective we also specified the distance to the near and far clipping planes. This was necessary for setting up the depth buffer. Since there is no depth buffer in ray tracing, these values are not needed, so to make our life simple, let us assume that the window is exactly one unit in front of the eye. (The distance is not important, since the aspect ratio and the field-of-view really determine everything up to a scale factor.)

The height and width of view window relative to its center point are

$$h = 2\tan\frac{\theta_y}{2} \qquad w = h \cdot \alpha.$$

So, the window extends from $-h/2$ to $+h/2$ in height and $-w/2$ to $+w/2$ in width (see Fig. **??**.)

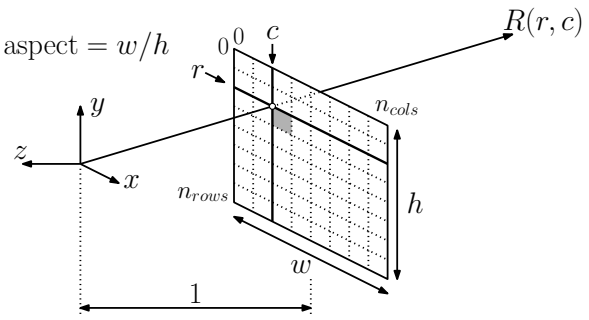

Fig. 3: Ray generation.

Next, we proceed to compute the viewing coordinate frame, very much as we did in our earlier lecture on 3-dimensional viewing. The origin of the camera frame is *eye*, the location of the eye. The view vector is directed from the eye to the point we are looking at, that is,

$$\overrightarrow{view} \;=\; \text{normalize}(at - eye)$$

Recall from the earlier lecture that the unit vectors for the camera frame are:

$$V.\vec{v}_z \;=\; -\overrightarrow{view}, \qquad V.\vec{v}_x \;=\; \text{normalize}(\overrightarrow{view} \times \overrightarrow{up}), \qquad V.\vec{v}_y \;=\; (V.\vec{v}_z \times V.\vec{v}_x).$$

Next, we need to determine the point on the image plane corresponding to row r and column c. Since we are interested in using this for picking, we will follow the standard convention (used, e.g., by GLUT) that rows are indexed from top to bottom, and columns are indexed from left to right. Every point on the view window has $V.\vec{v}_z$ coordinate of -1. Now, suppose that we want to shoot a ray for row r and column c, where $0 \le r < n_{rows}$ and $0 \le c < n_{cols}$. Observe that r/n_{rows} is in the range from 0 to 1. Multiplying by $-h$ maps us linearly to the (inverted) interval $[0, -h]$ and then adding $h/2$ yields the final desired interval $[h/2, -h/2]$. This means that the vertical offset of row r is:

$$a_y \;=\; -h\frac{r}{n_{rows}} + \frac{h}{2} \;=\; -h\left(\frac{r}{n_{rows}} - \frac{1}{2}\right).$$

By an analogous reasoning, the horizontal offset of column c is

$$a_x \;=\; w\left(\frac{c}{n_{cols}} - \frac{1}{2}\right).$$

Therefore, the location of the point corresponding to row r and column c on the image plane is

$$p[r,c] \;=\; eye + a_x \cdot V.\vec{v}_x + a_y \cdot V.\vec{v}_y - V.\vec{v}_z.$$

(Since *eye* is a point and the other three components involve the product of a scalar and a vector, it follows that $p[r,c]$ is a valid affine equation for a point in space.) The direction vector for the ray emanates from the eye and passes through this point, thus, it is

$$\vec{u}[r,c] \;=\; \text{normalize}(p[r,c] - eye).$$

Thus, the desired ray $R[r,c]$ (see Fig. **??**) has the origin *eye* and the directional vector $\vec{u}[r,c]$. In conclusion, the desired ray is:

$$R[r,c] \;=\; eye + t \cdot \vec{u}[r,c], \qquad (\text{for } t > 0).$$

Now that we have the desired ray, we next consider how to determine what it hits and where it hits it.

Rays and Intersections: Given an object in the scene, a *ray intersection procedure* determines whether the ray intersects and object, and if so, returns the value $t' > 0$ at which the intersection occurs. (This is a natural use of object-oriented programming, since the intersection procedure can be made a member function of the object.) Otherwise, if t' is smaller than the current t_0 value, then t_0 is set to t'. Otherwise the trimmed ray does not intersect the object. (For practical purposes, it is useful for the intersection procedure to determine two other quantities. First, it should return the normal vector at the point of intersection and second, it should indicate whether the intersection occurs on the inside or outside of the object. The latter information is useful if refraction is involved.)

Ray-Plane Intersection: Any plane in 3-dimensional space can be expressed as the set of points $p = (x, y, z)$ that satisfy a linear equation, that is

$$H : ax + by + cz + d = 0,$$

for some real-valued coefficients a, b, c, and d. For example, the plane containing the x, y-axes is the plane $z = 0$, which corresponds to the vector $(a, b, c, d) = (0, 0, 1, 0)$.

The question we wish to consider is, given a plane H, and given a ray $R : p + t\vec{u}$, does the ray hit the plane, and if so, where?

To determine the value of t where the ray intersect the plane, we could plug the ray's parametric representation into this equation and simply solve for t. If the ray is represented by $p + t\vec{u}$, then we have the equation

$$a(p_x + tu_x) + b(p_y + tu_y) + c(p_z + tu_z) + d = 0.$$

Soving for t we obtain

$$t_0 = -\frac{ap_x + bp_y + cp_z}{au_x + bu_y + cu_z}.$$

Note that the denominator is zero if the ray is parallel to the plane. We may simply assume that the ray does not intersect the polygon in this case (ignoring the highly unlikely case

where the ray hits the polygon along its edge). Once the intersection value t_0 is known, the actual point of intersection is just computed as $p + t_0\vec{u}$. If the value of t_0 is negative at the intersection point, then the intersection lies behind the viewer's eye, and it may be ignored. Otherwise, we report that $p + t_0\vec{u}$ is the intersection point.

Normal Vector: For the sake of computing lighting, it is necessary to have a surface normal vector. In the case of a plane, this is very easy to compute. In particular, if the plane's equation is $ax + by + cz + d = 0$, then the vector (a, b, c) is orthogonal to the plane. (For example, in the case of the plane $z = 0$, the equation is given by the coefficient vector $(a, b, c, d) = (0, 0, 1, 0)$, and hence the vector $(0, 0, 1)$ is normal to the plane's surface, as expected.