Lecture 2: Introduction to Ray Tracing

Yong-jin Kwon

September 13, 2009

1 Background

1.1 Ray Casting

The first ray casting algorithm for rendering was first introduced by Arthur Appel in 1968. Ray casting rendered the scene by shooting one ray per pixel from the eye and finding the closest object blocking the path of the ray. One significant advantage of ray casting over traditional scanline rendering algorithms was the ability to deal with non-planar sufaces and solids. Much of the animation in Tron was rendered using ray casting techniques.

1.2 Ray Tracing

An evolution in ray casting rendering came in 1979 when Turner Whitted continued the ray casting process by introducing reflection, refraction, and shadows. A reflected ray continues on in the mirror-reflection direction from a shiny surface. The reflected color is determined by the intersection of the reflected rays with objects in the scene. A refracted ray is created similarly to reflected rays except its direction is into the object and can eventually exit the object. The shadow is computed by creating shadow rays which originates from the intersection to all lights. If the shadow ray intersects an object before it reaches the light, then that intersection point is shadowed from that particular light.

1.3 Advantage/Disadvantage

The main advantage of ray tracing is its realitic rendering of reflections, refractions and shadows. Once the ray object intersection is coded, reflection, refractions, and shadows are able to be added very easily. In addition, anti-aliasing and depth of field effects are easily acchieved using ray tracing.

The most serious disadvantage is the heavy computational requirement of the ray tracing algorithm. Advanced lighting effects such as caustics are difficult to render usin ray tracing.

2 Algorithm

The general algorithm of ray tracing is performed by tracing a path from an imaginary eye through each pixel in a virtual screen, and calculating the color of the object visible through it.

Each ray is tested for interesection with objects and once the nearest intersection has been identified, the algorithm estimates the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel.

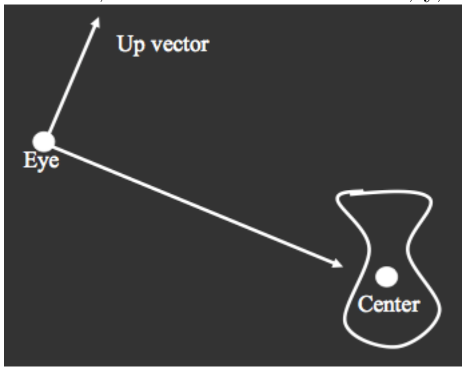
The algorithm is divided into five sections: Camera Ray casting, Ray-object intersection, dealing with object transformation, lighting calculations, and recursive ray tracing.

2.1 Camera Ray Casting

The first step in the algorithm is camera ray casting. This is very similar to the traditional ray casting algorithm and its ultimate goals it to find the ray direction given pixel i, j in the virtual screen.

2.1.1 Coordinate Frame

We must first set up the camera coordinate frame. We want to position our camera at the origin looking down at the -Z axis. Since the world coordinate frame is set up differently, we want to set up coordinate frame w, u and v in terms of the x, y, z.

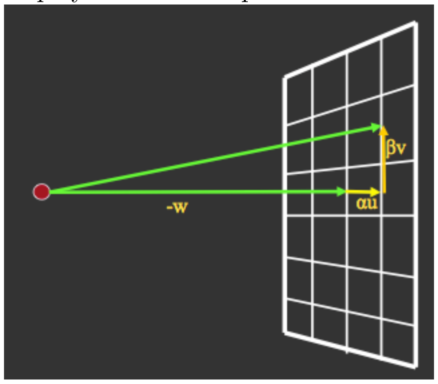


$w = \frac{a}{||a||}$
$u = \frac{b*w}{||b*w||}$
$v = w * u$

$a : center \rightarrow eye$
$b : upvector$

2.1.2 Canonical Viewing Geometry

Once we have the coordinate frame set up, we can use the following information to extract the ray direction given i, j in the virtual screen we set up as our output. Whatever we see on the virtual screen will be displayed on the output



$\alpha = tan(\frac{fovx}{2} * \frac{j-(width/2)}{width/2})$
$\beta = tan(\frac{fovy}{2} * \frac{(height/2)-i}{height/2}$
$ray = eye + \frac{\alpha u+\beta v-w}{|\alpha u+\beta v-w|}$

fovx : the horizontal size of the virtual screen
fovy : the bertical size of the virtual screen

i : horizontal index (left = 0)
j : vertical index (top = 0)

2.2 Ray object Intesection

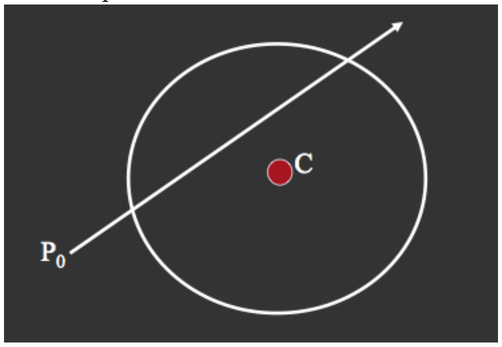
Now that we created the rays from the eye, we need a way to test whether these rays intersect any objects. The method to check object-ray intersection is different for each type of primitive. However, the general methodology is similar. Given the ray parametrc equation below, we substitute the ray into the parametric equation of the primitive we want to check against.

$ray \equiv \vec{P} = \vec{P_0} + \vec{P_1}t$

$\vec{P}0$: origin
$\vec{P}1$: direction

2.2.1 Ray-Sphere

A sphere is a good choice because it is easy to represent parametically, and it is hard to create using triangles. We take the sphere parametric equation and substitute the ray parametric equation to determine if our ray hit the sphere.



$sphere \equiv (\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) - r^2 = 0$
Substitute: $\vec{P} = \vec{P_0} + \vec{P_1}t$
$sphere \equiv (\vec{P_0} + \vec{P_1} - \vec{C}) \cdot (\vec{P_0} + \vec{P_1} - \vec{C}) - r^2 = 0$
$sphere \equiv t^2(\vec{P_1} \cdot \vec{P_1}) + 2t\vec{P_1} \cdot (\vec{P_0} - \vec{C}) + (\vec{P_0} - \vec{C}) \cdot (\vec{P_0} - \vec{C}) - r^2 = 0$

After the substitution we have a quadratic equation that we can solve for t. To determine if there is an intersection, we can first compute the discriminant. If the discriminant is negative, then there is no intersection. If the discriminant is positive, then we can solve for t to determine the intersection point. Here are the possible outcomes when solving for t:

- **2 Real Roots**: This means that the ray hits the sphere in two locations. Since we only care about the closest intersection, pick the smaller root
- **One Root**: A degenerate case of the 2 real roots case where the ray is tangent to the sphere.
- **One Positive, One Negative root**: This is a case where the ray originates from within the sphere and intersects the sphere from the inside. Select the positive root to find out where the intersection is
- **Complex Roots**: The sphere and the ray does not intersect

Intersection Point = $\vec{P_0} + \vec{P_1}t$
Normal = $\frac{(\vec{P} - \vec{C})}{|(\vec{P} - \vec{C})|}$

2.2.2 Ray-Plane

A efficient ray-triangle intersection algorithm is optimal because triangles are used abundantly to render scenes. One way to compute whether a ray hit a triangle is to first find out if and where the ray hit the plane that the triangle lies on. The methodology is very similar to the ray-sphere intersection method. Plug the ray parametric equation into the plane parametric equation and solve for t.

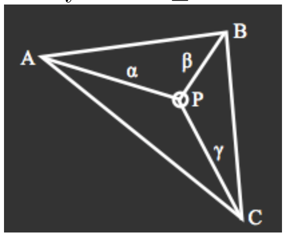
$plane \equiv (\vec{P} \cdot \vec{n} - \vec{A} \cdot \vec{n}) = 0$

$n = \frac{(C-A)*(B-A)}{|(C-A)*(B-A)|}$
Substitute: $\vec{P} = \vec{P_0} + \vec{P_1}t$
$t = \frac{\vec{A}\cdot\vec{n} - \vec{P_0}\cdot\vec{n}}{\vec{P_1}\cdot\vec{n}}$

Notice how t is undefined when the plane and the ray directions are parallel

2.2.3 Ray-Triangle

If the ray hits the plane, we can further test if the point of intersection is inside the triangle. Barycentric coordinates are a good way to test this. First find the barycentric coordinates of the intersection points and if they are all ≥ 0 then the intersection is inside the triangle



$P = \lambda_1 A + \lambda_2 B + \lambda_3 C$
$\lambda_1 + \lambda_2 + \lambda_3 = 1$

$x = \lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3$
$y = \lambda_1 y_1 + \lambda_2 y_2 + \lambda_3 y_3$
substitute $\lambda_3 = 1 - \lambda_1 - \lambda_2$
$\lambda_1(x_1 - x_3) + \lambda_2(x_2 - x_3) + x_3 - x = 0$
$\lambda_1(y_1 - y_3) + \lambda_2(y_2 - y_3) + y_3 - y = 0$
Solve for lambda:
$\lambda_1 = \frac{(y_2-y_3)(x-x_3)-(x_2-x_3)(y-y_3)}{det(T)}$
$\lambda_2 = \frac{-(y_1-y_3)(x-x_3)-(x_1-x_3)(y-y_3)}{det(T)}$
$\lambda_3 = 1 - \lambda_1 - \lambda_2$
where T = $\begin{pmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{pmatrix}$

If $\lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0$ then ray intersects inside of triangle

2.3 Transformed objects

What if we want to perform ray-ellpisoid intersection checks? One way is to make a new intersection function with rays and ellipsoids. But a much more efficient method is to use the ray-sphere intersection and apply a transformation to the sphere. There is a easy way of checking for an intersection of a ray and a transformed object

Given transform M to object, apply M^{-1} to the ray and do the regular ray-object intersection. Then transform the intersection back into real coordinate space. However, the normal does not transform with transformation M. The normal is defined by $n^T p = 0$:
Normal = M^{-t}n
$p \rightarrow Mp$
$n \rightarrow Qn$
Plugging in for p and n, we want to solve for the transformation Q which we need to apply to the normal of p with transformation of M.

$(Qn)^T(Mp) = 0$
$n^T Q^T Mp = 0$
$Q^T M = I$
$Q = M^{-T}$

2.4 Lighting Calculations

2.4.1 Shadows

Shadow effects are very easy to do with rays. When a ray intersects an object, create a ray from the intersection point to light source and see if it intersects anything. If there is an intersection, that means something is casting a shadow from the light source to the intersection point.

Caveat: The shadow ray may hit the original intersected objects causing false self shadowing. One solution is to move the ray forward a small amount before starting the intersection calculation

2.4.2 Shading

The following is an example shading algorithm that we can use to compute the color of the intersection. Notice that the K_e value would be the global illumination value which is difficult to compute using simple ray tracing.

$I = K_a + k_e + \sum_{i=1}^{n} V_i L_i max(l_i \cdot n, 0) + K_s max(h_i \cdot n, 0))^s)$
K_a : ambient
K_e : emission from material
K_d : diffuse
K_s : specular
V_i : visibility
L_i : intensity
l_i : light vector
h_i : reflection vector
s : specularity

2.5 Recursive Ray Tracing

For each pixel

- Trace Primary Eye Ray and find intersection
- Trace secondary shadow rays to all lights
- Trace Reflected/Refracted Ray

2.5.1 Shading Model

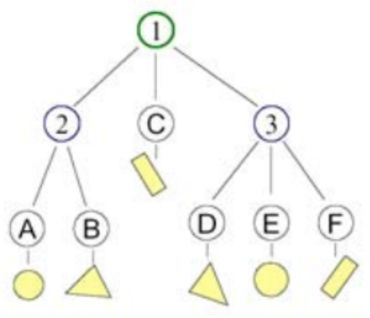
With recursive ray tracing, the last two elements adds reflection and refraction into the shading algorithm. Generally the ray tracer defines how many levels of recursion it uses to limit the number of ray created

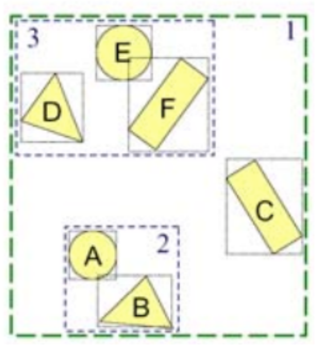
$$I = K_a + k_e + \sum_{i=1}^{n} V_i L_i max(l_i \cdot n, 0) + K_s max(h_i \cdot n, 0))^s) + K_R I_R + K_r I_r$$

2.5.2 Acceleration Structure

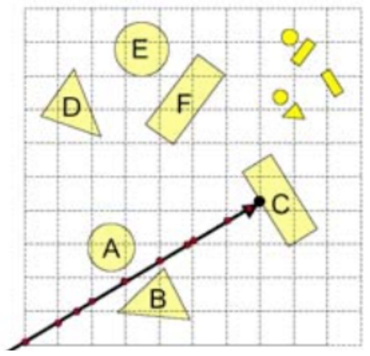
Acceleration structures are used to limit the number of objects to check to find the intersection. For example, if we have a ray and it intersects one object from a scene with thousands of objects, we would like to somehow intelligently weed off the objects which are far away from the ray.

- Bounding Volumes - Build hierarchical structure of bounding volumes. If bounding bolume is hit, check its children recursively

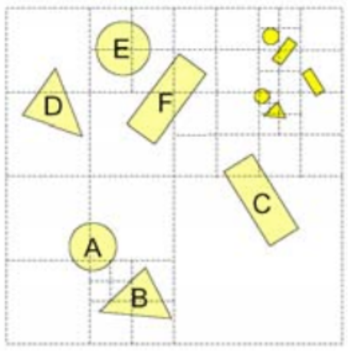




- Uniform Grid - Generate a uniform grid and create a structure which links each grid position with one more more objects which are in the grid location. For each grid that the ray touches, check the objects to see if the ray intersects them



- Octreee - Similar to uniform grid except the grid is generated dynamically by dividing each grid into octants


E
D
F
C
A
B