COSC 3P98: Ray Tracing Basics

Brian J. Ross
Dept. of Computer Science
Brock University
bross@brocku.ca

April 14, 2023

Abstract

These notes outline the technical information needed to implement a basic recursive ray tracer. Effects discussed include basic lighting, shadows, reflection, and refraction. Methods for computing intersections and setting up the camera are also discussed. A list of other helpful resources is given.

Contents

1 Introduction

The primary source of information for these notes is the book *An Introduction to Ray Tracing* by A.S. Glassner [2].

Ray tracing is a photorealistic rendering technique first proposed in 1979, and developed throughout the 1980's. It is capable of a high degree of photorealism, and is the main rendering technique used in most high-end 3D rendering and animation systems. It automatically accounts for such things as:

- localize lighting: specular, diffuse, ambient
- reflection
- refraction (transmission)
- shadows
- hidden surface determination

It also has the advantage of using a straight-forward algorithm, that is amenable to many optimizations.

Ray tracing has been said to produce "super-realistic" images. The high degree of detail is instantly recognizable in single images and animations, in which perfectly smooth spheres and surfaces, with sharp details and shadows, are the norm. However, such clarity is often pointed out as being a disadvantage as well, since scenes in the real world are rarely as pristine and perfect. In this sense, ray tracing by itself has not succeeded in creating photo-realistic images (a "holy grail" of computer graphics research), since the images do not portray the complexity and messiness of shading we usually see in the world around us. Since the introduction of the original ray tracing algorithm many years ago, much research has investigated ways to enhance ray tracing to make it more realistic (and less "super-realistic"). Other algorithms, such as radiosity rendering, have been proposed to tackle some of these issues as well.

Another disadvantage of ray tracing is that it is a computationally expensive algorithm. Each pixel must be separately analyzed, to determine what object in the scene is seen at that pixel location. This involves performing intersections between the rays cast through all the pixels, and the geometric objects composing the scene (Section 4). Furthermore, if the scene involves multiple reflective and/or refractive surfaces, then the recursive nature of the algorithm will create even more calculations. As a result, ray tracing is usually considered to be an off-line "batch mode" algorithm, and unsuitable for real-time processing. A single frame of an animation will usually take minutes or hours to render. (However, this is changing... more later!)

2 Mathematical Preliminaries

The following are review material for some important mathematical concepts required for a ray tracer. These notes assume that you have some knowledge of

basic algebra and geometry.

2.1 Rays

A ray is a mathematical object, that has an origin and a direction. The eye ray in the previous section is defined by an origin R_o and a direction $\vec{R}_d$:

$$R_o = (x_o, y_o, z_o) \qquad \vec{R}_d = (x_d, y_d, z_d)$$

The ray itself is denoted by:

$$\vec{R}(t) = R_o + \vec{R}_d * t \quad (t > 0) \quad [1]$$

where t is a numeric parameter defining a point in space along the ray. Note that if $t = 0$, then you are at R_o. If $t < 0$, then you are actually behind the origin, in the opposite direction of the ray.

Example: Let the ray origin be $R_o = (2, 1, 8)$, and let the ray be aimed through the point P at (1,2,-3). The direction D is computed as:

$$\begin{aligned} \vec{D} &= P - R_o \\ &= (1, 2, -3) - (2, 1, 8) \\ &= (-1, 1, -11) \end{aligned}$$

All of the intersection, lighting, and ray casting calculations assume that all ray directions are *normalized* (their magnitudes are 1). Therefore, dividing D by its magnitude creates a normalized direction:

$$\begin{aligned} \vec{R}_d &= \vec{D} / \, |D| \\ &= (-1, 1, -11) / \sqrt{(-1)^2 + 1^2 + (-11)^2} \\ &= (-1, 1, -11) / 11.091 \\ &= (-0.0902, 0.0902, -1.082) \end{aligned}$$

Note that rays should never have a magnitude of 0 (ie. direction vector of (0,0,0)).

2.2 Dot products and cosines

Given two vectors $\vec{V_0} = (x_0, y_0, z_0)$ and $\vec{V_1} = (x_1, y_1, z_1)$. The dot product is:

$$\vec{V_0} \bullet \vec{V_1} = x_0 x_1 + y_0 y_1 + z_0 z_1$$

Also, this is a known identity:

$$\vec{V_0} \bullet \vec{V_1} = |\vec{V_0}| \;\; |\vec{V_1}| \;\; cos\theta$$

where θ is the angle between the vectors. This then means:

$$cos\theta = \frac{\vec{V_0} \bullet \vec{V_1}}{|\vec{V_0}| \;\; |\vec{V_1}|}$$

If $\vec{V_0}$ and $\vec{V_1}$ are normal, then:

$$cos\theta = \vec{V_0} \bullet \vec{V_1}$$

This is a very efficient way to determine cosines, which are required during many stages of ray tracing. Also, note this trigonometric identity:

$$sin^2\theta = 1 - cos^2\theta$$

So sine can be denoted in terms of cosine, and thus dot products as well.

3 Basic Recursive Ray Tracing Algorithm

In the real world, light sources send off streams of photons in all directions. These photons bounce off of different objects in the world (the sky, walls, trees, people, mirrors...). Some of these photons are on a path that eventually enter our eyes, and hit receptors on the retina at the back of our eyeball. We see the world before us.

The above could be simulated by a computer by a "forward ray tracer". Here, a multitude of photons are emitted from light source(s), and bounced around a virtual world. The simulated photons that happen to hit a "screen of pixels" would be coloured appropriately. This would result in a highly impressive rendered scene. Unfortunately, such an algorithm is not practical. For one thing, there are far too many photons to simulate. Secondly, only a small fraction of photons actually hit the pixel screen (or the retina in the previous paragraph). Most calculations are therefore wasted.

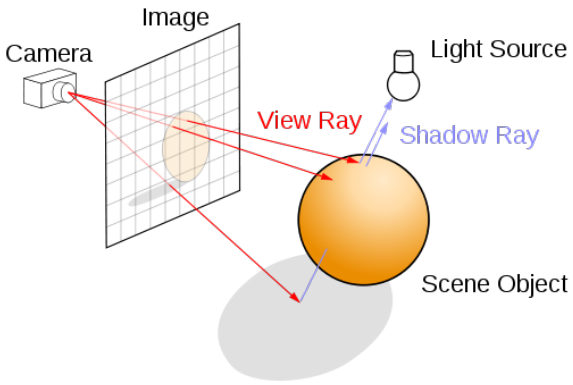

Figure 1: Basic ray trace step (from [5])

Ray tracing makes a clever optimization to the above. Rather than simulating *all* photons from the light source to the eye (which is impossible), the ray tracer algorithm goes backwards. It starts at a pixel on the eye (camera), sends

```
Colourtype RayTrace( rvec, rec_depth)                                    1
    If (rec_depth > MAX_DEPTH)                                           2
        Colour = background_colour                                       3
    Else {                                                               4
        If RayIntersection (rvec, hitobj, hitpt, hitnorm) {              5
            local_colour = shade(hitobj, hitpt, hitnorm)                 6
            If has_refl(hitobj) {                                        7
                refl_vec = calc_reflection(rvec, hitobj, hitpt, hitnorm);  8
                reflect_colour = RayTrace((hitpt, refl_vec), rec_depth+1)  9
            } // if                                                      10
            If has_trans(hitobj) {                                       11
                trans_vec = calc_transmission(rvec, hitobj, hitpt, hitnorm)  12
                trans_colour = RayTrace((hitpt, trans_vec), rec_depth+1)  13
            } // if                                                      14
            colour = local_colour + reflect_colour + trans_colour        15
        } // if                                                          16
        Else                                                             17
            colour = background_colour                                   18
    } // else                                                           19
    return ( colour )                                                    20
} // program                                                            21
```

Table 1: Ray Tracer Algorithm (based on [4])

a *ray* from that point through the middle of a pixel on the screen, and casts this ray into the scene (see Figure 1). This ray represents the single photon from the scene that is hitting that pixel. Should the ray hit an object, the point it first hits the object (the *hit point*) is determined, the lighting at that hit point is calculated, and the pixel is coloured appropriately. If that object has reflective properties, the path of the reflection vector for the photon at that point is determined, and the algorithm continues. This occurs on a pixel-by-pixel basis, until the entire set of pixels is coloured.

Although it was said earlier that ray tracers are computationally expensive, the big advantage of ray tracers is that they are *hugely more efficient* than an actual forward- casting light-to-eye simulation. The only photons simulated are those that will be seen at each pixel. The penalty paid for this, however, is that the lighting calculations are highly simplified compared to real-world lighting, where there can be multitudes of photons combining together at places in the scene, rather than the few that are used in ray tracers. However, this is a reasonable price to pay, since ray tracing algorithm is practical to implement and use, and generates good-quality images.

Table 1 outlines a basic ray tracer algorithm. The algorithm assumes the existence of a list of objects composing the scene. This list contains the following information about each object:

- type of object (plane, sphere)

- location, size, orientation (possibly rotations, scales, translations)
- material definitions for lighting: diffuse, specular, ambient properties (per R, G, B channel)
- reflection coefficient (0-1)
- transmission (refraction) index
- texture (if used)
- ...

The algorithm also assumes a list of lights illuminating the scene (locations, RGB).

The algorithm uses the following components:

1. *RayIntersection*: The hit point of the object that intersects the current ray vector *rvec* is determined. This means that, if multiple objects are intersected by the ray, the one whose intersection point is *closest* to the eye is returned. If this hit point is found, the object, hit point coordinate, and surface normal at the hit point are returned. They are then used further in the code. However, if no hit point is found, then a default background colour is used (20).
2. *shade*: The local lighting effects at the hit point are determined. This involves using all the lights, and computing the specular, diffuse, and ambient light at the hit point for the object. The object's material definition is used for this purpose. If there are textures being used, they would be used as well.
3. *calc_reflection*: If the object has reflective properties, then the reflection ray is determined at the hit opoint. This is used as a new ray for the recursive call to *RayTrace* at line 9. There is a limit to the number of recursive calls processed (line 2).
4. *calc_transmission*: If the object has refractive properties, then the refractive (transmission) ray is determined, and used in a recursive call to *RayTrace*.

Line 15 determines the final RGB colour of the pixel, by combining the local lighting, reflection and refraction colours. Note that simply adding the RGB values together may result in saturation, where one or more of the RGB channels maxes out. Other mathematically combinations of the colours may be preferable.

Figure 2 shows the various rays involved when the eye ray hits an object. The following 3 rays are combined in the image:

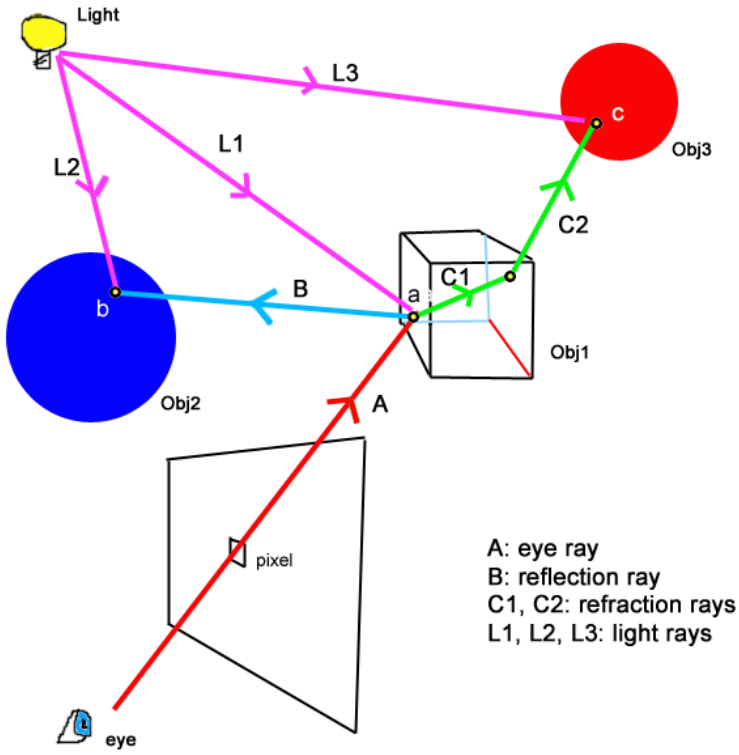

Figure 2: Rays combined at a hit point

1. *Eye Ray (A)*: The eye ray A is set up to originate at the eye (camera) and go through a pixel to be rendered. It hits the side of the cube at a hit point a. The local lighting is then computed at a, taking into account the material of a, the direction to the light $L1$, and the surface normal at a (not shown).

2. *Reflection Ray (B)*: If the cube at a is reflective, then the reflection ray B is computed and placed at hit point a. The ray tracer is recursed on this ray. The diagram shows that it intersects a blue sphere at hit point b. Local lighting is applied at that point, and the colour is computed for ray B.

3. *Refraction Ray C1:* If the cube has refractive properties, the refractionrat $C1$ is computed. It will intersect the cube as it leaves, and a new refraction ray $C2$ will be involved. It intersects the red sphere at hit point c, where local lighting is computed.

The result of the above is that the colours for A, B and C are combined at hit point a, which becomes the colour of the pixel. Note that in steps 2 and 3

above, recursion can cause many more reflections and refractions to occur. The recursive ray tracer will combine computed colours appropriately, resulting in multiple reflection and refraction effects.

4 Finding Intersections

4.1 Intersections of a Ray with a Sphere

Spheres are one of the simplest objects for computing intersections. The following from [2] is an algebraic derivation of a ray-sphere intersection calculation.

First, we have the eye ray as in equation [1] above. A sphere is also present, which has a center $S_c = (x_c, y_c, z_c)$ and a radius S_r. The surface points (x_s, y_s, z_s) of the sphere are defined by the implicit equation:

$$(x_s - x_c)^2 + (y_s - y_c)^2 + (z_s - z_c)^2 = S_r^2$$

The parametric equations of the eye ray are:

$$\begin{aligned} x &= x_o + x_d * t \\ y &= y_o + y_d * t \\ z &= z_o + z_d * t \end{aligned}$$

Substituting these ray equations in the sphere equation lets us solve for the intersection of the ray and sphere:

$$(x_o + x_d t - x_c)^2 + (y_o + y_d t - y_c)^2 + (z_o + z_d t - z_c)^2 = S_r^2$$

A bit of algebra...

$$\begin{aligned} & x_o^2 + 2x_o x_d t - 2x_o x_c - 2x_c x_d t + x_d^2 t^2 + x_c^2 \\ & + y_o^2 + 2y_o y_d t - 2y_o y_c - 2y_c y_d t + y_d^2 t^2 + y_c^2 \\ & + z_o^2 + 2z_o z_d t - 2z_o z_c - 2z_c z_d t + z_d^2 t^2 + z_c^2 - S_r^2 \quad = 0 \end{aligned}$$

and simplifying...

$$\begin{aligned} & (x_d^2 + y_d^2 + z_d^2) t^2 \\ & + t(2x_o x_d - 2x_c x_d + 2y_o y_d - 2y_c y_d + 2z_o z_d - 2z_c z_d) \\ & + x_o^2 + y_o^2 + z_o^2 + x_c^2 + y_c^2 + z_c^2 - 2x_o x_c - 2y_o y_c - 2z_o z_c - S_r^2 \quad = 0 \end{aligned}$$

This is a quadratic equation $A_t^2 + Bt + C = 0$, where:

$$\begin{aligned} A &= x_d^2 + y_d^2 + z_d^2 = 1 \quad (if\ \ eye\ \ ray\ \ is\ \ normalized) \\ B &= 2(x_d(x_o - x_c) + y_d(y_o - y_c) + z_d(z_o - z_c)) \\ C &= (x_o - x_c)^2 + (y_o - y_c)^2 + (z_o - z_c)^2 - S_r^2 \end{aligned}$$

The solutions are...

$$t_0 = \frac{-B - \sqrt{B^2 - 4C}}{2} \qquad t_1 = \frac{-B + \sqrt{B^2 - 4C}}{2}$$

Let $Disc = B^2 - 4C$. To solve:

1. Check discriminant: If $Discr < 0$, then no intersection. Else the smaller positive root is the intersection...

2. Check t_0: if positive, then use it. Else if negative...

3. Check t_1: if positive, use it. (Else if negative then no root.)

If the above finds a suitable positive t value, then it is used to find the sphere intersection point r_i:

$$\begin{aligned} r_i &= (x_i,\ y_i,\ z_i) \\ &= (x_o + x_d t,\ y_o + y_d t,\ z_o + z_d t) \end{aligned}$$

The normal at the hit point is also useful to compute, as it is used for other calculations such as local lighting:

$$\vec{r}_{normal} = \left(\frac{x_i - x_c}{S_r},\ \frac{y_i - y_c}{S_r},\ \frac{z_i - z_c}{S_r} \right)$$

In other words, it is the direction from the sphere centre to the intersection point, normalized by the sphere radius.

Although the above algebraic derivation is straight-forward, it does not provide the most efficient calculation possible. An alternative geometric derivation is given in [2], which will result in a faster calculation. It does this by breaking the intersection down into smaller stages, which permit you to avoid needless calculations when possible. In ray tracers, every skipped mathematical operation helps!

4.2 Intersection of a Ray and a Plane

As before, let the ray be:

$$\begin{aligned} &R_0 = (x_0, y_0, z_0) \\ &\vec{R}_d = (x_d, y_d, z_d) \quad where \quad |\vec{R}_d| = 1 \end{aligned}$$

$$\vec{R}(t) = R_0 + R_d t \quad (t > 0)$$

A plane P is defined by the parameters (A,B,C,D), where:

$$\begin{aligned} &Ax + By + Cz + D = 0 \qquad (A^2 + B^2 + C^2 = 1) \\ &\vec{P}_n = (A, B, C) \qquad (unit\ \ normal\ \ vector) \end{aligned}$$

The distance of this plane to a parallel plane that passes through the origin $(0, 0, 0)$ is $|\, D \,|$.

To find the intersection of $\vec{R}$ and P, substitute the ray's parameteric equations into the equation for P:

$$A(x_0 + x_d t) + B(y_0 + y_d t) + C(z_0 + z_d t) + D = 0$$

Solve for t:

$$t = \frac{-(Ax_0+By_0+Cz_0+D)}{Ax_d+By_d+Cz_d}$$

$$= -\frac{\vec{P}_N \bullet R_0+D}{\vec{P}_N \bullet \vec{R}_d}$$

To implement this:

1. Calculate $\vec{P}_N \bullet R_d = Ax_d + By_d + Cz_d$. If it equals 0, the ray is parallel to the plane, and there is no intersection. If it is > 0, the normal of plane is pointing away from the ray (if 1-sided plane). Else if it is < 0, calculate t in step 2...

2. Calculate t: if $t < 0$, then plane behind ray, and no intersection.

3. Else $t > 0$:

$$r_i = (x_i, y_i, z_i)$$
$$= (x_0 + x_d t,\ y_0 + y_d t,\ z_0 + z_d t)$$

4.3 Intersection of a Ray with a Polygon (triangle)

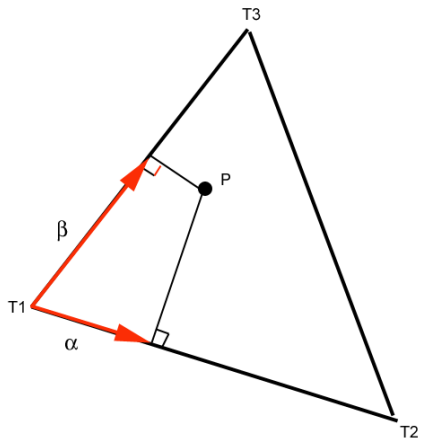

Figure 3: Interior point and triangle in 3D

Here is a basic strategy [1]:

1. Define the equation of the plane on which the triangle is laying upon. Recommended to do this for *all triangles* in the scene *once* at the beginning of execution, and save the equation coefficients with each triangle in the scene.

2. To determine an intersection of a ray with a triangle, first determine if the eye ray intersects the triangle's plane (Subsection 4.2). If it does, then...

3. Do an interior test of a point P and a triangle (T_1, T_2, T_3) in 3D (Figure 3):

   (a) Calculate α, β:

   $$P = \alpha(T_2 - T_1) + \beta(T_3 - T_1)$$

   (b) Check if P inside triangle:

   $$0 \leq \alpha \leq 1 \quad \textit{and} \quad 0 \leq \beta \leq 1$$

5 Local Lighting and Shadows

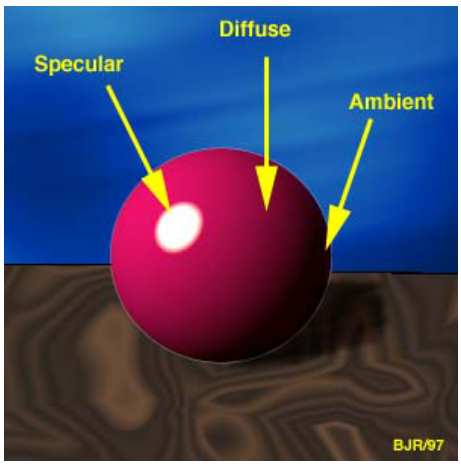

Figure 4: Local lighting

Ray tracers use the same basic local lighting model as used in libraries such as OpenGL. Therefore, a full explanation of these lighting effects can be found in any computer graphics textbook that uses OpenGL.

Figure 4 shows the local lighting effects. Ambient lighting is the lighting resident uniformly in the environment. It permits objects to be seen behind shadows. Diffuse lighting is the light effect seen when a surface reflects an equal amount of light in all directions. The user's eye position is not a factor with diffuse. Specular, on the other hand, is a simulation of the shiny reflection that can occur on some materials. It represents the shininess of the surface material. The intensity and colour of these lighting effects depends on the definition of the materials being assigned to different objects in the scene.

Figure 5 shows some of the factors involved to compute local lighting. As shown in Table 1, local lighting is determined for the closest hit point to the cast ray. This hit point is the point P in Figure 5. The surface normal at P is

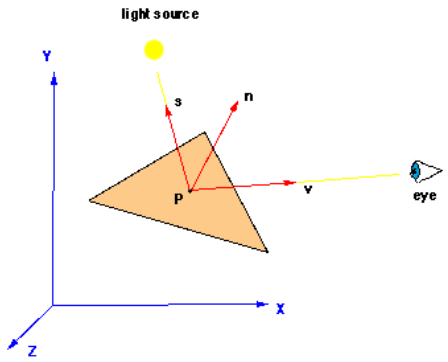

Figure 5: Local lighting factors

important, as well as the direction from P to the eye, as well as the direction to the light source. The position and intensity (colour) of the light source will be of use, as well as the particular material definition on the surface where P resides.

5.1 Ambient

Ambient light is taken to be a constant for the scene. Each material also has an ambient coefficient, to be applied with the environment constant. For example, by setting a strong coefficient for a material, it can be made to "glow" in a scene. The equation for ambient light is (monochrome):

$$I_{amb} = I_a * R_a$$

where I_a is the intensity of ambient light in the environment, and R_a is the ambient surface coefficient for the material ($0 \leq R_a \leq 1$).

5.2 Diffuse

Diffuse lighting depends totally on the the position of a light with respect to the object surface. The viewer's eye position is irrelevant to the diffuse lighting computation. The equation (monochrome) is:

$$I_d = I_s * R_d * cos(A)$$

where I_s is the light intensity, R_d is the material's diffuse coefficient ($0 \leq R'_d leq 1$), and A is the angle of incidence between the surface (hit point) normal and the direction of the light.

Figure 6 illustrates the basic idea of diffuse lighting. When $A = 0$, then there is a maximally direct exposure of the surface to the light: the surface is pointing right at the light, and so it is being lit very intensely. This means that $cos(A) =$

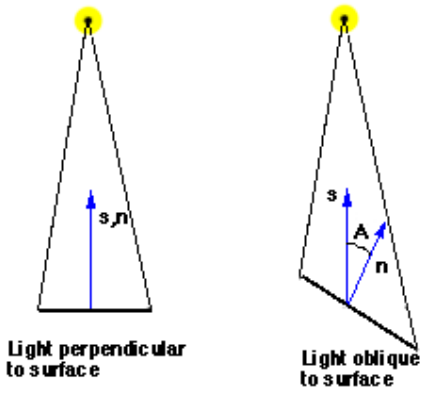

Figure 6: Diffuse lighting

$cos(0) = 1$, which maximizes the diffuse equation above. However, when the light becomes more oblique to the surface, A increases, and $cos(A)$ reduces. This reduces the diffuse intensity. The extreme case is when $A \geq 90$ where $cos(A) \leq 0$ should be considered 0, which will not produce any diffuse light. (Of course, surfaces facing away from the light will not be exposed anyway.)

When computing diffuse lighting, remember to use the dot product to compute the cosine (Subsection 2.2). Also remember to normalize all vectors.

5.3 Specular

Specular lighting is the reflection of light off of shiny surfaces. It causes surface highlights, which is responsible for much of the realism in 3D graphics. The expression for specular lighting is (monochrome):

$$I_{sp} = I_s * R_s * cos(B)^f$$

where I_s is the light intensity, R_s is the material's specular reflection coefficient ($0 \leq R_s \leq 1$), B is the angle between the reflection ray $\vec{r}$ and user eye direction $\vec{v}$, and f is a specular highlight coefficient ($f \geq 1$) for the material.

Consider Figure 7 (top). Specular lighting is strongest when the viewer eye lines up with the reflection vector to the light source. This represents the eye seeing the light reflected on the surface. In the above equation, this means that $B = 0$, and so $cos(0) = 1$. As the eye vector deviates away from this reflection vector, specular reduces, until a point when it has no effect.

To compute the specular equation, the angle B needs to be computed. This requires that the reflection ray for the light (labelled r in Figure 7) must be determined. The other vectors (eye ray is known, and direction to hit point from light is easy to calculate). Section 6 describes how to determine the light's reflection ray.

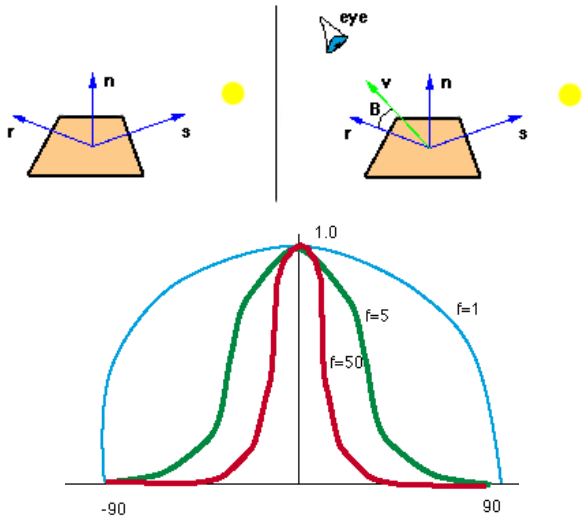

Figure 7: Specular lighting: (top) factors; (bottom) specular highlight.

The degree to which specular reflection "falls out" can be set by the user, in terms of a specular highlight coefficient. Figure 7 (bottom) shows that by changing f, the degree of specular fall off can be controlled.

5.4 Combining together

Ambient, diffuse, and specular lighting are combined together, to give the overall local illumination I for a point on an object surface:

$$I = I_a * R_a + I_s * (R_d cos(A) + R_s * cos(B)^f)$$

The illumination of each point is computed this way. If there are multiple lights in the scene, then it is recomputed for each light, and added together (or combined mathematically some other way).

However, the above discussion has been for monochrome illumination – grey scale light between black and white. To create full colour, each expression above is done for each red, green, and blue colour channel. This means that all the coefficients have R, G, and B components, ambient lighting is RGB, and the lights are also RGB. If you peruse OpenGL's lighting library calls, the material definitions there will show what would also be used in the ray tracer for local lighting.

5.5 Shadows

Shadows are actually an incidental aspect of the local lighting computation. When you are about to use a light to calculate the local lighting for a hit point,

first determine whether there is an unobstructed path from the hit point to the light. In other words, determine whether a ray cast from the hit point to the light intersects any other object in the scene (possibly including the object containing the hit point!). If the intersection tests determine that there is an obstructing surface, then skip the local lighting for that light, and move on to the next light (if any). The effect of this is that an obstructed light will not result in diffuse or specular lighting, and hence that point will be rendered darker than the rest. Ambient light *will* be applied, however, so that some lighting may be resident in shadows.

6 Reflection

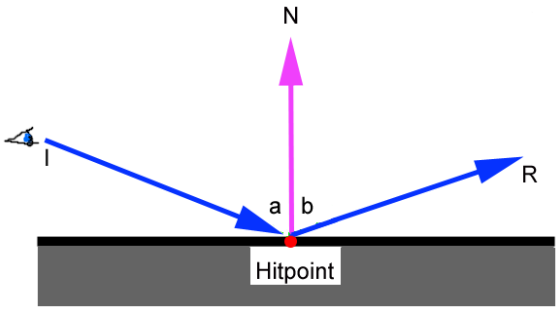

Figure 8: Reflection: Incidence ray I, reflection ray R, surface normal N.

Mirror reflection requires finding a reflection ray, and recursing with it in the ray tracer. Consider Figure 8. The incidence ray $\vec{I}$ (eye ray) is reflected at the hit point, resulting in the reflection ray $\vec{R}$. The normal at the hit point is $\vec{N}$. Angle a = angle b. $\vec{R}$ can be defined as:

$$\vec{R} = \alpha\vec{I} + \beta\vec{N} \qquad (*)$$

In other words, $\vec{R}$ is a linear combination of $\vec{I}$ and $\vec{N}$. Note that $\vec{R}$ is a linear combination of $\vec{I}$ and $\vec{N}$, and $\vec{R}$, $\vec{I}$ and $\vec{N}$ are all planar. Now,

$$cos(a) = (-\vec{I}) \bullet \vec{N}$$

for acute a (or else $\vec{I} \bullet \vec{N}$ gives $cos(\pi + a)$). Also,

$$cos(b) = \vec{N} \bullet \vec{R}$$

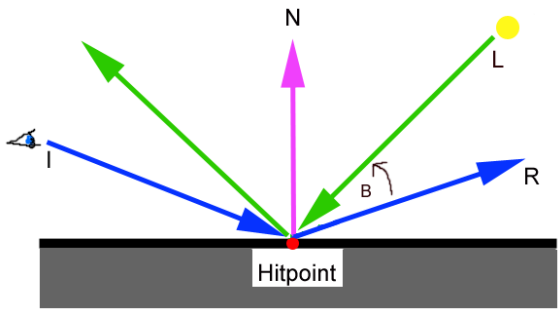

Figure 9: Combining reflection and specular calculations.

Then

$$\begin{array}{rll} cos(a) & = cos(b) & : a = b \\ (-\vec{I}) \bullet \vec{N} = & N \bullet R & : subst.\ \ above\ \ equalities \\ & = \vec{N} \bullet (\alpha\vec{I} + \beta\vec{N}) & : subst.\ \ \vec{R}\ \ (*) \\ & = \alpha(\vec{N} \bullet \vec{I}) + \beta(\vec{N} \bullet \vec{N}) & : distribute\ \ dot\ \ prod. \\ & = \alpha(\vec{N} \bullet \vec{I}) + \beta & : \vec{N}\ \ is\ \ normal \end{array}$$

Let $\alpha = 1$. Then $\beta = -2(\vec{N} \bullet \vec{I})$. So from $(*)$ above...

$$\begin{array}{ll} \vec{R} & = \alpha\vec{I} + \beta\vec{N} \quad or... \\ \vec{R} & = \vec{I} - 2(\vec{N} \bullet \vec{I})\vec{N} \end{array}$$

This can be used to find the reflection ray $\vec{R}$.

A helpful optimization for local lighting is shown in Figure 9. If finding specular lighting and reflection on the same object surface, you can combine the computation of specular angle B in Figure 7 with the computation of the reflection ray here. In other words, once you have $\vec{R}$, then B is the angle between $\vec{R}$ and $-\vec{L}$, or $cos(B) = (-\vec{L}) \bullet \vec{R}$. This prevents you from computing both the reflection vector $\vec{R}$ and the reflection of light!

7 Refraction

Refraction (or transmission) is the effect of light bending through a semi-transparent material. For example, light bends through glass, and through water. The amount of bending depends on the physics of the material, as different materials bend light at different levels. Air bends light a little, while diamond bends it a lot. This bending effect actually happens at the junction of two different materials, for example, glass and air.

When calculating the bending of rays due to refraction, the refractive properties of materials are taken into account. In particular, the index of refraction

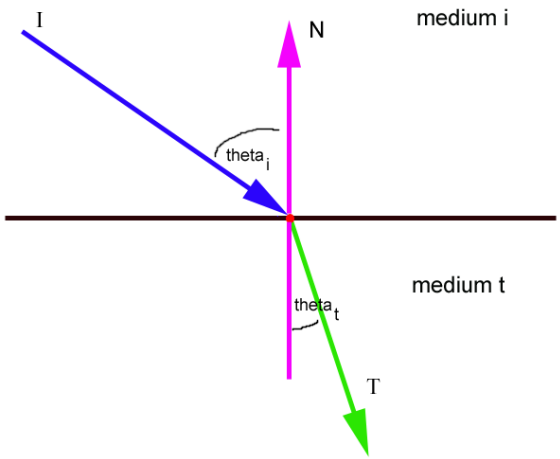

Figure 10: Refraction

Air	1.0003
Water	1.33
Quartz	1.46
Glass	1.5-1.7
Diamond	2.42

Table 2: Some refraction indices

n_i for material i will be used. The ratio of these indices for the two materials in which light is passing will be used. Some example indices are shown in Table 2.

The amount of bending between two materials is determined by Snell's Law:

$$\frac{sin(\theta_i)}{sin(\theta_t)} = \eta_{ti} = \frac{\eta_t}{\eta_i} \quad (*)$$

where i is the material of incidence, t is medium of transmission, η_j is the index of refraction of medium j, and η_{ti} is the index of refraction of medium t with respect to medium i. A higher index means more bending: diamond is high, while air is low.

Figure 10 shows how the incidence vector $\vec{I}$ bends at a hit point between the incidence medium and transmission medium, resulting in a new transmission vector $\vec{T}$. $\vec{N}$ is the normal at the hit point. The transmission vector can be defined as:

$$\vec{T} = \alpha\vec{I} + \beta N$$

Note that:

$$cos(\theta_i) = \vec{N} \bullet -\vec{I} \quad (= C_i)$$

and

$$cos(\theta_t) = -\vec{N} \bullet \vec{T} \quad (= C_t)$$

Take Snell's equation (*), and invert and square it. This gives...

$$Condition\ 1: \quad sin^2\theta_i \eta_{it}^2 = sin^2\theta_t \qquad (2)$$

where $\eta_{ti} = \frac{1}{\eta_{it}}$. Now given the trig identity,

$$sin^2\theta + cos^2\theta = 1$$

we convert (2) to:

$$(1 - cos^2\theta_i)\eta_{it}^2 = 1 - cos^2\theta_t$$

Some algebra...

$$\begin{aligned}
(1 - C_i^2)\eta_{it}^2 &= -C_t^2 \\
&= -(-\vec{N} \bullet \vec{T})^2 \\
&= -(-\vec{N} \bullet (\alpha\vec{I} + \beta\vec{N}))^2 \\
&= -(\alpha(-\vec{N} \bullet \vec{I}) + \beta(-\vec{N} \bullet \vec{N}))^2 \\
&= -(\alpha(-\vec{N} \bullet \vec{I}) - \beta)^2 \\
(1 - C_i^2)\eta_{it}^2 &= -(\alpha C_i - \beta)^2 \qquad (3)
\end{aligned}$$

We also define $\vec{T}$ to have unit length (all vectors are normalized). This gives us *Condition 2*:

$$1 = \vec{T} \bullet \vec{T}$$

Some algebra...

$$\begin{aligned}
1 &= \vec{T} \bullet \vec{T} \\
&= (\alpha\vec{I} + \beta\vec{N}) \bullet (\alpha\vec{I} + \beta\vec{N}) \\
&= \alpha^2(\vec{I} \bullet \vec{I}) + 2\alpha\beta(\vec{I} \bullet \vec{N}) + \beta^2(\vec{N} \bullet \vec{N}) \\
1 &= \alpha^2 - 2\alpha\beta C_i + \beta^2 \qquad (4)
\end{aligned}$$

Solving for α and β in (3) and (4), will give 4 pairs of answers. The pair that we want is:

$$\alpha = \eta_{it} \qquad \beta = \eta_{it} C_i - \sqrt{(1 + \eta_{it}^2(C_i^2 - 1))}$$

Therefore, the final transmission vector $\vec{T} = \alpha I + \beta N$ is:

$$T = \eta_{it}\vec{I} + (\eta_{it} C_i - \sqrt{(1 + \eta_{it}^2(C_i^2 - 1))})\vec{N}$$

where $\eta_{it} = \frac{\eta_i}{\eta_t}$ and $C_i = -\vec{N} \bullet \vec{I}$. However, this only holds **iff the root term is positive**! Otherwise there is *total internal reflection* and it is ignored.

One important implementation detail for refraction is that the ray tracer has to keep track of the order of materials that rays are transmitting from and to. In other words, it needs to know which is the incident material (η_i), and which is the transmission material (η_t). This might be done by keeping track of the incidence material index within the ray trace call.

8 Setting Up the Camera and Screen

A basic way to set up the screen (pixel array) is as follows.

1. Place the eye (camera) at a location in the scene. This is the eye ray origin R_o, an XYZ coordinate. It will be used as the origin of the initial ray.

2. Now define the coordinates of two opposite corners of the screen (pixel window). They should be an equidistance from the eye. Of course, the eye and the window should be appropriately placed so that the models defining the scene are suitably viewed.

3. It is now an easy matter to compute the coordinates of all the pixels, based on the desired resolution of the final image.

Although the above is easy to set up, it is not the simplest approach for setting up an arbitrary camera within a complex scene. Not only must you make sure that your eye and screen are viewing the scene, but you need to make sure that the screen itself is not distorted. This can happen if the screen corners are not equidistant from R_o. Also, be aware that if the eye is too close to the screen, extreme perspective distortion may occur.

A more sophisticated way to set up the camera and screen is given in [3]. Although much more complex than the basic approach given above, it is more flexible for setting up cameras in scenes. It is also not prone to screen distortions that can happen with the basic setup, which easily occurs when the screen corners and eye coordinates are misaligned.

9 Implementation Advice

The following pointers should be helpful...

1. Manually test your intersection computations with pre-made scenes. Render black-and-white mask images to show which pixels contain intersections (hit points) and which do not.

2. Normalize all your vectors! If you forget to do this, virtually every aspect of the ray tracer will fail: intersections, local lighting, reflection, refraction.

3. Implement your ray tracer incrementally. Get local lighting to work first. Then add shadows. Then add reflection. Then refraction.

4. To speed up the ray tracer, pre-compute all constant expressions, for example, η_i/η_t. These expressions are constant for the entire run of the ray tracer, and should not have to be re-evaluated on a pixel-by-pixel basis.

5. A common bug is to see black splotches or dots on your final rendered image. The reason for this is as follows. When computing a new (recursive) ray to trace, don't count intersections that are a small distance from ray

source (eg. $< |error|$). Otherwise, you will get "stuck" at the ray sources, which create dark dots. This point is relevant at all points of recursion, both reflection and refraction. One way to deal with this is by keeping track of the object where the ray origin is on. If the intersection is on the same object, then ignore it. (This presumes all objects are either convex geometric objects, or polygons.)

6. Again, refraction won't work if the root term is negative: ignore it.

7. Research the literature for fast intersection algorithms. Shaving off a few multiplications for a sphere intersection test can mean saving minutes off the overall rendering!

8. The most drastic optimizations will happen if you introduce intersection optimization techniques. There are many ways in which intersections can be optimized; please check the ray tracing literature such as [2] and elsewhere. For example, complex polygonal models can be wrapped in bounding spheres or bounding boxes. These bounding spheres are first used as a preliminary intersection test (spheres are fast to process). A ray must first intersects an object's bounding sphere, before the actual object geometry is used to find the real intersection point (if it exists). Hierarchies of bounding spheres can be used for very complex models.

   Even more dramatic accelerations in ray tracing large scenes will occur with spatial partitioning techniques.

9. It is not difficult to use an algorithmic (procedural) texture on materials. Associate a texture equation for a material. This equation will use the hit point location, and possibly the hit point normal, to compute a local RGB colour.

10 Conclusion

Ray tracing is an established technique for high-quality 3D rendering. It is also used in many commercial and open-source graphics systems. The major animation systems all use ray tracers for final animation rendering. Furthermore, ray tracing is here to stay. As hardware such as graphics processing units (GPU) continues to become faster and cheaper, ray tracers are now entering the domain of possibility of being real-time rendering engines. Developments in this area will be exciting in the coming years.

References

[1] T. Funkhouser. Ray casting (cs426 lectures). `http://www.cs.princeton.edu/courses/archive/fall2000/cs426/`, 2000. Last accessed January 10, 2014.

[2] A.S. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989.

[3] F.S. Hill Jr and S.M. Kelley. *Computer Graphics using OpenGL, 3e*. Prentice Hall, 2007.

[4] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. ACM Press, 1992.

[5] Wikipedia. Ray tracing. `http://en.wikipedia.org/wiki/Ray_tracing_(graphics)`, 2014. Last accessed January 10, 2014.