

Global Illumination Using Ray Tracing

K. M. Patel

(Computer Engineering, School of Engineering/RK University, India)

ABSTRACT

Hardware like rasterization provides interactive frame rates for rendering dynamic scenes, but lot of abilities of ray tracing required for efficient global illumination simulation. Existing ray tracing based methods yield high quality renderings but are far too slow for interactive use. We present a global illumination algorithm and C# OOP based programs that perfectly scales, has minimal preprocessing and communication overhead, applies highly efficient sampling techniques and benefits from shooting coherent groups of rays. Thus a performance is achieved that allows for applying arbitrary changes to the scene, while simulating global illumination including shadows from area light sources, specular effects, and caustics at interactive frame rates. Ceasing interaction rapidly provides high quality renderings.

Keywords: Illumination; Ray Tracing; Refraction; Reflection, Shadow; Texture;

I. INTRODUCTION

Overview

In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scan-line rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television special effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and chromatic aberration.

An image with the following characteristics can be considered as realistic image:

- Light effects (One or more)
- Shadowing
- Refraction of lights
- Refraction
- Specular Reflection

By applying the concepts of local illumination and global illumination we can produce photo-realistic image. Illumination refers interaction of light with surface points to determine their final color and brightness

The governing principles for computing the illumination

- Light attributes (light intensity, color, position, direction, etc.)

- Object surface attributes (color, reflectivity, transparency, etc)
- Interaction among lights and objects (object orientation)

Interaction between objects and eye (viewing dir.)

Global Illumination:

It is a method (algorithm) of computation for light calculation in the scene which, takes in to account the light bounces from the neighboring surfaces, along with the normal illumination of direct lights. In Other words GI calculates the Indirect light also, thus it makes the renders more photo-realistic. Examples of GI methods are Radiosity and Ambient Occlusion in Blender and on a general scale Radiosity, Ray tracing and Caustics all use different GI algorithms.

Incorporating global illumination is important step towards realism in computer graphics. There are many areas where graphics realism is high priority. In this thesis, recursive raytracing only supports some of basic primitive objects (plane, cube and sphere).

Basics of Ray Tracing

Camera is defined by its Position in the Scene (a 3D Vector), a point to LookAt (the purple arrow) which points at the center of the Viewport, and the tilt of the Camera (the blue arrow) called Top (it usually points strait up).

- The *Light* is defined by its Position in the scene and the *Color* of the light denoted by the light bulb.

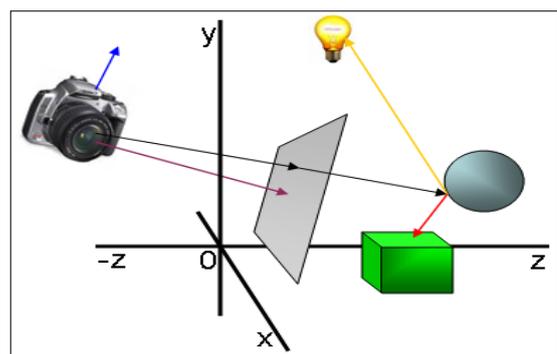


Fig-1 : Ray Tracing Example

- The Viewport is derived from the Camera settings and is defined by the LookAt point of the Camera and a fixed size of (-3,-3)-(3,3).
- A *Ray* is defined by a starting Position, and a *Direction* in which the Ray is casted.

The Background is defined by a Color that will be displayed if it is not covered by any other shape.

In a typical raytracing setting a Ray is casted through each pixel in the viewport into the scene, in this example the

black arrow. The raytracer will try and find out if the ray is intersecting with any object/shape in the scene. In this example it will intersect with the Sphere. Otherwise it will simply display the background color. To determine the color to display for the pixel, a number of techniques can be used and mixed referred as shading effects.

Shading effects and Color

Because ray tracing scenes require usually a high precision of calculations, so the R, G and B components are scaled down to a floating point number between 0 and 1. Also some of the common arithmetic operators have been overridden, so it will be easier to add, multiply and blend Colors.

The most basic technique is by simply displaying the intrinsic color of the Sphere itself. This is called Ambient lighting. Ambient light is the so called background light that will light up all objects in the scene slightly (see figure a).

The color is also influenced by the amount of light emitted by surrounding other light sources. In this case the light bulb will light up the surface of the sphere depending on how well the surface is exposed to the light. The yellow arrow shows the direction in which the light is traced back to its source. Based on this direction, and the direction the surface of the sphere is facing, the amount of light is calculated. This is called Diffuse light. It gives a nice shading effect (see figure b).

Additionally the effects can be enhanced by introducing Highlights, if the surface is somewhat reflective and the rays from the light source are reflected on the shape's surface straight into the camera, a highlight appears: usually a very shiny and bright color (see figure c).

Now for even more effects we can add Reflection and Refraction. In the case of Reflection, the Ray casted from the Camera is reflected on the surface of the sphere onto the green box denoted by the red arrow. This means the particular pixel the Ray travels through will light up with a somewhat greenish color also: the box is reflected into the sphere (see figure e).

Refraction is somewhat more complicated. Refraction is the effect of a ray bending when traveling through a different Material. This applies to transparent objects/shapes. An example of this is a glass ball, where the light rays are bent when traveling through the ball.

Another type of effect added to the scene is Shadows. Shadows do not add color to a pixel, but instead reduce the amount of Color. To find out if an intersection with an object is in a shadow of another object, simply trace the path back to the light source from intersection point (yellow arrow) and find out if any object is blocking it (does it intersect with any other object than the light source?). If it is blocked, simply reduce the amount of light by a factor.

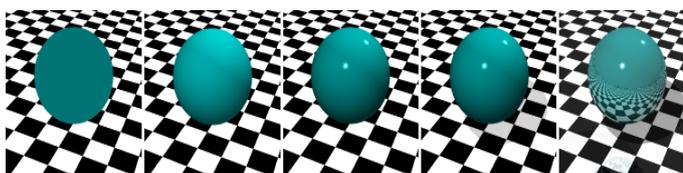


Fig-2 : Shading effects: a) Ambient, b) Diffuse, c) Highlights, d) Shadows and e) Reflection (notice the reflection on the floor also)

II. STEPS TO DETERMINE THE SHADING EFFECT:

The following steps to be used for evaluating shading effects for different cases:

- put the mathematical equation for the ray into the equation for the object and determine if there is a real solution.
- If there is a real solution then there is an intersection (hit) and we must return the closest point of intersection and the normal (**N**) at the intersection point
- For a shadow ray we must return whether any ray-object intersection is closer than the ray-light intersection
- For a ray tested against a boundary volume, we just return a simple hit or no hit
- For texture mapping we need the intersection point relative to some reference frame for the surface

Ambient Light:

- Each light source has an ambient light contribution (I_a)
- Different objects can reflect different amounts of ambient (different ambient reflection coefficient K_a , ($0 \leq K_a \leq 1$))
- So the amount of ambient light that can be seen from an object is:

$$Ambient = I_a \times K_a$$

Diffuse Light:

The illumination that a surface receives from a light source and reflects equally in all direction. It does not matter where the eye is because it distributes the light in all direction equally.

Lambert's law: the radiant energy D that a small surface patch receives from a light source is:

$$D = I \times \cos(\theta)$$

Specular:

It appears as bright spot on the object as shown in figure,

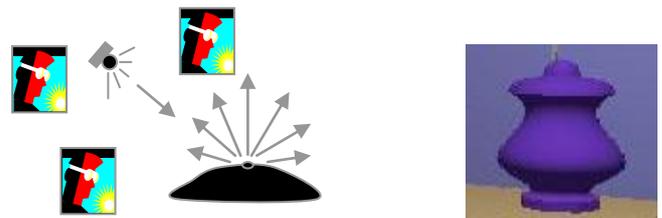


Fig-3 : Diffuse Light Effect

the result of total reflection of the incident light in a concentrate region. How much reflection you can see that depends on position of an observer.

$$specular = K_s \times I \times \cos(\theta)$$

Where,

K_s : specular reflection coefficient

N : surface normal at P

I : light intensity

θ : angle between V and R

L : Light Source

Reflection:

Mirror like reflections are made by calling both the intersection and illumination routines as part of the illumination calculation. We create a reflected ray in the correct direction and call both the intersection routine and the illumination routine for that ray. To get what can be seen

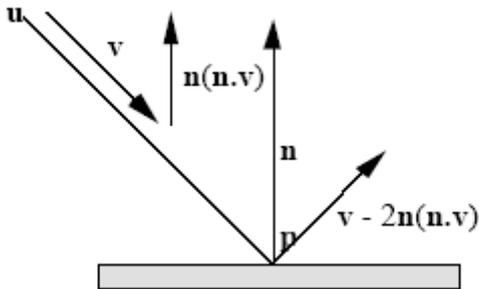


Fig-4: Reflection

in that direction from this surface. Having got back the colour of this ray, our illumination routine has to include it in the total colour calculation for the surface. We introduce another coefficient, k_s , which is multiplied by the brightness of the reflected ray and added to the total for the current surface. The original and reflected rays make equal angles with the surface normal. The direction of the reflected ray can be found easily using vectors. Assuming that the normal is represented by a unit vector, n , the component of the ray direction, v , in the direction of the normal is:

$$(v \cdot n) * n$$

Reflection has the effect of reversing this component without changing the component parallel to the surface. So the reflected ray is:

$$p + (v - 2(v \cdot n) n)t$$

Refraction

Water like transparent refraction made by calling both the intersection and illumination routines as part of the illumination calculation. We create a refracted ray in the correct direction and call both the intersection routine and the illumination routine for that ray that determine by the following equation:

$$T = [\frac{n \cdot I}{n \cdot N} - (1 - \frac{n \cdot I}{n \cdot N})^2]^{0.5} * N - \frac{n \cdot I}{n \cdot N} * N$$

III..TEXTURE

One of the important factor to get more realism is texture. To make any scene look even more realistic one must be able to add textures to any shapes. Basically texture can be compared to a piece of gift wrapper, which is wrapped around the object. There are two types of texture materials: a texture material based on a colormap or image and a texture material that is calculated (e.g. the chessboard effect).

Textures are flat and therefore require two coordinates to determine the color to display: often the u and v notation is used. The (u,v) coordinates are mapped onto $(-1,-1)$ to $(1,1)$ and from there on the color is either read from the colormap, or calculated respectively. The difficulty lies in calculating

the (u,v) coordinates from an intersection point with the shape. Depending on the shape, the (u,v) coordinates need to be calculated in different ways, but this is up to that programmer to implement.

IV.ANTI-ALIASING

One other important feature to have in a Raytracer is the ability to cope with anti-aliasing. Anti-aliasing is a technique to soften huge color differences between neighbouring pixels, so it will look more soothing for the eye. Several techniques can be used to counter this aliasing effect. A quick but dirty technique is to simply apply a "mean filter". The pixel will get the mean color value of neighbouring pixels. This is implemented as the 'Quick' AntiAliasing method in this thesis.

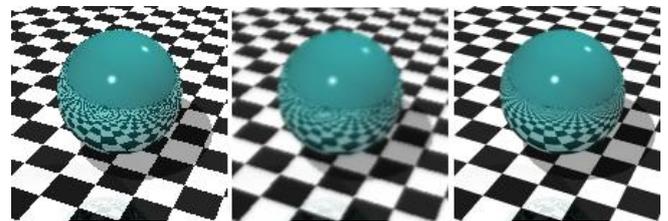


Fig-5: AntiAliasing methods: a) None, b) Mean filter, c) Monte Carlo sampling

This results is a smoothed image, however the image may also appear a bit vague/blurry. A much nicer way of anti-aliasing is using the 'Monte-carlo' method. The idea here is instead of casting a single ray into the scene through a pixel on the viewport, instead we cast multiple rays through a single pixel, scanning the neighbourhood and taking the average color of those. Although the method is slower, since we are now casting multiple rays for a single pixel, the accuracy is much better, resulting in much smoother but sharp anti-aliased images as shown in the figure below.

V. SHAPES

Have you ever wondered why in every raytraced image you always see a lot of spheres? Well apart from the nice shading effects on a sphere, more importantly, the intersection of a ray with a sphere can be calculated very fast. This is probably the most important aspect of a shape definition: how easy is it to calculate the intersection with the shape. Secondary to that, how easy is it to calculate its surface normal vector.

Calculating the intersection of a ray with arbitrary shapes turns out to be rather difficult. Instead different methods have been invented such as Voxel techniques or Marching cubes in order to determine the intersection points.

This raytracer however has not been optimized much for performance, and therefore only supports a limited set of shapes: Plane, Sphere and a Cube.

Scene contains Box (texture), Sphere (texture, reflective material) and plane (texture) with shadow and reflection effect

Ray-Sphere Intersection

A ray is defined by: $R(t) = R_0 + t * Rd$, $t > 0$

$R_0 = [X_0, Y_0, Z_0]$ = Position of ray

$Rd = [X_d, Y_d, Z_d]$ = Direction of ray

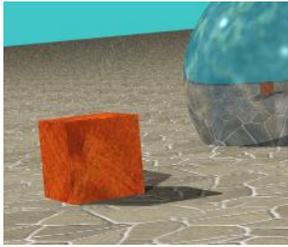


Fig-6 : Scene Example

A sphere can be defined by its center and radius with
 $S_c = [X_c, Y_c, Z_c]$

So, a sphere of radius S_r is:
 $S =$ the set of points $[X_s, Y_s, Z_s]$,
 where $(X_s - X_c)^2 + (Y_s - Y_c)^2 + (Z_s - Z_c)^2 = S_r^2$

To solve algebraically, substitute the ray equation into sphere equation and solve for t .

For a ray:
 $X = X_0 + X_d * t$
 $Y = Y_0 + Y_d * t$
 $Z = Z_0 + Z_d * t$

putting X, Y, Z into the sphere equation for X_s, Y_s, Z_s
 $(X_0 + X_d * t - X_c)^2 + (Y_0 + Y_d * t - Y_c)^2 + (Z_0 + Z_d * t - Z_c)^2 = S_r^2$

OR

$$A*t^2 + B*t + C = 0 \text{ (Quadratic Form)}$$

Where:
 $A = X_d^2 + Y_d^2 + Z_d^2$
 $B = 2 * (X_d * (X_0 - X_c) + Y_d * (Y_0 - Y_c) + Z_d * (Z_0 - Z_c))$
 $C = (X_0 - X_c)^2 + (Y_0 - Y_c)^2 + (Z_0 - Z_c)^2 - S_r^2$

Note: If $|Rd| = 1$ (normalized), then $A = 1$. So we can compute Sr^2 once.

So with $A = 1$, the solution of the quadratic equation is

$$t_0, t_1 = (-B \pm \sqrt{B^2 - 4*C}) / 2$$

where t_0 is for (-) and t_1 is for (+)

If the discriminant is < 0.0 then there is no real root and no intersection. If there is a real root ($Disc. \geq 0.0$) then the smaller positive root is the closest intersection point. So we can just compute t_0 and if it is positive, then we are done, else compute t_1 . The intersection point is:

$$R_i = [x_i, y_i, z_i] = [x_0 + x_d * t_i, y_0 + y_d * t_i, z_0 + z_d * t_i]$$

Unit Normal at surface is
 $SN = [(x_i - x_c) / S_r, (y_i - y_c) / S_r, (z_i - z_c) / S_r]$

VI. RAY TRACING ALGORITHM

```
RAYTRACE( ray )
{
    find closest intersection
    cast shadow ray, calculate colour_local
    colour_reflect = RAYTRACE( reflected_ray )
}
```

```
colour_refract = RAYTRACE( refracted_ray )
colour = k1*colour_local + k2*colour_reflect
          + k3*colour_refract
return( colour )
}
```

Limitations of ray tracing

The underlying idea of ray tracing is imitating what light rays do. But the real behaviour of light is rather complicated. We can cast a ray at a point light source to find shadows, but real light sources are not points. As a result, shadows are soft rather than sharp. There is no known cheap way to get realistic shadows from a ray tracer.

VII. IMPLEMENTATION DETAILS & RESULTS

In this work, module (Global Illumination using Ray Tracing) have been implemented as given below:

Language : C# (C-Sharp) .NET Framework v2.0

Configuration:

To run this module, there is no need to configure any additional library. But .NET 2005 must be installed on the system. Open the project in .NET, compile and run it. GUI of this module is as shown below:

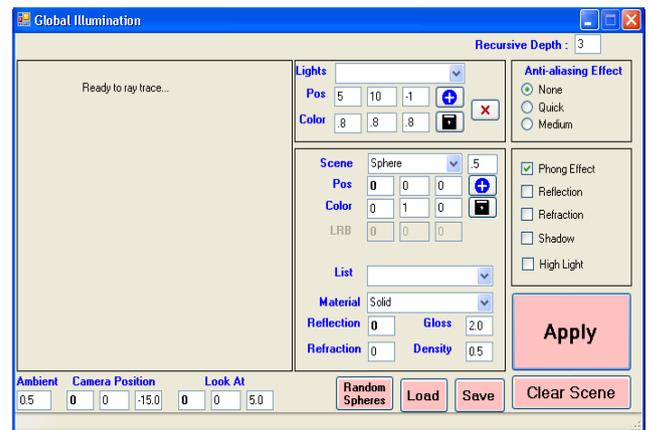
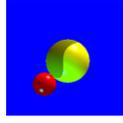


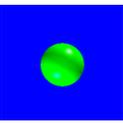
Fig-7: User Interface of Ray Tracing

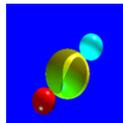
Experimental Results:

Output	Description
Light	: -Nil-
Object	: Sphere
Surface	: Solid
Type	: 0
Reflection Coefficient	: 0
Refraction Coefficient	: 0
Density	: 0
(Highlight) Gloss	: 0

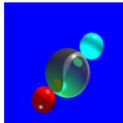
	Light	: 1 (10,5,-1)
	Object	: Sphere
	Surface	: Solid
	Type	
	Reflection Coefficient	: 0
	Reraction Coefficient	: 0
	Density :	: 0
(Highlight)	: 0	
Gloss :		

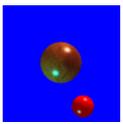
Shadow: 	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Sphere, Sphere
	Surface	: Solid, Solid
	Type	
	Reflection Const.	: 0, 0
	Reraction Const.	: 0, 0
	Density :	: 0, 0
(Highlight)	: 1.0, 1.0	
Gloss :		

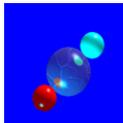
	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Sphere
	Surface	: Solid
	Type	
	Reflection Coefficient	: 0
	Reraction Coefficient	: 0
	Density :	: 0
(Highlight)	: 1.0	
Gloss :	: 2.0	

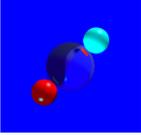
Two Shadows 	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Sphere, Sphere, Sphere
	Surface	: Solid, Solid, Solid
	Type	
	Reflection Const.	: 0, 0, 0
	Reraction Const.	: 0, 0, 0
	Density :	: 0, 0, 0
(Highlight)	: 1.0, 1.0, 1.0	
Gloss :		

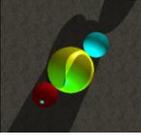
	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Sphere
	Surface	: Texture : Wooden
	Type	
	Reflection Coefficient	: 0
	Reraction Coefficient	: 0
	Density :	: 1.0
(Highlight)	: 1.0	
Gloss :	: 1.0	

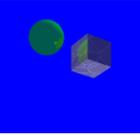
Reflection + Shadow: 	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Sphere, Sphere, Sphere
	Surface	: Solid, Solid, Solid
	Type	
	Reflection Const.	: 0.3, 0, 0
	Reraction Const.	: 0, 0, 0
	Density :	: 0, 0, 0
(Highlight)	: 1.0, 1.0, 1.0	
Gloss :		

	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Sphere, Sphere
	Surface	: Texture : Wooden, Solid
	Type	
	Reflection Const.	: 0, 0
	Reraction Const.	: 0, 0
	Density :	: 0, 0
(Highlight)	: 1.0, 1.0	
Gloss :		

Relection + Texture: 	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Sphere, Sphere, Sphere
	Surface	: Texture, Solid, Solid
	Type	
	Reflection Const.	: 0.3, 0, 0
	Reraction Const.	: 0, 0, 0
	Density :	: 0, 0, 0
(Highlight)	: 1.0, 1.0, 1.0	
Gloss :		

Refraction + Shadow: 	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Sphere, Sphere, Sphere
	Surface	: Solid, Solid, Solid
	Type	
	Reflection	: 0, 0, 0
Const.		
Reraction	: 1.0, 0, 0	
Const.		
Density :	: 0, 0,0	
(Highlight)	: 1.0, 1.0, 1.0	
Gloss :		

Plane + Shadow 	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Sphere, Sphere, Sphere, Plane
	Surface	
	Type	: Solid, Solid, Solid, Texture
	Reflection	
Const.	: 0, 0, 0, 0	
Reraction	: 0, 0, 0, 0	
Const.		
Density :	: 0, 0,0, 0	
(Highlight)		
Gloss :	: 1.0, 1.0, 1.0, 0	

Cube + Sphere + Reflection: 	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Cube, Sphere
	Surface	: Texture, Solid
	Type	
	Reflection	: 0.3, 0.3
Const.		
Reraction	: 0, 0	
Const.		
Density :	: 0.5, 0	
(Highlight)	: 0, 0	
Gloss :		

Random Spheres + Reflection : 	Light	:1: (10,5,-1), 2: (-10,-5,-5)
	Object	: Thirty Spheres
	Surface	: All are Solid
	Type	
	Reflection	: 0.2 each
Const.		
Reraction	: 0 each	
Const.		
Density :	: 0 each	
(Highlight)	: 2.0 each	
Gloss :		

VIII. CONCLUSION

The most important aspects of a point based rendering system are the use of a compact and concise data representation, and a fast and efficient rendering algorithm. Reducing the workload of the renderer is important and there are several techniques for doing so. Culling is also a popular technique. If used with a tree based data representation, complete sub-trees can be culled, or pruned. Backface culling, coupled with the use of normal cones, provides an effective means of reducing the number of points to be considered. The inclusion of frustum and occlusion culling techniques may increase the overheads of the rendering algorithm, and may provide little or no benefit when the whole object is on screen, but they can speed up the rendering time for complex scene. Point hierarchies, represented as trees, are used by the point based rendering system mentioned. Although the implementation of the trees and their meanings may differ slightly, their result is the same. They provide storage solution in a manner that allow traversal quickly. The best solution would be to implement a tree based structure to store the points with octree and store information at each node.

ACKNOWLEDGMENT

Author is thankful to the Department of Computer Engineering and Information Technology of R.K.College of Engineering & Technology, Kasturbadham, Rajkot, for providing infrastructure facilities during progress of the work. Also thankful to Prof. D G Thakor & Prof. P. B. Swadas, assistant professors, BVM College of Engineering, for giving his moral and technical support to make completion of this work successfully. Authors are grateful to everyone who contributed with data to make this work successful.

REFERENCES

- [1] [1] J. Arvo and D. Kirk. Fast Ray Tracing by Ray Classification, *Computer Graphics (Proceedings of SIGGRAPH 87)*, 21(4):55–64, 1987.
- [2] M. Cohen and J. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Cambridge, 1993.
- [3] E. Veach and L. Guibas. Metropolis Light Transport. In *Proceedings of SIGGRAPH 97*, Annual Conference Series, pages 65–76, 1997.
- [4] I. Wald, C. Benthin, and P. Slusallek. A simple and practical Method for Interactive Ray Tracing of Dynamic Scenes. Technical report, Saarland University, 2002.
- [5] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3), 2001.
- [6] G. Ward. Adaptive Shadow Testing for Ray Tracing. In *Photorealistic Rendering in Computer Graphics (Proceedings of the 2nd Eurographics Workshop on Rendering)*, pages 11–20. Springer, 1994.
- [7] G. Ward and P. Heckbert. Irradiance Gradients. In *3rd Eurographics Workshop on Rendering*, Bristol, United Kingdom, May 1992.
- [8] T. Whitted. An improved illumination model for shaded display. *CACM*, 23(6):343–349, June 1980.
- [9] *Computer Graphics using OpenGL* by F.S. Hill, Edition-2, ISBN: 0321535863
- [10] *Procedural Elements for Computer Graphics* by David F. Rogers, Edition-3, ISBN:0070473714.