Boston College
Computer Science Department

Senior Thesis 2002
John Weicher
Distributed 3D Raytracing
Prof. William Ames

Introduction

Since their advent, computers have been used to aid humans in tasks that would be too complex or too time consuming to do without them. However, as computers became more and more powerful, they also began to show potential usefulness in areas that before were completely beyond our ability at all. People were finding a use for computers in all areas of activity, and visual art was no exception. The idea arose that perhaps computers could be used to generate pictures that looked so real, a person would the obvious reason of it simply being too dark to see anything, but because it is these particles of light which make the very image our eyes see altogether!

The process of image formation within the eye is very simple. Every scene or environment in which we can see has light sources: things that actively emit light themselves, or things that "emit" light by reflecting it. When an object emits light, such their color and intensity. Finally an “eye-point” and direction are defined. This is the location from which the image will be generated. Finally, the raytracing algorithm is applied to the scene. Light rays are simulated leaving the active light sources (those which actively emit light), bouncing off the objects in the scene based on the surface normal of the objects at the points of contact, having their colors altered depending on what objects they come in contact with. Determining the color of light rays that eventually come in contact with the eye-point forms an image (Fig. 4).

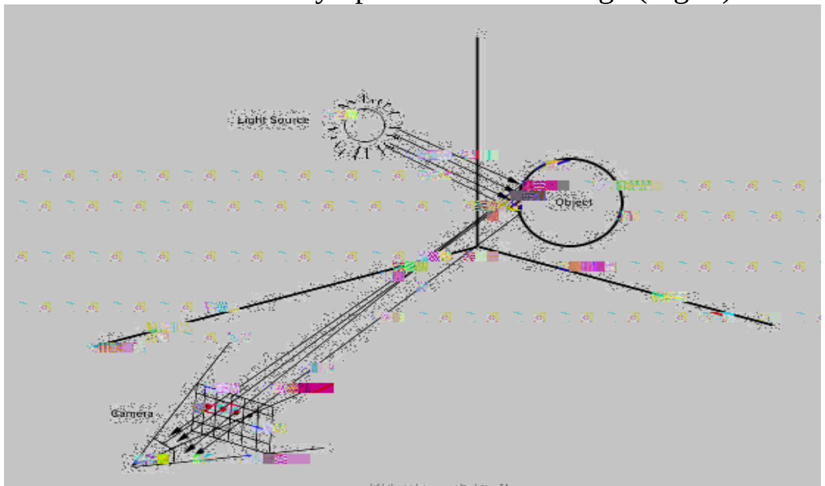


Fig. 4 Forward Raytracing

This process is more accurately called *forward raytracing*, as it models how light rays actually leave their source and travel forward in their journey until they either reach the eye-point or it is determined they never will. In *theory* this is an algorithm which

3. Implementation

Now that I have explained *what* backwards raytracing is, it is now appropriate to discuss my particular implementation of a backward raytracer. This will also serve to better explain how the raytracing process is actually achieved. I have chosen to implement my raytracer in the C++ language, as C is a very efficient procedural language. Although a language such as Java would have offered a much easier means of

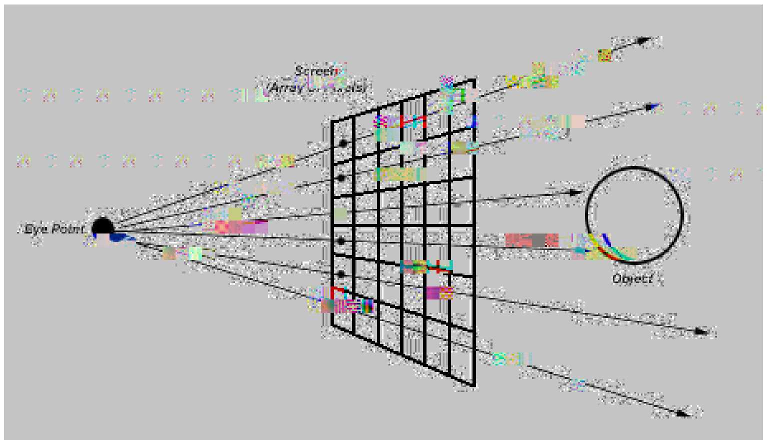


Fig.5. An example of using the pixel locations to isolate the desired rays that are of use to the image

Any

one intersection is calculated for a ray, it would need to be determined which intersection is “first” or closer to the eye. The algorithm would need to do this to ensure that the proper object’

- Set that pixel to the returned color

intersection point, all that is needed is the surface normal at that point, a vector pointing to the light source from that point, and the object's original color. Cos ! can be easily computed by taking the dot product of the two vectors after normalization. An example of this calculation is below:

Normalize both vectors:	$V_L = \text{sqrt}(LV_x^2 + LV_y^2 + LV_z^2)$
	$LV_{norm} = (LV_x/\lvert V_L\rvert, LV_y/\lvert V_L\rvert, LV_z/\lvert V_L\rvert,)$
	$V_{SN} = \text{sqrt}(SN_x^2 + SN_y^2 + SN_z^2)$
	$SN_{norm} = (SN_x/\lvert V_L\rvert, SN_y/\lvert V_L$

The computation for a reflected vector, much like that of Lambertian shading, is dependant only on the incoming ray (vector), and the surface normal vector of the object at the intersection point. The angle of the incoming vector relative to the surface normal of the object at that point is the same as that angle that the reflected vector will make with the surface normal at the same point (Fig. 8).

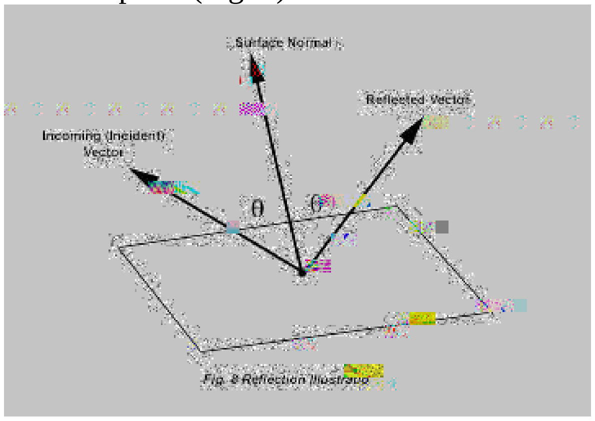


Fig. 8 Reflection Illustratio

A reflected vector can be computed using the formula:

$$V_{REF} = 2 * (V_{INC} \bullet V_{SN}) * V_{SN} - V_{INC}$$

The operation within the parenthesis is the dot product operation, just like in the computation for Lambertian Illumination. I will not go through an example of this computation within this paper, as it is a bit time consuming. It is sufficient to know however, that this calculation generates a reflected vector that can be used to generate the reflected ray of a mirrored object.

4. Additional Implementation Information

4.1 Matrix Transformations

One of the problems that I encountered when initially designing my implementation was the deriving of the equations that actually solve for the intersection between the objects and a line. This was often a very tedious and difficult task. For example, deriving the equation to solve for t even with the help of software such as Mathematica. Of course these derivations are not using the Composite Matrix of the object.

This process produces the same resulting intersections as would be computed using the original ray and the much more complicated equations derived for arbitrary versions of every object. By using this process of matrix transformations, I eliminated the need for doing these derivations. Matrices also allow for an easy means to stretch, rotate, and move objects in creative ways.

4.2 Anti-Aliasing

One of the problems with raytracing is the fact that the pixels of a computer are a finite size, and can only be set to one color. Because pixels are the smallest unit of color on a screen, it is impossible to set one half of a pixel to one color, and the second half to another. This causes problems because situations can arise (and usually do), in which if we could "zoom in" on a scene, we would notice places in the image where the edge of an object really only should cover part of a pixel. This usually occurs because pixels are often represented as a square. Therefore, trying to represent curved edges in particular usually results in an edge that looks "jagged." In Figures 10 and 11, we see how trying to represent a true circle with square pixels is impossible. Figure 10 represents the circle we would *like* to draw on the screen, but Figure 11 shows the "circle" we have to settle with

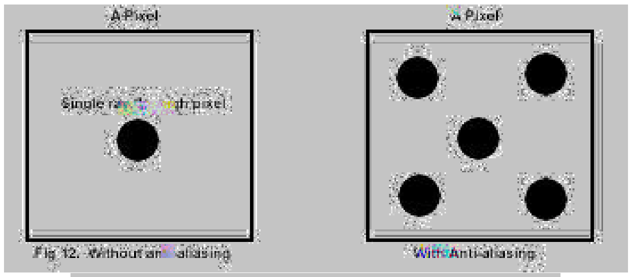


Fig 12. Without anti-aliasing With Anti-aliasing

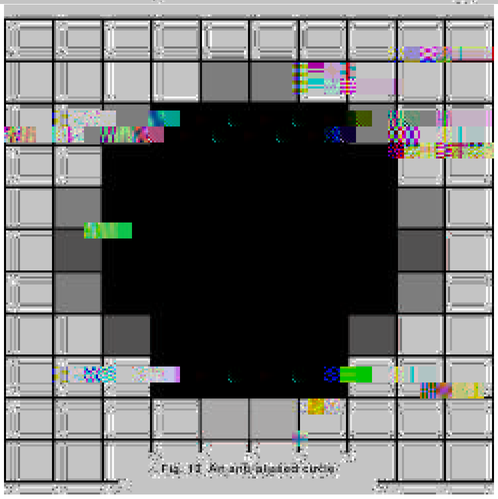


Fig. 13 An anti aliased circle

When viewing this circle at its normal size, and not enlarged to the pixel level as it is above for the sake of explanation, it would appear as a much more accurate circle. It should also be noted that while anti-aliasing makes an image look more realistic and servers to smooth edges, it obviously takes much longer. In the case of my implementation, there are five times as many rays fired, and so five times the number of calculations to perform per pixel. I have included other examples of anti-aliasing that has been applied to actual images generated by my raytracer in Appendix B of this paper.

4.3 Shadows

Another, very simple to implement component of raytracing is shadowing. To make an image more realistic, objects that are between a light source and other objects

The vast majority of objects in this world are not made up of just a single color.

occurred to me, too much of my application was already coded hard fast to the idea of a single light source.

These are the three primary improvements that I wish I could have made to my program. Obviously there are countless other *features* that I would have like to incorporate as well, such as the ability to crea

Bibliography

Angel, Edward, Interactive Computer Graphics: A Top-down Approach with OpenGL. (New York: Addison Wesley Longman, 2000.)

Glassner, Andrew S., Ed. An Introduction to Ray Tracing. (New York: Academic Press, Inc., 1989.)

Ma, Kwan-Liu, Painter, James S., Hansen, Charles D., Krogh, Michael F. "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering." ACM Computer Graphics. (SIGGRAPH Proceedings 1993). (New York: ACM Press, 1993.)

A special thanks to Prof. William Ames for all his time and help.

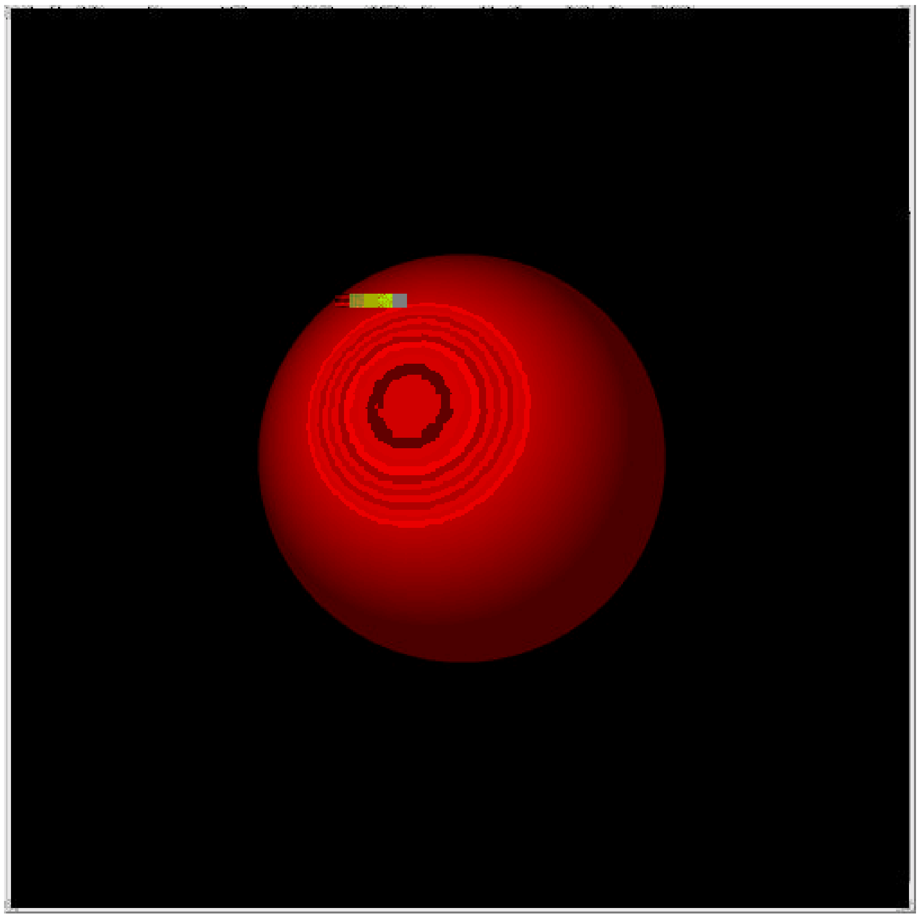
Appendix A – Scene File Format

Below is the format for a scene file:

```
[rayfile]

[GLOBALS]
attribute1 = value1
attribute2 = value2
…
attributen = valuen
[/GLOBALS]

[OBJECT]
type = value
attribute1 = value1
attribute2 = value2
…
attributen = valuen
[/OBJECT]

…

more object definitions

…

[/rayfile]
```

Appendix B – Samples Images

Below is an example image of the Lambertian shading model used on an illuminated sphere:



(App. B Con’t)

Appendix C – Transformation Matrices

Below are the matrices through which the various transformations can be applied to a point: Translation, Scaling, and Rotation in each of the three axes.

Translation

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

X-Axis Rotation

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-Axis Rotation

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-Axis Rotation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Appendix D – Source Code

Below is the listing of all source code:

Client:
Main.cpp
Main.h

Server:
Server.cpp
Server.h
Tracer.cpp
Tracer.h
Sphere.cpp
Sphere.h
Plane.cpp
Plane.h
Parser.cpp
Parser.h
Error.cpp
Error.h