A Prediction Model for Ray Tracing

by

Marc A. Lebovitz

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 21, 1999

Copyright 1998 Marc A. Lebovitz. All rights reserved.
The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author_____
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by_____
Seth Teller
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Prediction Model for Ray Tracing

by

Marc A. Lebovitz

Submitted to the

Department of Electrical Engineering and Computer Science

May 21, 1999

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis presents methods to predict ray traced rendering times given scenes and rendering options. In order to predict total rendering time, ray tracing is decomposed into a set of computation segments and basic operations. For each piece of the ray tracing algorithm, a time per call is determined for the function, as well as the number of calls during the desired segment of computation.

Predicting the recursive process is accomplished by modeling node generation as a branching process of a Markov chain for reflection and transmission rays. The results of the reflection/transmission branching process are then used as initial values in a shadow ray branching process of a Markov chain.

The prediction model was applied to a sample set of scenes, resulting in predicted rendering time errors ranging from 5.5% to 50.4%. The errors were a result of a variety of approximations necessary in the model. Error sources include object space bounding box error, screen space bounding box error, and approximating surface areas with volumes. Determining ray-object intersection probabilities was the greatest source for error as a result of empirically determined constants of proportionality that varied among scenes.

Thesis Supervisor: Seth Teller
Title: Associate Professor of Computer Science and Engineering

CONTENTS

ABSTRACT 2

1 INTRODUCTION 6

1.1 BACKGROUND 6
1.2 PREVIOUS WORK 7
1.3 A PREDICTION MODEL FOR RAY TRACING 7
1.3.1 Motivation 7
1.3.2 Overview of the Model 8

2 DIVIDING THE COMPUTATION 9

2.1 COMPUTATION SEGMENTS 9
2.2 BASIC OPERATIONS 12

3 USER INTERFACE 15

3.1 RAY TRACER 15
3.2 DATA VIEW 15
3.2.1 Goals 15
3.2.2 Views 16
3.2.2.1 Compare 16
3.2.2.2 By Cross-Reference 18
3.2.2.3 By Segment 19
3.2.2.4 By Operation 19
3.2.3 Search Data 20
3.2.4 Saving/Loading Data 21
3.2.5 Ray Tracing Options 22

4 COLLECTION 23

4.1 TIMING THE RAY TRACER 23
4.2 USES OF THE COLLECTION MATRIX 23

5 PREDICTION 25

5.1 PREDICTING THE TIME 25
5.2 PREDICTING THE COUNT 26
5.2.1 Basic Operations 26
5.2.2 Questions Raised 26
5.2.3 Eyeray Hits 26
5.2.4 Recursively Generated Rays 31

6 RECURSION MODEL 33

6.1 SHADOW TREES 33
6.1.1 Probabilities and Assumptions 33
6.1.2 The Recursive Branching Process 35
6.1.3 PMF 36
6.1.4 Expectations 37
6.2 REFLECTION/TRANSMISSION TREES 39
6.2.1 Probabilities and Assumptions 39
6.2.2 The Recursive Branching Process 41
6.2.3 PMF 43
6.2.4 Expectations 43
6.2.5 Depth 44
6.2.5.1 User-set Maximum depth and Minimum ray weight 44
6.2.5.2 Average Scene Weights 45
6.2.5.3 Finding a Minweight Bounded Depth 45
6.2.6 Relating the Branching Process to Our Questions 46

6.2.7 Passing the minimum weight requirement for rays ____ *48*
6.2.8 Assume rays pass minweight in pairs ____ *48*
6.2.9 Finding S_g for all generations g ____ *49*
6.2.10 Modeling branching depth more accurately ____ *54*
6.3 ASSUMPTIONS, STRENGTHS, AND WEAKNESSES OF THE MODEL ____ 57

7 RESULTS ____ 59

8 CONCLUSIONS ____ 68

9 APPENDICES ____ 69

9.1 BASETIME RECONSTRUCTION BY BASIC OPERATION ____ 69
9.2 COUNT RECONSTRUCTION BY BASIC OPERATION ____ 72
9.3 SCENES USED TO COLLECT DATA ____ 76
9.4 BASETIME RESULTS ____ 79
9.5 RECURSIVE RAY CAST DATA BY DEPTH ____ 80

10 BIBLIOGRAPHY ____ 86

Figures

FIGURE 1: HIERARCHICAL BREAKDOWN OF COMPUTATION TIME SEGMENTS 11
FIGURE 2: BASIC OPERATION CALLING TREE 14
FIGURE 3: DATA VIEW OF UI – COMPARE VIEW 17
FIGURE 4: DATA VIEW OF UI – CROSS-REFERENCE VIEW 18
FIGURE 5: DATA VIEW OF UI – SEGMENT VIEW 19
FIGURE 6: DATA VIEW OF UI – OPERATION VIEW 20
FIGURE 7: DATA VIEW OF UI – SEARCH VIEW 21
FIGURE 8: WORLD SPACE BOUNDING BOXES ARE TRANSFORMED TO EYE SPACE 28
FIGURE 9: SCREEN SPACE AND OBJECT SPACE BOUNDING BOXES 28
FIGURE 10: OVERLAPPING SCREEN SPACE BOUNDING BOXES ARE CLIPPED 30
FIGURE 11: RECURSIVE RAYS CAST ON RAY-OBJECT INTERSECTION 32
FIGURE 12: CLIPPED BOUNDING BOXES MAKE SURFACE AREA COMPUTATION DIFFICULT 34
FIGURE 13: A SAMPLE SHADOW RAY TREE 36
FIGURE 14: THE RECURSIVE BRANCHING PROCESS 43
FIGURE 15: BRANCH EXAMPLE ; G = R + T 50
FIGURE 16: EXAMPLE ILLUSTRATING S_G AND SINGLY-RECURSIVE TREES 52
FIGURE 17: PREDICTION ERROR OF TOTAL TIME SPENT RENDERING BY SCENE 59
FIGURE 18: PREDICTION ERROR OF *RAYCAST()* BASETIME BY SCENE 60
FIGURE 19: PREDICTION ERROR OF *SETPIXELIMMED()* BASETIME BY SCENE 61
FIGURE 20: PREDICTION ERROR OF EYERAY HITS BY SCENE 62
FIGURE 21: PREDICTION ERROR OF REFLECTION RAYS CAST BY SCENE 64
FIGURE 22: PREDICTION ERROR OF TRANSMISSION RAYS CAST BY SCENE 64
FIGURE 23: PREDICTION ERROR OF SHADOW RAYS CAST BY SCENE 65
FIGURE 24: PREDICTION ERROR OF REFLECTION RAY-OBJECT HITS BY SCENE 65
FIGURE 25: PREDICTION ERROR OF TRANSMISSION RAY-OBJECT HITS BY SCENE 66
FIGURE 26: PREDICTION ERROR OF REFLECTION RAY-OBJECT MISSES BY SCENE 66
FIGURE 27: PREDICTION ERROR OF TRANSMISSION RAY-OBJECT MISSES BY SCENE 67
FIGURE 28: SCENE 1 REFLECTION AND TRANSMISSION RAY CASTS 80
FIGURE 29: SCENE 2 REFLECTION AND TRANSMISSION RAY CASTS 81
FIGURE 30: SCENE 3 REFLECTION AND TRANSMISSION RAY CASTS 81
FIGURE 31: SCENE 4 REFLECTION AND TRANSMISSION RAY CASTS 82
FIGURE 32: SCENE 5 REFLECTION AND TRANSMISSION RAY CASTS 82
FIGURE 33: SCENE 6 REFLECTION AND TRANSMISSION RAY CASTS 83
FIGURE 34: SCENE 7 REFLECTION AND TRANSMISSION RAY CASTS 83
FIGURE 35: SCENE 8 REFLECTION AND TRANSMISSION RAY CASTS 84
FIGURE 36: SCENE 9 REFLECTION AND TRANSMISSION RAY CASTS 84
FIGURE 37: SCENE 10 REFLECTION AND TRANSMISSION RAY CASTS 85
FIGURE 38: SCENE 11 REFLECTION AND TRANSMISSION RAY CASTS 85

1 Introduction

To introduce the topic of ray tracing prediction, we will first provide some background on ray tracing in general. Then we will cite some previous work done on the subject of ray tracing prediction and give motivation for our study. Finally, we will present an overview of the model we use to predict the time to ray trace a given scene.

1.1 Background

Ray tracing is one of the most popular techniques for rendering a 2D image from a 3D scene. Based on the physics of light, it can accurately model reflections, refraction, soft shadows, caustics, and a host of other effects. The most basic ray tracing algorithm is computationally intensive and each of these effects adds even more to the task of rendering.

In order to improve rendering time, many acceleration techniques have been developed. These techniques utilize data structures, numerical and statistical methods, and computational geometry, among others. Many of these techniques can be employed together, but there are also competing techniques of each type.[1] When building a ray tracer, the programmer must pick and choose which effects and optimizations will best suit the application to maximize image quality without taking too much time to render.

How long is too long? Because of its large rendering time for realistic scenes, ray tracing is often used to render batches of frames off-line. This method is usually chosen for rendering animation sequences where image quality is the highest concern. The scene is set up along with cameras, lighting, and animation. Then the ray tracer is started and renders overnight or over the course of days. It is never clear exactly how long it will take to render any given scene. Rendering time ranges literally from seconds to days depending on the complexity of the scene and ray tracer. The only information that provides a clue

[1] Arvo et. al., 203.

for how long it will take is the time the ray tracer took to render similar scenes. Also, ray tracers will often be benchmarked for comparison against each other.[2]

1.2 Previous Work

There has been substantial work done on giving theoretical orders of growth to acceleration algorithms for ray tracing. James Arvo and David Kirk bring together a broad range in *A Survey of Ray Tracing Acceleration Techniques.*[3] They compare and contrast the data structures and algorithms, discussing when each is appropriate to use, and drawing on a large body of work.

Fujimoto, Tanaka, and Iwata took actual time measurements of ray traced scenes with different acceleration techniques implemented[4]. The ray tracer was run on a VAX 11/750. They compared these times with estimates for a non-accelerated ray tracer.

Ray tracers as entire applications have been benchmarked against each other. Eric Haines is among the contributors to Ray Tracing News (he also happens to be the editor) who benchmarked a variety of ray tracers.[5]

1.3 A Prediction Model for Ray Tracing

1.3.1 Motivation

When a scene or animation is set to render off-line, artists have little more than a vague idea of how long it will take. What's more, they do not know how long the ray tracer is spending in each section of the computation. There could be a bottleneck in the ray-object intersection computation, or perhaps the calculations for caustic effects are taking longer than they are worth. Maybe tweaking aspects of the scene or ray tracer would drastically reduce rendering time without sacrificing image quality.

---

[2] Haines, http://www.acm.org/tog/resources/RTNews/html/rtnv3n1.html#art10.
[3] Arvo, et. al.
[4] Fujimoto, et. al,.

To prevent going into off-line rendering blindly, we have created a prediction model for ray tracing performance. The model is given information about the scene to be rendered as well as which rendering effects and acceleration techniques will be used. The model uses this information to provide an estimate on total rendering time, as well as estimates on the rendering time for each aspect of computation.

Users are able to use these estimates to determine what effects are worthwhile when weighing image quality against rendering time. Also, users are able to determine which ray tracer acceleration settings are most effective for the scene.

1.3.2 Overview of the Model

To predict where the time is being spent in the ray tracer, we will break up the algorithm into ten computation segments. We will also identify eighteen most time consuming operations out of the functions implementing the algorithm as those functions. We will predict the time taken by each of these operations in each of the computation segments, then add the times up to get the total time predicted for rendering.

To predict the time taken by each operation, we will predict the time taken per call to the operation, as well as the number of calls to the operation in that computation segment. Predicting the time per call, the baseTime, will be done for most operations by averaging baseTimes across a sample set of renderings. For a few operations whose baseTime varies greatly from scene to scene, we will average the time per call in 1000 iterations of calls.

To predict the number of calls to each segment-operation pair, we will first estimate the number of rays from the camera into the scene that intersect objects. We will also need to model the number of recursive rays generated. We will model the recursive ray generation as branching processes. This will tell us how many reflection, transmission, and shadow rays were cast, as well as which of those intersected objects in the scene.

[5] Haines, http://www.acm.org/tog/resources/RTNews/html/rtnv3n1.html#art10.

2 Dividing the Computation

Ray tracing as a whole encompasses a host of algorithms and operations. To be able to predict the time a scene will take to ray trace, we must break down our ray tracer into manageable chunks. We will break it down in two orthogonal ways, computation segments and basic operations. Each basic operation takes place in one or more computation segment.

2.1 Computation Segments

To predict where the time is being spent in the ray tracer, we will break up the algorithm into ten computation segments.

Import

Loads the appropriate files for the scene and environment and sets up pointers to all objects, lights, and cameras in the scene.

Build

Builds the associated data structures, e.g. octree, BSP tree, jitter coefficients.

ComputeEyeRay

Computes the direction of all rays from the camera through the image plane and into the scene.

QueryEyeRay

Queries object data structures to determine if a given ray from the camera through the image plane intersects an object.

ComputeShadowRay

Computes the direction of shadow rays from the surface of an object hit by a ray from the camera through the image plane.

QueryShadowRay

Queries object data structures to determine if a shadow ray arising from a camera ray hit intersects an object.

ShadingModel

Computes Phong radiance at a point on the surface of an object hit by a ray from the camera through the image plane. This does not include shadow or recursive computations.

Reflection

Computes radiance at a point on the surface of an object arising from all recursive reflection computation. This includes ray generation, data structure queries, and shading model computation for all recursive reflection rays.

Refraction

Computes radiance at a point on the surface of an object arising from all recursive reflection computation. This includes ray generation, data structure queries, and shading model computation for all recursive reflection rays.

Display

Displays pixels on the screen.

These computation segments can be related as follows:

$$T_{Total} = T_{Setup} + T_{Rendering}$$

$T_{Setup} = T_{Import} + T_{Build}$

$T_{Rendering} = T_{Frame} * (\#\ frames)$

$T_{Frame} = T_{Pixel} * (\#\ pixels) + T_{Display}$

$T_{Pixel} = T_{Sample} * (\#\ samples/pixel)$

$T_{Sample} = T_{EyeRay} + T_{Shade}$

$T_{EyeRay} = T_{ComputeEyeRay} + T_{QueryEyeRay}$

$T_{Shade} = T_{ShadowRay} + T_{ShadingModel} + T_{Reflection} + T_{Refraction}$

$T_{ShadowRay} = T_{ComputeShadowRay} + T_{QueryShadowRay}$

Figure 1 shows these relation as a hierarchical structure of computation segments. All computation takes place in one of the ten leaf nodes of the tree.

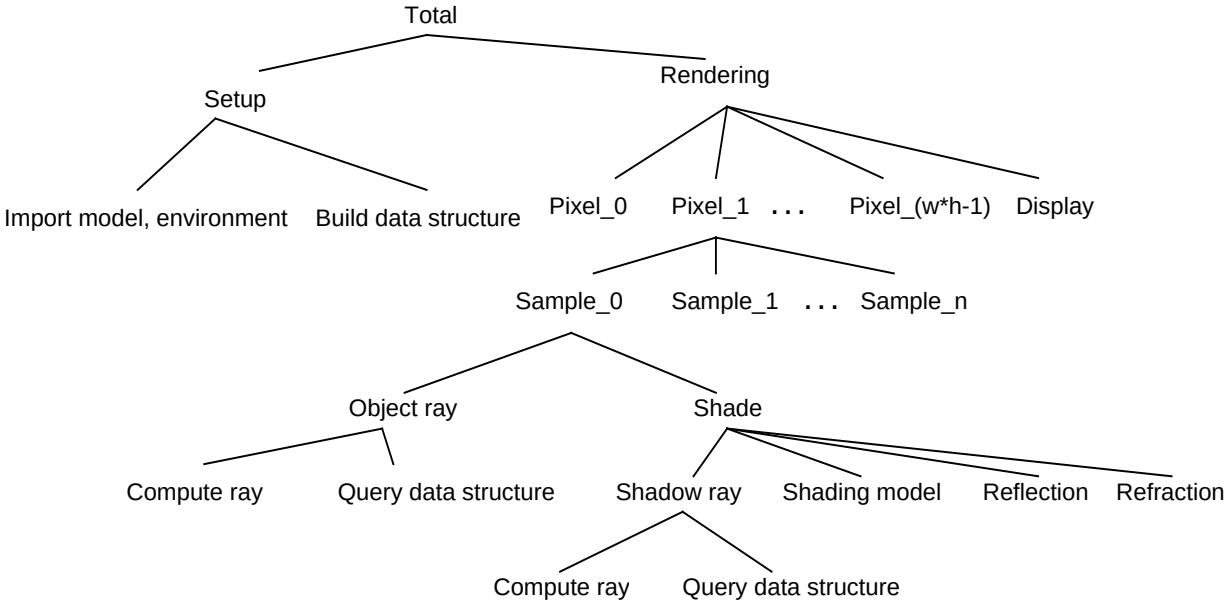

Figure 1: Hierarchical breakdown of computation time segments

It could be argued that recursive calls to the Phong model or shadow ray computation arising from reflection or refraction rays should be part of *ShadingModel* and *ComputeShadowRay/QueryShadowRay* respectively. We have chosen to include this computation instead with *Reflection* and *Refraction* computation. This is because a common application of the prediction model will be to decide whether or

not to include reflection or refraction effects in a rendering. The user would want to know how much these effects are costing in terms of total time. In addition, it is useful to know the breakdown of time spent by depth of recursion.

2.2 Basic Operations

The majority of the work in the ray tracer we have developed takes place in one of eighteen functions that we will refer to as the basic operations. When we reconstruct the time spent in ray tracing, all of it will be assumed to be in one of these operations. When we predict how much time a given scene will take to render, we will predict how many calls are made to each of these operations in each computation segment, as well as how long each call will take. What we will be left with is the ability to determine in what functions the ray tracer spends its time, as well as in what segments of computation. Figure 2 shows how the functions are called.

getObjectsAndLights

Parses scene files and extracts information on objects, lights, cameras, and the environment.

MakeJitter

Creates jitter coefficients for sample ray positions within a pixel.

ComputeEyeRay

Computes the direction of rays from the camera through the image plane.

Shade

. Computes the radiance at a ray-object intersection.

BackgroundMap

Computes the background radiance at a ray-object intersection.

ReflectionDirection

Given an incoming ray and an intersecting object, computes the direction of the reflection ray.

TransmissionDirection

Given an incoming ray and an intersecting object, computes the direction of the transmission ray.

ReflectionRadiance

Computes the radiance at a ray-object intersection due to reflection.

TransmissionRadiance

Computes the radiance at a ray-object intersection due to reflection.

Shadowing

Determines visibility of point by repeatedly casting rays towards a light until we are past the light, or we are occluded from the light.

AngularAttenuation

Calculates how much of the light coming from a spotlight (shining in its original direction) is still visible at the angle the light is at (relative to a ray-object intersection point).

DistanceAttenuation

Calculates how much of the light coming from a local light or a spotlight (shining in its original direction) is still visible at the distance the light is at from a ray-object intersection point.

SpecularRadiance

Collects the specular radiance emanating from the surface at the point we are shading.

DiffuseRadiance

Collects the diffuse radiance emanating from the surface at the point we are shading.

AmbientRadiance

Collects the ambient radiance emanating from the surface at the point we are shading.

EmissionRadiance

Collects the emissive radiance emanating from the surface at the point we are shading.

setPixelImmed

Displays a pixel on the screen.

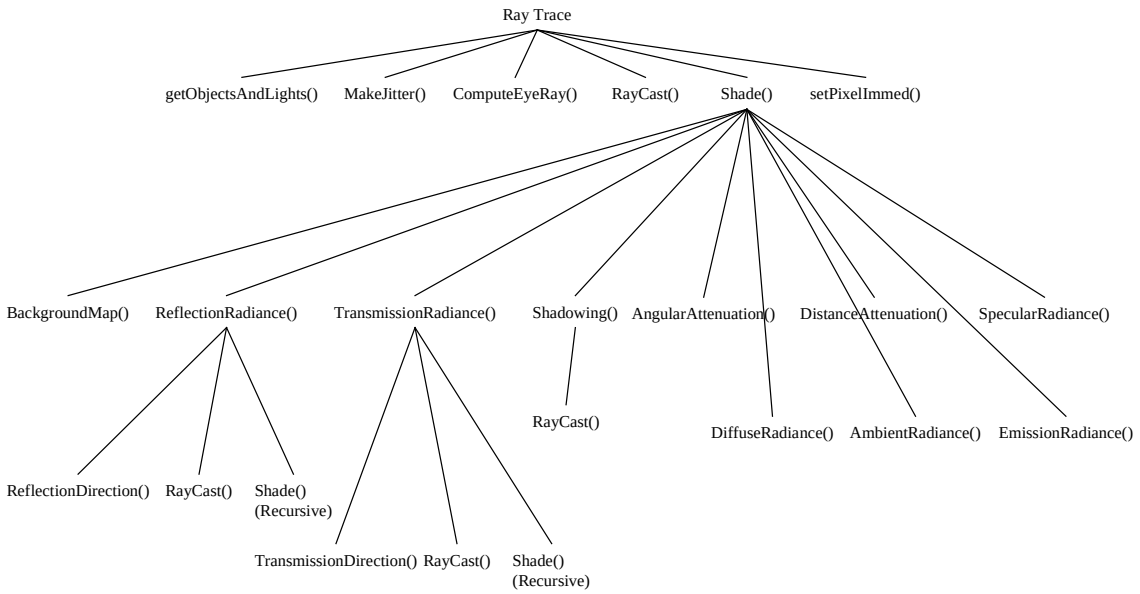

Figure 2: Basic operation calling tree

3 User Interface

The user interface to our ray tracing predictor consists of two separate windows, the ray tracer and the data view. The ray tracer allows the user to load scenes into a window where they can be viewed and manipulated. The data view provides the user with commands to make predictions and record data, as well as view the data in a variety of ways.

3.1 Ray Tracer

The ray tracer used was developed for 6.837: Introduction to Computer Graphics. It is based on the Open Inventor™ SoSceneViewer interface. Users can load and save scenes, manipulate the camera and lights, and scale objects. When the render button is depressed, the rendered pixels are displayed over the view of the scene as they are computed. The scene is rendered with the camera, lights, and object positions shown in the viewport.

3.2 Data View

3.2.1 Goals

The data view presents information on the collected and predicted data for the scene displayed in the ray tracer view. The goals of the data view are:

- To present collected data so that user can determine the time spent in each segment-operation pair, as well as the number of times the pair was called
- To present collected data so that user can relate the time spent in each segment-operation pair, as well as the number of times the pairs was called, to values for other pairs

- To present comparison of total collected data to measured time to determine accuracy of reconstructed time
- To present predicted data so that user can determine time spent in each segment-operation pair, as well as the number of times the pair was called
- To present predicted data so that user can relate time spent in each segment-operation pair, as well as the number of times the pairs was called, to values for other pairs.
- To present comparison of predicted data to collected data by segment-operation pair to determine accuracy of predictions
- To have the ability to search the predicted data for predictions on time or number of calls that are inside or outside a given error bound from collected data

3.2.2 Views

There are a number of different views the user can assign to the data view window, each allowing the user to discern unique information from the data. The different views are the compare view, the cross-reference view, the segment view, and the operation view.

3.2.2.1 Compare

The user can view the compare screen to compare the collected time to the measured time, as well as the comparing the predicted data to the collected data. On the left of Figure 3, the total time the scene actually took to render is displayed. Below that is a graph containing two percentage bars. The first compares total measured time to total reconstructed collected time. The second compares the total predicted time to the total collected time.

The colors of the bars indicate which of the two values is greater. For the first bar, a red color indicates the measured time is greater and so the bar represents the percentage of measured time the

collected time is. A green color indicates the collected time is bigger. For the second bar, a blue color indicates the collected time is greater, while a red color indicates the predicted time is greater.

On the right of Figure 3, the user can choose from among segments and operations that have either a non-zero time per call, a non-zero number of calls, or both, in either the collected or predicted data. Below, a graph displays the percent comparison of the time per call and number of calls for the segment-operation pair between the collected and predicted data. The color scheme is the same as for the second bar on the left side of the screen.

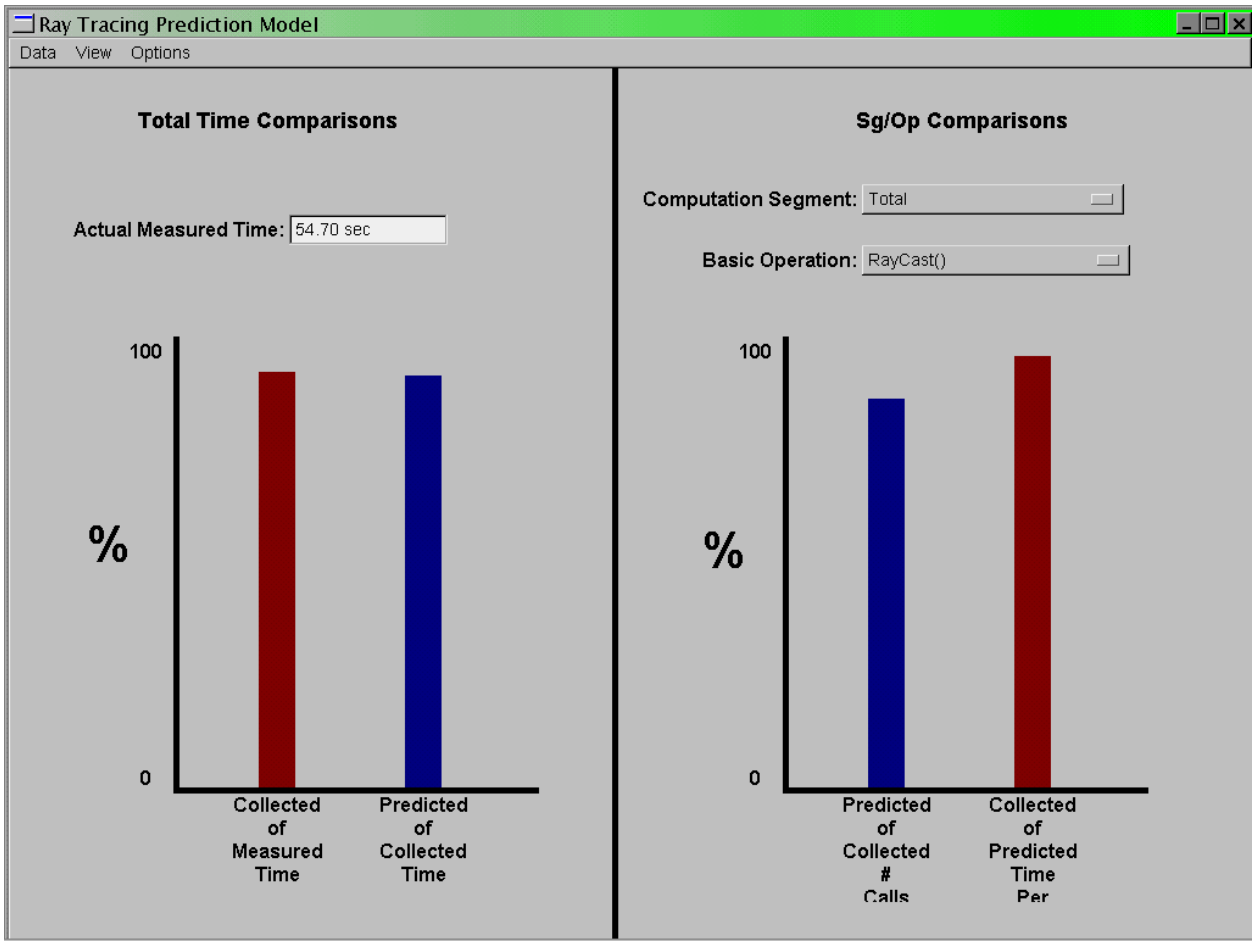

Figure 3: Data View of UI – Compare View

The compare view allows the user to compare predicted with collected data quickly and easily.

3.2.2.2 By Cross-Reference

The user can view the cross-reference screen to view either predicted or collected data on a chosen segment-operation pair. As seen in Figure 4, the total time is displayed, along with the time for the selected pair, the number of times the pair is called, and the average time spent for each call. Below, a graph presents the percentage of total time that the selected segment-operation pair takes.

The user can choose to view the predicted data, the collected data, or both together.

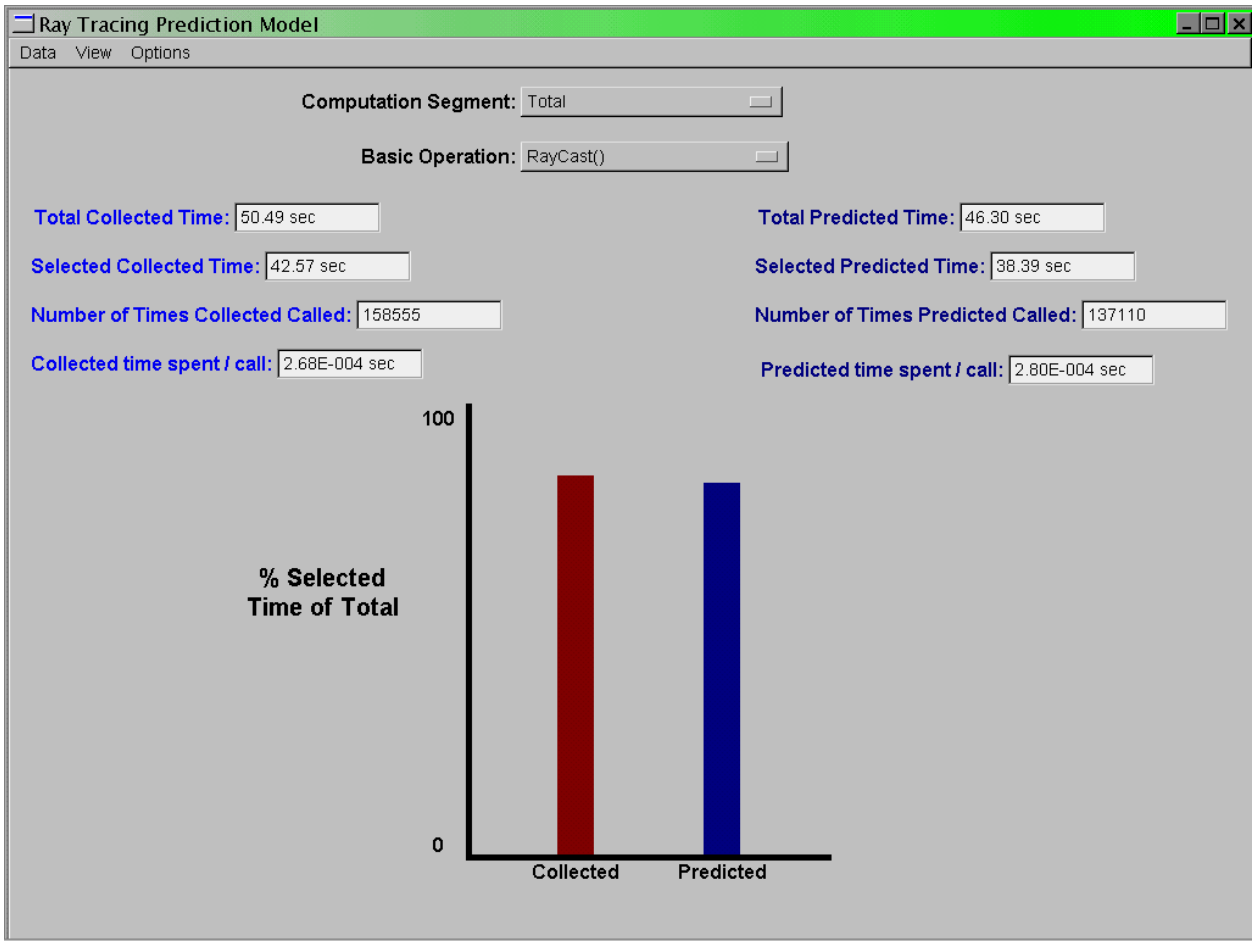

Figure 4: Data View of UI – Cross-Reference View

The cross-reference view allows the user to determine all the information available on each segment-operation pair for both the collected and predicted data.

3.2.2.3 By Segment

The user can view the segment screen to compare the percentage of total time spent among segments of computation in either the predicted or collected data. The operation is selected and the data is display in a sorted bar graph. Each bar in Figure 5 represents the percentage of total time that the selected operation takes in the given segment.

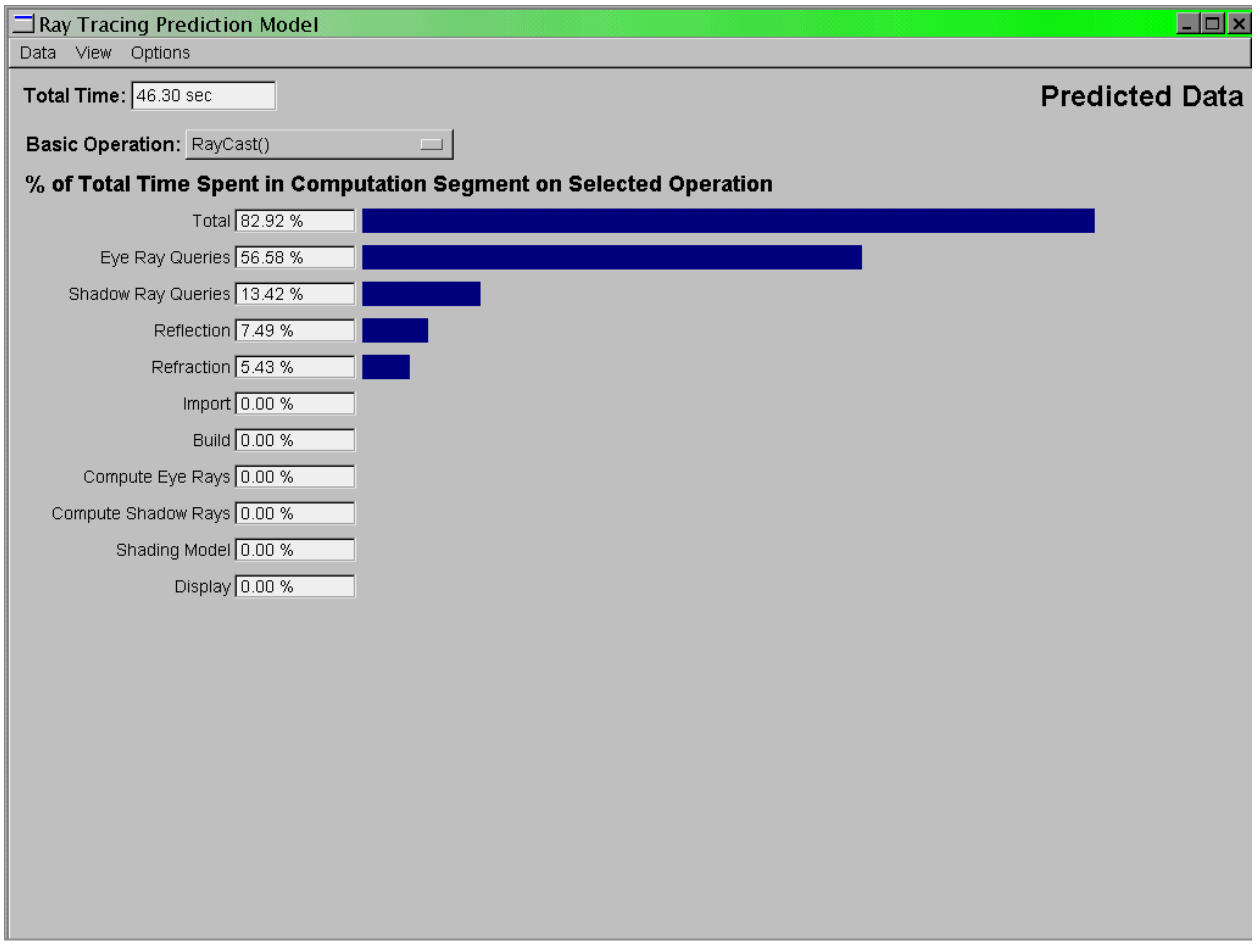

Figure 5: Data View of UI – Segment View

The segment view allows the user to determine in which segments for a given operation the majority of the rendering time was spent.

3.2.2.4 By Operation

The user can view the operation screen to compare the percentage of total time spent among basic operations in either the predicted or collected data. The computation segment is selected and the data is display in a sorted bar graph. Each bar in Figure 6 represents the percentage of total time that the given operation takes in the selected segment.

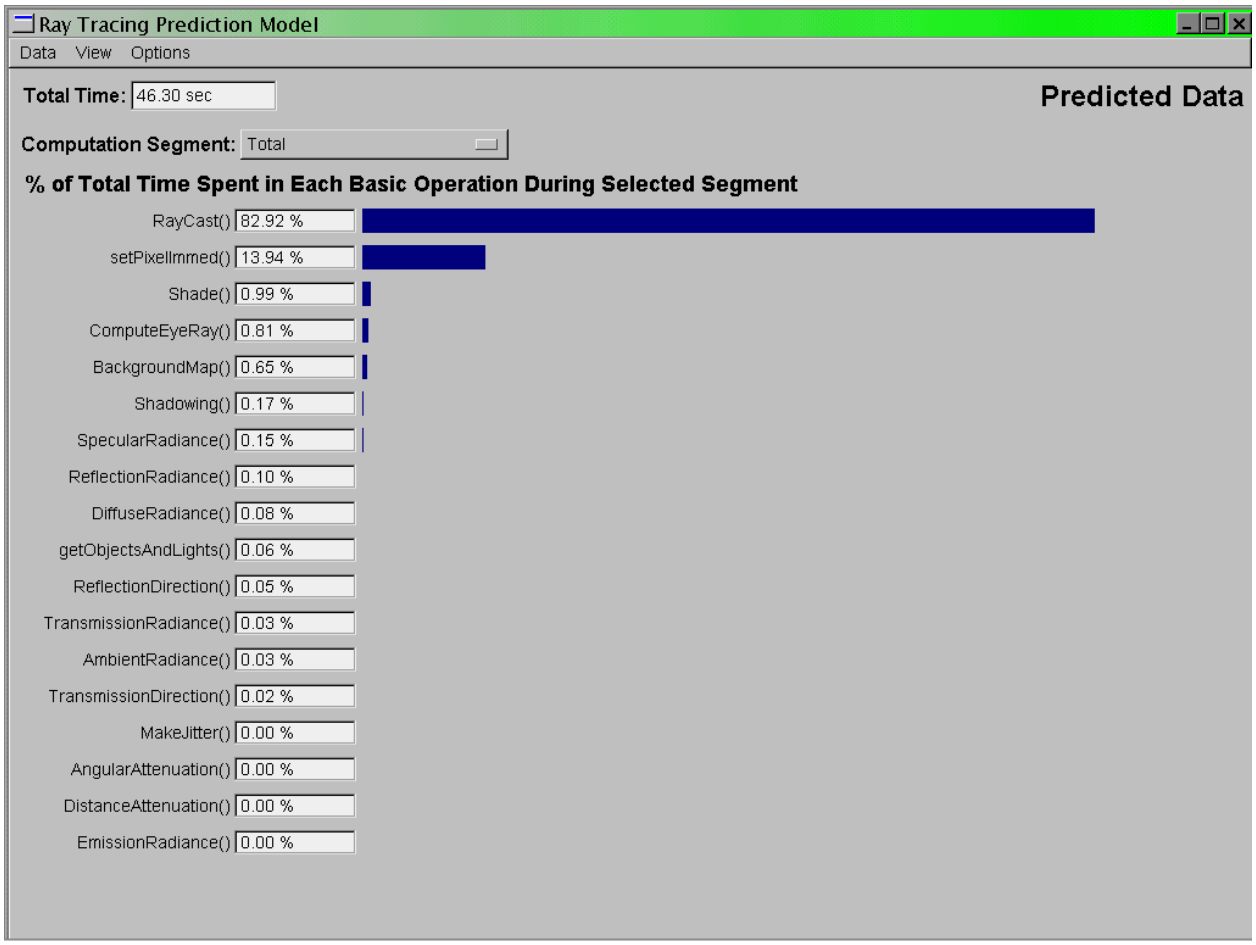

Figure 6: Data View of UI – Operation View

The operation view allows the user to determine in which operations for a given segment most of the rendering time was spent.

3.2.3 Search Data

The user can view the search screen to search the predicted data for segment-operations pairs that have a certain error characteristic. The scroll box in Figure 7 will list segment-operation pairs that are within or outside of a percent error of the corresponding collected data.

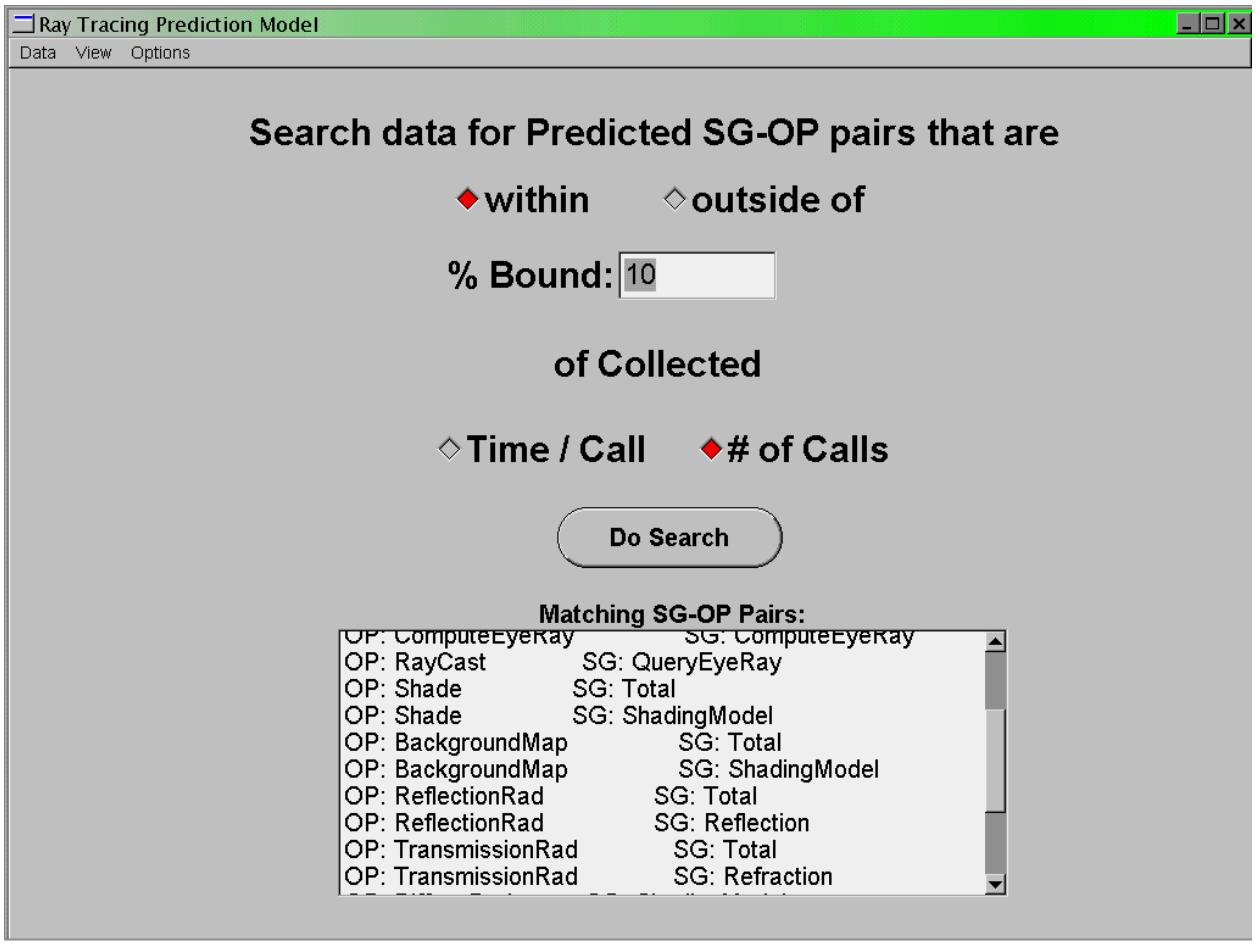

Figure 7: Data View of UI – Search View

The search view allows the user to quickly find segment-operation pairs of interest to investigate using the other views.

3.2.4 Saving/Loading Data

The collected and predicted data can be saved into a file and then reloaded into the UI at a later time. In addition, the data can be added to a database for post-processing. The database is a collection of

20 files, one for each basic operation. Within each file, the time per call and number of calls is listed for each segment in which the operation was called for each entry to the database.

3.2.5 Ray Tracing Options

Through the data view, the user can change a number of rendering options. Reflection and refraction can be toggled on and off. The maximum ray depth and minimum ray weight for recursive rays can be set. Shadows can be toggled on and off. The shading can be set to flat or Phong.

4 Collection

In order to determine the accuracy of our predictions, we must first develop a way to break the time measured to render a scene into computation segments and basic operations.

4.1 Timing the Ray Tracer

Each call to a timer to find the elapsed time is expensive enough that placing many of them in the code significantly alters the time taken to render a scene. As a result, we cannot time every basic operation individually. Instead, we increment counters every time a basic operation was called. Also, the total time taken by the ray tracer is recorded.

The counters are part of a collection matrix. Elements are accessed by computation segment and basic operation. Each time a basic operation is called in a given segment, the corresponding element in the matrix is incremented. In this way, we record the number of calls to each basic operation in each computation segment.

After the scene is rendered, we reconstruct the time spent in each operation-segment pair. Operations in a segment are called the recorded number of times with argument appropriate for the scene. The arguments are, whenever possible, exactly what were used for the rendering of the scene. The calls are timed together in these blocks and the time is also recorded in the collection matrix.

4.2 Uses of the Collection Matrix

The collection matrix is first tested for accuracy. The total reconstructed time is compared with the total measured time for renderings. Total reconstructed time is determined by summing time taken by each operation-segment pair.

Once the validity of the collection mechanism was established, we were able to use it to collect data on the average time taken for calls to each of the basic operations. These values are used in predicting the time any scene will take.

When predictions are made for a scene, the predicted number of calls and time per call for each operation-segment pair can be compared with the recorded counts and reconstructed times.

5 Prediction

The prediction for the time spent in a segment-operation pair is found by multiplying the time spent per call for the operation with the number of calls to the operation in that segment. We will now detail how to find those values.

5.1 Predicting the Time

In order to simplify prediction, all calls to a basic operation, regardless of computation segment, are assumed to take the same amount of time per call. We will refer to this value as the baseTime for that operation. The total time taken for a basic operation in a given segment is equal to the baseTime for that operation multiplied by the number of calls to the operation in that segment.

The baseTime for a given operation is not constant across scenes. Some argument values or user options change what the basic operations will do and how long they will take. Also, some operations have loops that will execute a certain number of times based on the scene being rendered.

The baseTime for each basic operation is determined by averaging the values from a group of rendered scenes. In the case where user options or argument values change baseTime values, one baseTime is recorded for each possibility. When later predicting the time spent for another scene, the appropriate baseTime is used. If more than one argument value may occur in the scene, the expected count for each case is determined and multiplied by the respective baseTime. The results are then added together.

For operations comprised of one large loop, a loopTime is recorded instead of a baseTime. When predicting the baseTime for other scenes, the loopTime is then multiplied by the expected number of loops to get the baseTime. No basic operation consists of more than one loop.

For certain operations, the baseTime varies greatly from scene to scene. For these operations, we do not average values across a sample set of rendered scenes. Instead, for a given scene, we execute the operations 1000 times using random arguments and find the average time per call.

Appendix section **9.1** details how the baseTime is reconstructed for each basic operation.

5.2 Predicting The Count

5.2.1 Basic Operations

We will look at each basic operation in turn and develop a prediction model for the number of times it is called in each computation segment. Appendix section **9.2** details how the number of calls in each computation segment is found for the eighteen basic operations.

5.2.2 Questions Raised

There are a number of questions about the scene that are raised when trying to predict the number of calls to the basic operations. How many of the initial rays from the camera into the scene hit objects? How many reflection rays are recursively generated? Of those, how many hit objects in the scene? How many transmission rays are recursively generated? Of those, how many hit objects in the scene? What is the probability that a point on the surface of an object is in shadow? How many shadow rays are generated? If we can answer these questions, we will have a good prediction for the number of calls to each of the basic operations during rendering.

These questions will be answered by first detailing a model to find the expected number of nodes in a shadow ray tree. We will relate this value to the number of reflection and transmission rays. A branching process will be used to model reflection and transmission ray recursion. We will find the PMF and expectation for the number of nodes in each generation of the tree. The model will then be expanded to account for the user-set minimum ray weight and maximum ray depth values. Finally, we will review the assumptions, strengths, and weaknesses of the model.

5.2.3 Eyeray Hits

We will refer to the rays cast from the camera through the image plane and into the scene as eyerays. The ray tracer provides for super-sampling eyerays within a pixel and averaging sample contributions to a pixel. Samples can be jittered randomly within a pixel or non-randomly. Non-random jittering positions pixels in a grid such that not only are samples within a pixel equidistant from each other, they are also equidistant from samples in neighboring pixels. The number of samples generated per pixel and their jittering type are options set by the user.

Estimating the number of eyeray hits is central to the prediction of most of the basic operations. When the shade function is called for every sample, all shading model and recursive routines are called based on whether the sample intersected an object in the scene. Since raycasting is a computationally intensive process, the prediction model must be capable of predicting the number of eyeray hits without casting rays into the scene.

The predictor will estimate eyeray hits using the screen space bounding box of scene objects. First, we compute the world space bounding boxes for each scene object. These bounding boxes are transformed into eye space (see Figure 8). In eye space, the camera is the origin and the world orthobasis is aligned with the direction of viewing and the axes of the image plane. To get the tightest screen space bounding box fit, the corners of each eye space bounding box are projected into screen space. Then a two-dimensional screen space bounding box is constructed around the projected corners (see Figure 9).

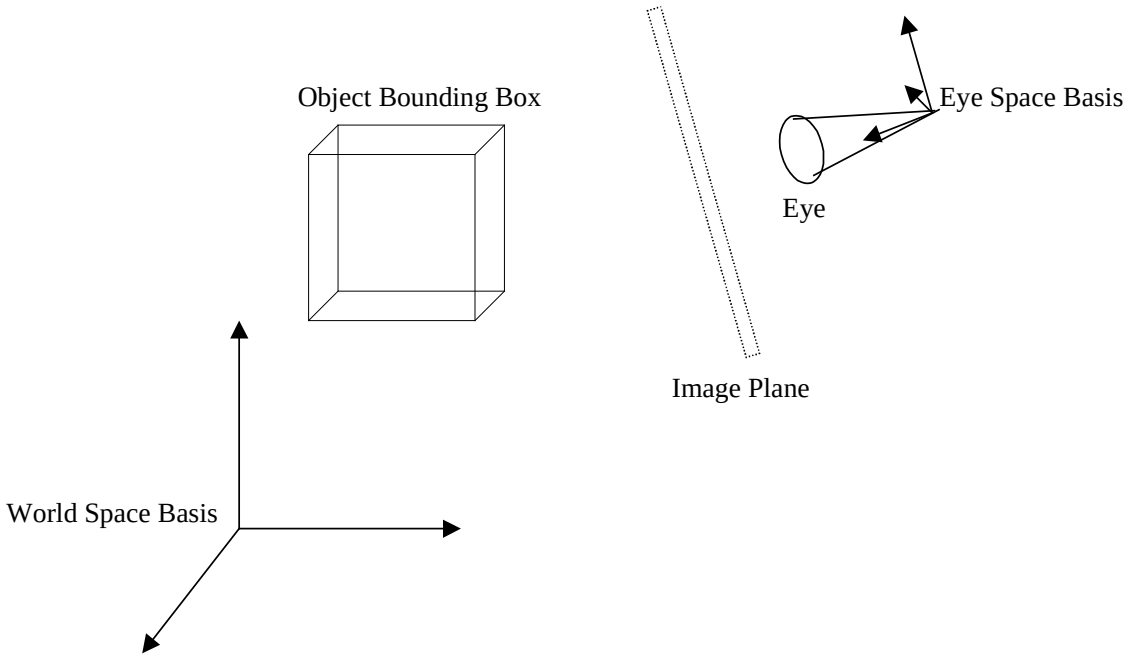

Figure 8: World space bounding boxes are transformed to eye space

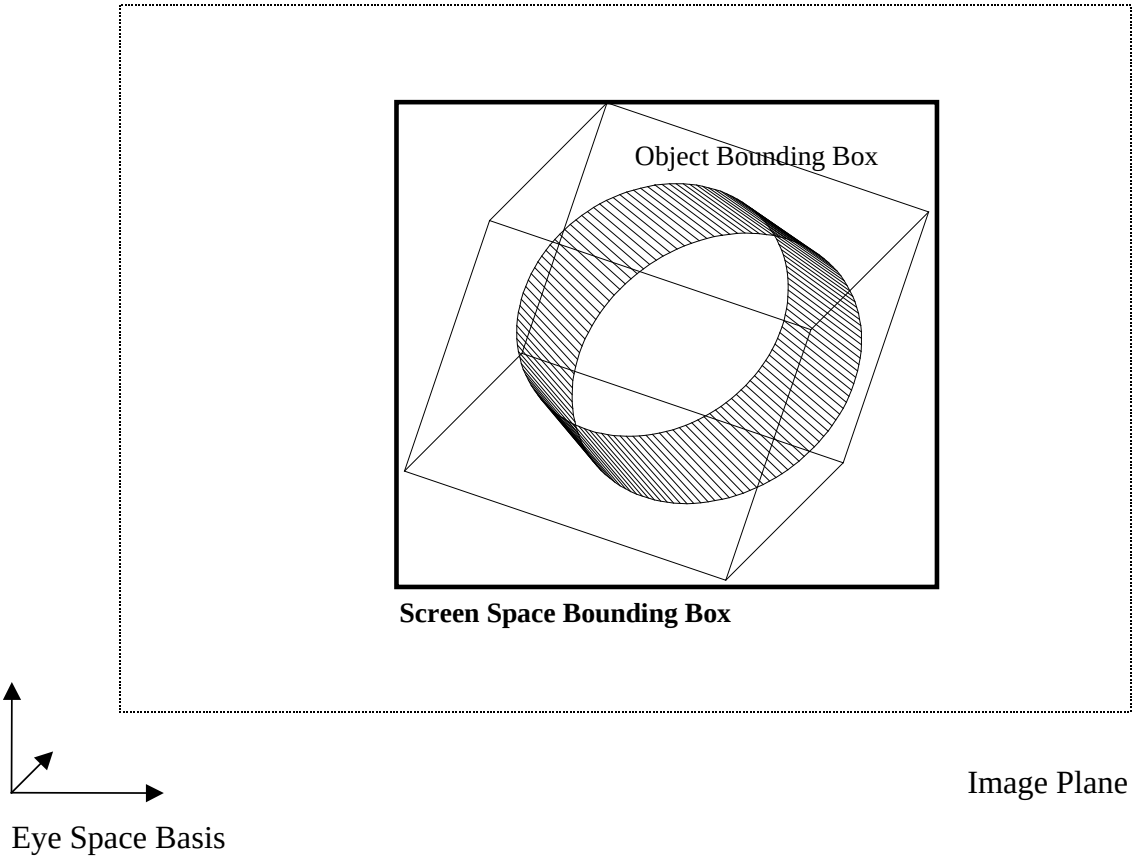

Figure 9: Screen space and object space bounding boxes

If, instead, an eye space-aligned bounding box had first been constructed around the world space-aligned bounding box, more empty space would result in the two-dimensional screen space bounding box. This is because the projection of the eye space-aligned bounding box onto screen space would project, and so increase the error between the eye space and world space bounding boxes as well.

The two-dimensional screen space bounding boxes that result are computed in terms of screen space pixel integers. The error that results from not using floating point representation to capture individual sample activity is far outweighed by the general error in using bounding boxes. In addition, computing how random samples would jitter involves approximation even if floating point representation was used for the screen space bounding boxes.

The screen space bounding boxes that are computed for scene objects may overlap. However, the eyerays cast by the ray tracer will only intersect a single object in their path. Recursive rays will be treated separately. The screen space bounding boxes must be clipped to one another to prevent double counting pixel eyeray hits. Screen space bounding boxes are clipped into a variable number of boxes, all of which remain screen space axis-aligned (see Figure 10).

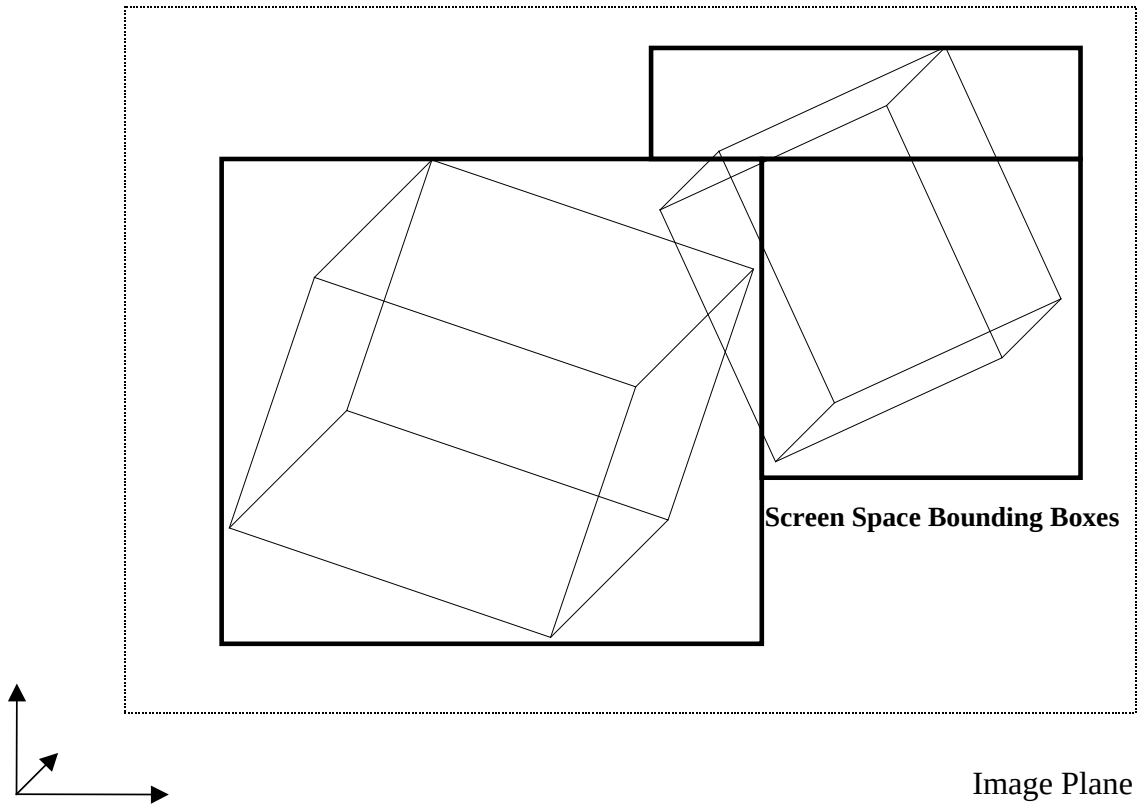

Figure 10: Overlapping screen space bounding boxes are clipped

All pixels within the screen space bounding boxes are considered pixels that would generate eyeray hits. The sum of the areas in pixels of all these boxes is divided by the total number of pixels in screen space. The resulting floating point number is referred to as the screen space density and represents the fraction of eyerays that intersect with objects in the scene.

The greatest contributor to error in the screen space density is the error in the original world space bounding boxes of each object. This error is then compounded when the bounding box is transformed into eye space and bounded in two dimensions. We will later show how this error affects scene prediction as a whole. It should also be pointed out that the screen space density does not distinguish between objects in the scene. It approximates the fraction of eyerays which hit any object, not objects in particular.

If the bounding box an object occupies less than one pixel in area, it is ignored even though the object may be hit by an eyeray. If a screen space bounding box reaches beyond the bounds of the image plane, it is clipped to the image plane.

5.2.4 Recursively Generated Rays

When an eyeray hits an object in the scene, the shading model may generate reflection and transmission rays. Whether these rays are generated depends on user preferences and the properties of the objects hit. When these rays in turn hit other objects, more reflection and transmission rays are generated in a recursive process. The user can set whether the rays are generated at all and, if so, what the maximum depth is for the recursive process. In addition, rays will only be cast if the weight of the ray is above a user-set minimum weight. Ray weight depends on the reflective and transmissive properties of the object compounded with those of previous objects hit in the branch of recursion (see Figure 11).

When a ray hits an object, if that object is reflective and its specularity constant, ks, multiplied by the current recursion branch weight is above the user-set minimum, a reflection ray is cast. If the object is non-opaque and its transparency multiplied by the current recursion branch weight is above the user-set minimum, a transmission ray is cast. Therefore, each ray-object intersection has the potential to recursively cast both a reflection and a transmission ray. Each intersection also casts one shadow ray tree for each light. Shadow rays in the direction of a light source are recursively generated until either the light or an opaque object is intersected, or the ray does not intersect anything.

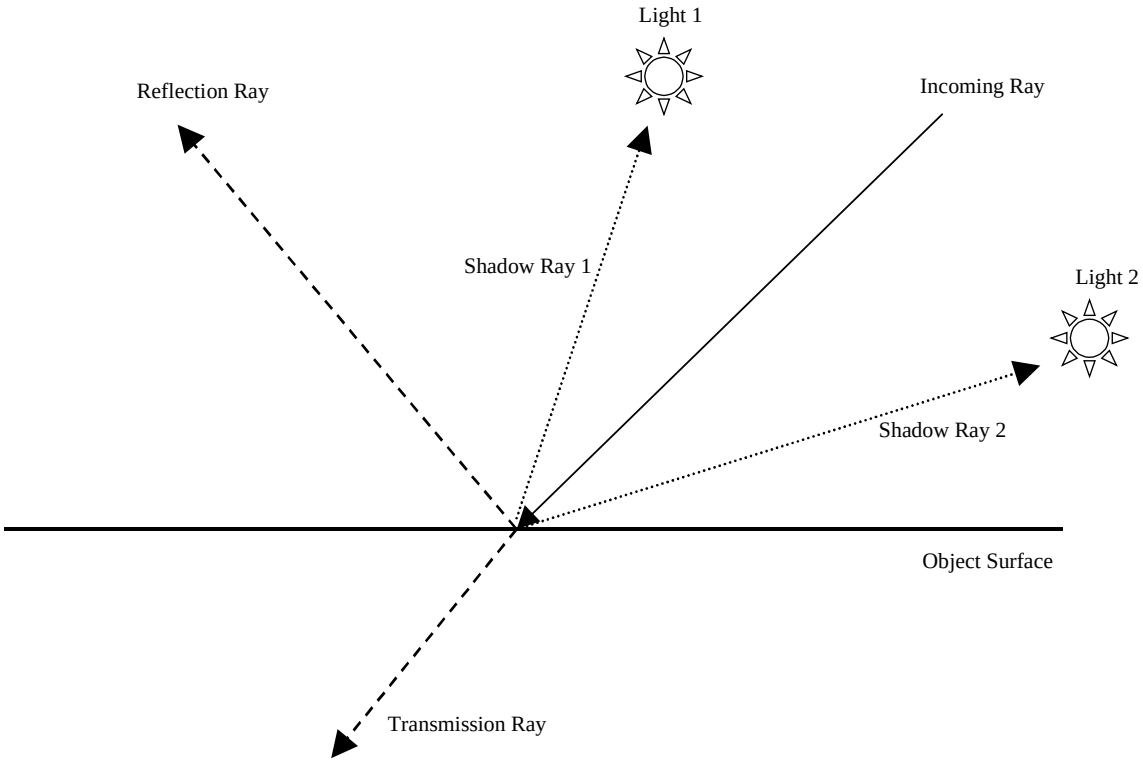

Figure 11: Recursive rays cast on ray-object intersection

6 Recursion Model

We will address our questions on the recursive process by first modeling the generation of shadow rays as a branching process. Then we will model reflection and transmission ray generation as a separate branching process. Finally, we will critique our models, exploring their assumptions, strengths, and weaknesses.

6.1 Shadow Trees

We will begin with shadow trees since they are less complicated. If a shadow ray intersects a non-opaque object, another shadow ray is recursively generated in the same direction. The tree ends when a ray intersects the light at which it is aimed, an opaque object, or nothing at all.

6.1.1 Probabilities and Assumptions

Let j = probability that a random ray, R, intersects a non-opaque object

Let s = probability that a random ray, R, intersects an opaque object

= fraction of ray-object intersection that are in shadow

The probability that a random ray, originating from the surface of one object, intersects a non-opaque object is proportional to the density of non-opaque objects in the scene. We model this probability as the sum of volumes of non-opaque objects divided by the volume of a bounding box of the scene. Since a ray is no more likely to intersect two overlapping objects than one object occupying the same volume of space, the bounding boxes of all non-opaque objects are clipped in three dimensions against each other. This results in a variable number of non-overlapping bounding boxes surrounding all non-opaque objects in the scene. The bounding box for the scene encloses all objects together, including the empty space between them, and the light that corresponds to this shadow tree.

j = A * (non-opaque object density) where A is a constant

Using the scenes given in appendix section 9.3, A was found empirically to be 3.93.

Using surface areas of objects to compute the density may be a more accurate approach. However, we have chosen to use volumes because the bounding boxes must be clipped against each other. The resulting variable number of bounding boxes will consist both of faces that represent part of the exposed surface area, and part of the interior of an object or object group. In addition, some faces may be only partially exposed (see Figure 12). Finding the appropriate surface area would be a non-trivial task. We have chosen to use the volume instead.

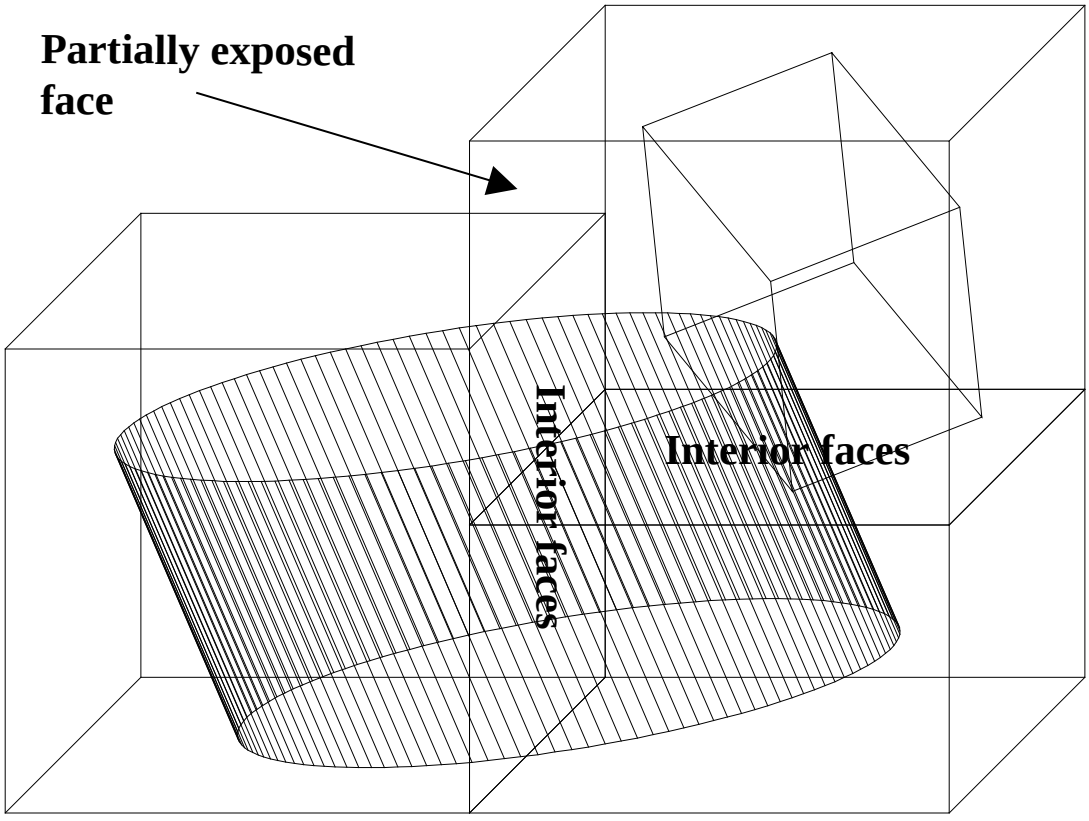

Figure 12: Clipped bounding boxes make surface area computation difficult

If there are closed objects in the scene, however, the situation is complicated. A ray-object intersection with a non-opaque object will spawn a shadow ray originating on the object's surface and in the direction of the object's interior. If the object is closed, this shadow ray has a probability of 1 of intersecting the same object again, as long as the light is not inside the object.

Assuming all objects to be not only closed, but also convex, and no lights to be allowed inside of objects, twice as many shadow rays will intersect non-opaque objects as predicted; one intersection is for exterior rays and one is for interior rays.

$$p = 2^*j$$

6.1.2 The Recursive Branching Process

We will model the recursive process of shadow ray generation as a tree where each node represents a ray-object intersection (see Figure 13). Let the random variable $Y_{k,g}$ represent the number of children node k in generation g produces in generation $g+1$. $Y_{k,g}$ is defined as follows:

$$\text{Prob}(Y_{k,g} = 0) = 1 - p \qquad \text{for all } k, g$$

$$\text{Prob}(Y_{k,g} = 1) = p$$

Let, $$X_g = \sum_k Y_{k,g}$$

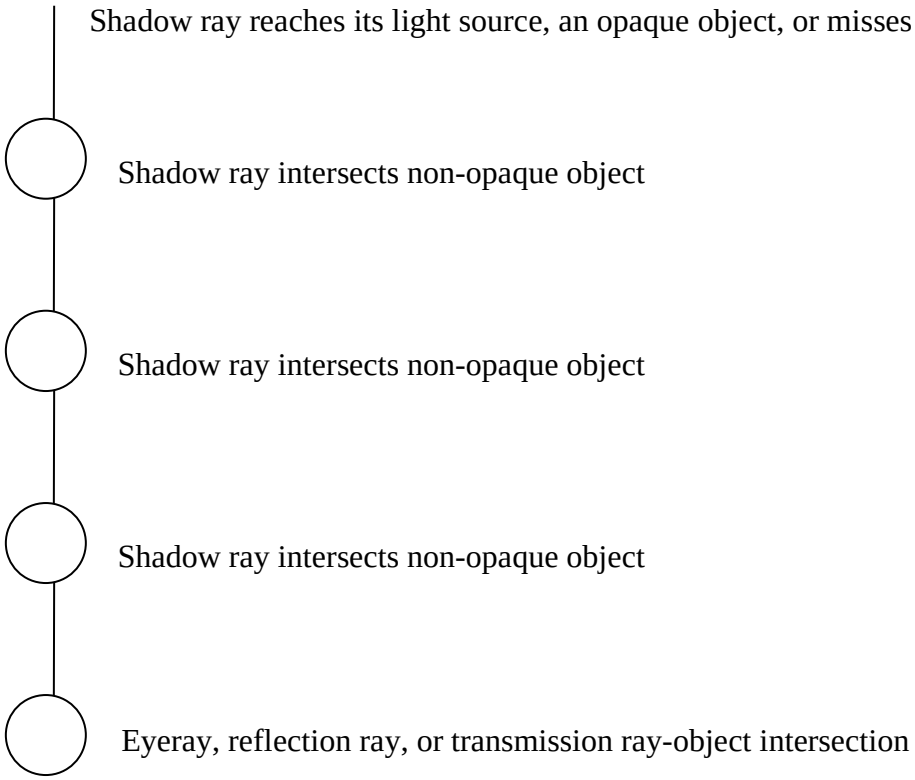

Figure 13: A sample shadow ray tree

6.1.3 PMF

We can find the probability the branching process will eventually die out.

Let u_i = Prob($Y_{k,g}$ = i)

$F_{i,0}(n) = (F_{1,0}(n))^i$ = Prob(process will die out in n generations | $E(X_0) = i$)

$F_{1,0}(n) = g(F_{1,0}(n-1))$

where $g(z) = \sum_{k \geq 0} u_k z^k$

$F_{1,0}(1) = u_0$

$F_{1,0}(\infty)$ = the smallest root of the equation $g(z) = z$

$g(z) - z = u_0 + u_1 z - z = 0$

$z = \frac{u_0}{1-u_1} = F_{1,0}(\infty) = 1$

We can find the probability mass function of our random variable X:

$P(X_{g+1}=0) = u_0^{Xg} = (1-p)^{Xg}$

$P(X_{g+1}=1) = X_g u_0^{Xg-1} u_1 = X_g(1-p)^{Xg-1}p$

6.1.4 Expectations

A more important value for our prediction, however, is the expected number of nodes in the branching process. First, we will find the expected number of nodes in each generation, g.

$$E(X_g) = \sum_{k=1}^{X_{g-1}} E[Y_{k,g-1}]$$

$= E(X_{g-1}) * E(Y_{k,g-1})$ by the independence of X, Y

$= E(X_0) * (E(Y))^g$

For our random variable Y, $E(Y) = p$

$E(X_g) = E(X_0) * p^g$

The sum of X_i over all generations i gives us the number of nodes in the branching process.

$$E(\#\text{ nodes in shadow tree}) = \sum_g E[X_g]$$

Since at most one shadow ray could be generated for each non-opaque object in the scene, the number of non-opaque objects in the scene is the maximum depth of each shadow tree. Let d = the number of non-opaque objects in the scene.

$$E(\#\text{ nodes in shadow tree}) = \sum_{g=0}^{d} E[X_g] = \sum_{g=0}^{d} E[X_0] * p^g$$

The number of rays in generation 0 is 1 for each shadow tree.

$$E(\#\text{ nodes in shadow tree}) = \sum_{g=0}^{d} p^{g}$$

Of course, p will be different for the shadow tree of each light source.

Let n = the number of shadow tree generated for a given light source. Let l = the number of light sources.

Let p_i = p for light source i.

$$E(\#\text{ nodes in all shadow trees}) = \sum_{i=1}^{l}[n * \sum_{g=0}^{d} p_i{}^{g}]$$

$$= n^* \sum_{i=1}^{l}\sum_{g=0}^{d} p_i{}^{g}$$

Since one shadow tree is generated for each light source for every ray-object intersection resulting from eyerays, reflection rays, or transmission rays. We have already shown how to find the number of eyeray-object intersections. Next we will find the number of reflection and transmission ray-object intersections.

Before we do that, we will return to the question raised in section **6.3**. When predicting how long we expect a call to *shadowing()* to take, we must find the number of times the loop is run by estimating the number of non-opaque objects between the point we are shading and the light source given as an argument. To do so, we will find the generation g in which the sum of all shadow trees to a given light have less than 1 node. The value is then averaged across light sources.

```
Let l = # of light given as an argument to shadowing
Let d = the number of non-opaque objects in the scene (same as above)
Let n = the number of shadow tree generated for a given light source
(same as above)

      int numNodes = 1;
```

```
g = 0;
int numLoops = 0;
for (int l=0;l<numLights;l++) {
      while ((numNodes≥1)&&(g<d)) {
            numNodes = n*p_l^g;
            g++;
      }
      numLoops += g-1;
}
numLoops /= numLights;
```

6.2 Reflection/Transmission Trees

The recursive process generates many calls to basic operations and greatly affects the rendering time of the scene. For every reflection and transmission ray cast, several operations are called. Which operations are called further depends on whether the recursive ray intersected an object in the scene. Modeling the recursive reflection and transmission rays is crucial as a result. We will model this recursion as a branching process.

6.2.1 Probabilities and Assumptions

For a random ray, R, and a random object, O,

r = Prob(O is reflective)

t = Prob(O is non-opaque)

h = Prob(R hits an object)

Given that we have hit an object, O, and are computing the radiance at the point of intersection:

p = Prob(a reflection ray is cast and that ray hits an object)

= r * h

q = Prob(a transmission ray is cast and that ray hits an object)

= t * h

The probability that an object that is hit is reflective is proportional to the relative size of reflective objects as compared to all objects. We model the probability that a given object is reflective by finding the percentage contribution of reflective objects to overall object volume. The volumes of objects are approximated by the volumes of their bounding boxes. Although surface areas may again be more appropriate, and bounding boxes are not clipped against each other this time, we will continue to use volumes for consistency.

ks_i = specularity constant of object i; $0 \leq ks \leq 1$

vol_i = volume of bounding box of object i

$$r = \left(\sum_i (vol_i * ceil(ks_i))\right) / \sum_i vol_i$$

Similarly for transmissive objects:

$trans_i$ = transparencyof object i; $0 \leq transparency \leq 1$

vol_i = volume of bounding box of object i

$$t = \left(\sum_i (vol_i * ceil(trans_i))\right) / \sum_i vol_i$$

The probability that a random ray, originating from the surface of one object, intersects another object is proportional to the density of objects in the scene, assuming that the camera is not completely enclosed by objects or inside an object. We model this probability as the sum of volumes of all objects divided by the volume of a bounding box of the scene. Whereas in the previous case for r and t we did not care if object bounding boxes overlapped, we cannot allow that here. A ray is no more likely to intersect

two overlapping objects than one object occupying the same volume of space. The bounding boxes of all objects are clipped in three dimensions against each other, resulting in a variable number of non-overlapping bounding boxes surrounding all objects in the scene. The bounding box for the scene encloses all objects together, including the empty space between them.

$$h = C * (\text{object density})$$

If there are closed objects in the scene, however, the situation is complicated. A ray-object intersection with a sufficiently transparent object will spawn a transmission ray originating on the object's surface and in the direction of the object's interior. If the object is closed, this transmission ray has a probability of 1 of intersecting the same object again.

Furthermore, if the object is also reflective, then the next hit will spawn a reflection ray pointed toward the interior of the object. Reflection rays will bounce around the interior of the object with a probability of 1, also generating more transmission rays, until the maximum recursion depth is reached. I will use the fact that p is proportional to the density of objects that are both reflective and transmissive to model this effect.

The object density constant, C, has been included in constants A and B.

$$p = A * r * (\text{object density} + \text{density of reflective and transmissive objects})$$

Using the scenes given in appendix section 9.3, A was found empirically to be 0.75.

$$q = B * t * \text{object density}$$

Using the scenes given in appendix section 9.3, B was found empirically to be 2.74.

6.2.2 The Recursive Branching Process

We will model this recursive process as a tree where each node represents a ray-object intersection (see Figure 14). Let the random variable $Y_{k,g}$ represent the number of children node k in generation g produces in generation $g+1$. $Y_{k,g}$ is defined as follows:

$\text{Prob}(Y_{k,g} = 0) = 1 - p - q + pq$ for all k, g

$\text{Prob}(Y_{k,g} = 1) = p + q - 2pq$

$\text{Prob}(Y_{k,g} = 2) = pq$

Here, p can be thought of as the probability of spawning a node reflectively, while q can be thought of as the probability of spawning a node transmissively. Let,

$X_g = \sum_k Y_{k,g-1}$ = # of nodes in generation g

We can find the probability the branching process will eventually die out.

Let $u_i = \text{Prob}(Y_{k,g} = i)$

$F_{i,0}(n) = (F_{1,0}(n))^i = \text{Prob}(\text{process will die out in n generations} \mid E(X_0) = i)$

$F_{1,0}(n) = g(F_{1,0}(n-1))$

where $g(z) = \sum_{k \geq 0} u_k z^k$

$F_{1,0}(1) = u_0$

$F_{1,0}(\infty)$ = the smalles root of the equation g(z) = z

$g(z) - z = u_0 + u_1 z + u_2 z^2 - z = 0$

$z = (1 - u_1 \pm \sqrt{(1-u_1)^2 - 4u_2 u_0}) / 2u_2 = F_{1,0}(\infty)$

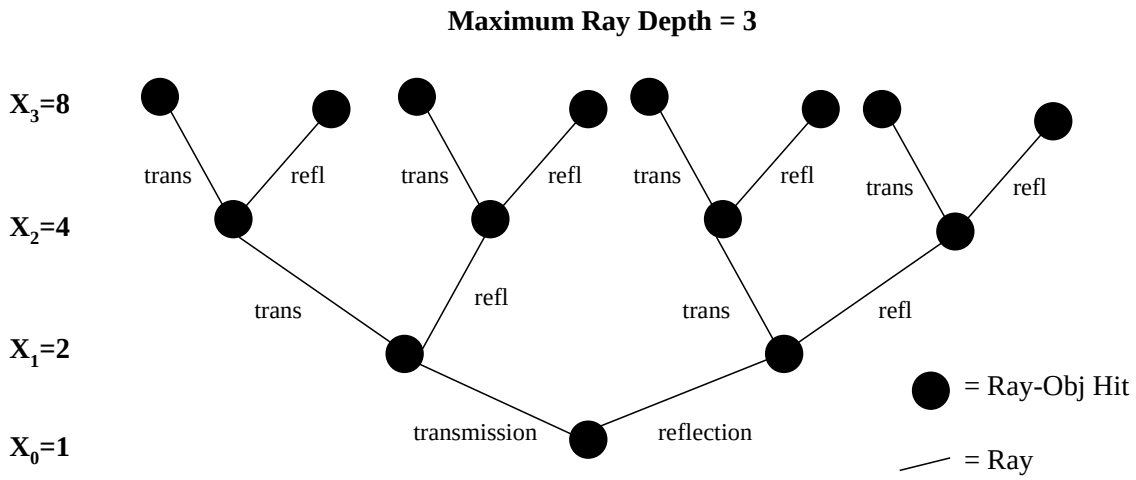

Figure 14: The recursive branching process

6.2.3 PMF

We can find the probability mass function of our random variable X:

$$P(X_{g+1}=0) = u_0^{X_g}$$

$$P(X_{g+1}=1) = X_g u_0^{X_g-1} u_1$$

$$P(X_{g+1}=2) = X_g u_0^{X_g-1} u_2 + \frac{X_g!}{2!(X_g-2)!} u_0^{X_g-2} u_1^2$$

…

$$P(X_{g+1}=k) = \sum_{i=0}^{floor(\frac{k}{2})} \frac{X_g}{(k-i)!(X_g-(k-i))!} u_2^{\,i} u_1^{\,k-2i} u_0^{\,X_g-k+i}$$

6.2.4 Expectations

A more important value for our prediction, however, is the expected number of nodes in the branching process. First, we will find the expected number of nodes in each generation, g.

$$E(X_g) = \sum_{k=1}^{X_{g-1}} E[Y_{k,g-1}]$$

$$= E(X_{g-1}) * E(Y_{k,g-1}) \qquad \text{by the indepence of X, Y}$$

$$= E(X_0) * (E(Y))^g$$

For our random variable Y, $E(Y) = p + q$.

$$E(X_g) = E(X_0) * (p+q)^g$$

The sum of X_i over all generations i gives us the number of nodes in the branching process.

$$E(\#\text{ nodes in process}) = \sum_g E[X_g]$$

6.2.5 Depth

The next question is how many generations will there be in our branching process?

6.2.5.1 User-set Maximum depth and Minimum ray weight

The user sets the maximum depth of the recursive ray generation process (MRD). This number is the upper bound for the number of generations in the process. However, branches may die out sooner. As reflection rays are bounced around the scene, they lose energy unless the objects hit are perfect reflectors. As transmission rays pass through objects, they lose energy unless the objects are perfectly transparent. The loss of energy is simulated by the lowering of the weight of the ray. When a ray's energy is low, it will not contribute greatly to the radiance of the next object it hits. To prevent unnecessary computation, the user can set the minimum value for a recursive ray's weight. If the ray's weight is below the lower bound, recursion is stopped.

6.2.5.2 Average Scene Weights

In order to predict at what depth a ray will fall below the lower bound on its weight, we must estimate the specularity constants and transparencies for objects the ray hits. We do so by finding a surface area weighted average for the scene's specularity and transparency.

ks_i = specularity constant of object i

vol_i = volume of bounding box of object i

scene reflectivity = $(\sum_i (ks_i * vol_i)) / \sum_i vol_i$

$trans_i$ = transparency of object i

vol_i = volume of bounding box of object i

scene transparency = $(\sum_i (trans_i * vol_i)) / \sum_i vol_i$

Objects with more volume will, on average, be hit more often than those with less, so their values should contribute more to the scene's average. All specularity constants and transparencies are between 0 and 1, inclusive so the scene averages are as well.

6.2.5.3 Finding a Minweight Bounded Depth

If the rays of a recursive branch all strike objects with ks = transparency = w, we can determine the depth at which the accumulated weight will fall below the minimum ray weight.

minweight bounded depth = $\log_w$ (minweight)

The overall maximum ray depth is given by:

$$\text{MRD} = \text{minimum(minweight bounded depth, user-set MRD preference)}$$

We now have:

$$\text{E(\# nodes in process)} = \sum_{g=0}^{MRD} E[X_0] * E[Y]^g$$

6.2.6 Relating the Branching Process to Our Questions

Since each node represents a ray-object intersection, the nodes in the 0^{th} generation represent hits resulting from eyerays. Therefore,

$$\text{E(\# nodes in process)} = \sum_{g=0}^{MRD} E[eyerayhits] * (p+q)^g = \sum_{g=0}^{MRD} E[eyerayhits] * (r*h + t*h)^g$$

Every node in the tree represents a ray-object intersection. Generations 1 through MRD represent reflection and transmission ray hits.

$$\text{Reflection ray hits} = (\text{E(\# nodes in process)} - \text{E}(X_0)) * \frac{p}{p+q}$$

$$\text{Transmission ray hits} = (\text{E(\# nodes in process)} - \text{E}(X_0)) * \frac{q}{p+q}$$

How many reflection and transmission rays were cast that did not intersect any objects? Since every node has the potential to cast both a reflection and transmission ray, if a node does not generate two children for

the next generation, either one or more of the rays were not cast or they were cast and did not intersect any objects. If we assume both types of rays are cast from every node,

$$\text{Reflection ray misses in generation g | all nodes cast both ray types} = E(X_{g-1}) * \frac{1-p}{p+q}$$

$$\text{Transmission ray misses in generation g | all nodes cast both ray types} = E(X_{g-1}) * \frac{1-q}{p+q}$$

A node casts a reflection ray if two conditions are true:

1. the object hit represented by the node is reflective ($k_s > 0$)
2. given the object is reflective, the ray weight will be above a minimum user-set preference

A node casts a transmission ray if two conditions are true:

1. the object hit represented by the node is non-opaque (transparency > 0)
2. given the object is non-opaque, the ray weight will be above a minimum user-set preference

For the moment, we will ignore the second condition for both ray types. Since we have already found the probability a random object is reflective, r, or non-opaque, t,

$$\text{Reflection ray misses in generation g} = E(X_{g-1}) * \frac{1-p}{p+q} * r$$

$$\text{Transmission ray misses in generation g} = E(X_{g-1}) * \frac{1-q}{p+q} * t$$

This gives us:

$$\text{Reflection ray misses} = \sum_{g=1}^{MRD} E[X_{g-1}] * \frac{1-p}{p+q} * r$$

$$\text{Transmission ray misses} = \sum_{g=1}^{MRD} E[X_{g-1}] * \frac{1-q}{p+q} * t$$

The total number of rays cast become:

Reflection rays cast = Reflection ray hits + Reflection ray misses

Transmission rays cast = Transmission ray hits + Transmission ray misses

We will show, however, that this is not the whole story.

6.2.7 Passing the minimum weight requirement for rays

In our branching process, however, ray branches may be made up of nodes generated from both reflection and transmission rays, but the minweight bounded depths are different for each type of ray since their average scene weights differ. At any depth in the recursion, the ray may accumulate reflectivity or transparency. We cannot merely find an overall minweight-bounded depth by averaging the scene's reflectivity and transparency. If the reflectivity were high and the transparency were low, we would expect a recursive process with a high depth, made up of both ray types in the lower levels, but only reflection rays in the higher levels. Averaging the scene's reflectivity and transparency would result in a predicted process of medium depth consisting of reflection and transmission rays equally throughout. Since reflection and transmission rays call separate procedures, which take different amounts of time, predicting the type of ray that generates each ray-object intersection is important.

To find the expected number of ray-object intersections in generation g, start with the number of nodes in generation g-1. Find how many of these start from intersections with reflective and/or non-opaque objects. Then find how many of those pass the minweight requirement. Finally, predict how many cast rays will intersect other objects.

6.2.8 Assume rays pass minweight in pairs

For now, we will ignore branches where one type of ray passes and the other fails the minweight test. Suppose we know, for any generation g, how many branches, S_g, could survive the minweight test. We can find the fraction that survive in any generation g:

fraction of survivors in g = $\frac{S_g}{2 * S_{g-1}}$

$S_0 = E(X_0)$ since all eyerays have weight 1 $\geq$ minimum ray weight

The expected number of ray-object intersections in any generation does not differentiate between those whose rays pass the minweight test and those whose rays do not pass the minweight test. Therefore, we can say:

$E(X_0) = E(\text{eyeray hits})$

$E(X_1) = E(X_0) * (p+q) * \frac{S_1}{2 * S_0}$

$E(X_2) = E(X_1) * (p+q) * \frac{S_2}{2 * S_1}$

…

$E(X_g) = E(X_{g-1}) * (p+q) * \frac{S_g}{2 * S_{g-1}}$

6.2.9 Finding S_g for all generations g

If the minweight bounded depths using both scene averages are above the user-set MRD preference, all rays that are attempted will pass the minweight test since the user-set bound will be reached before the minweight bound. If not, some will fail the minweight test. We can find which process branches survive and which die out due to the minweight bound using our average reflectivity and transparency. Suppose that the scene reflectivity is greater than the transparency. Transparent rays will then push the weight of a branch toward the minimum ray weight faster than reflective rays. Let k equal the number of reflective rays that lower the branch weight as much as one transparent ray. Let r equal the

number of nodes in the current branch generated by reflection rays and let t equal the number generated by transmission rays, where r + t = g, the current generation (see Figure 15). Let rMRD be the maximum number of reflection rays cast in a row before the minweight test fails. rMRD can be found by setting w = aveKS and using the equations in section **6.2.5.3** to find how many aveKS-weighted rays will be needed to fall below the minweight. tMRD can be found similarly using w = aveTrans.

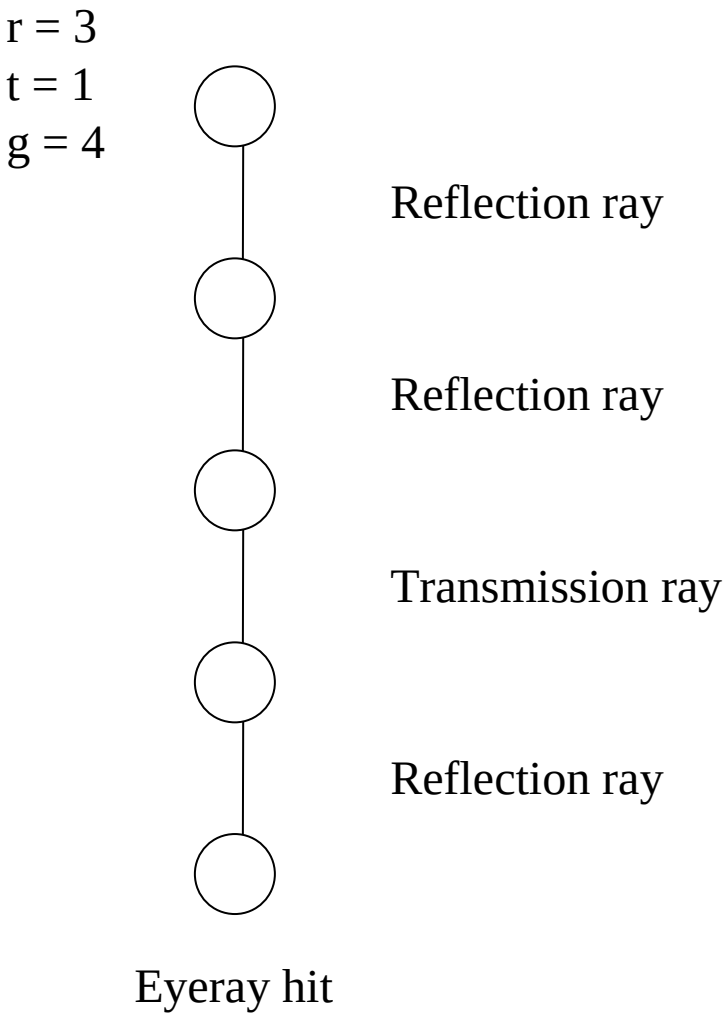

Figure 15: Branch example ; g = r + t

```
aveKS = scene average reflectivity
aveTrans = scene average transparency
if (aveKS>aveTrans) { // case 1
      k = log_aveKS(aveTrans)
```

```
            if ((r+(k*t)) > rMRD) then ray for generation r+t fails
minweight test
      }
      else {// case 2
            k = log_aveTrans(aveKS)
            if ((t+(k*r)) > tMRD) then ray for generation r+t fails
minweight test
      }
```

In case 1, if `((r + (k*t)) > rMRD)` then the ray for the current generation will fail the minweight test, whether that ray was reflection and contributing to r, or transmission and contributing to t. If k=1, both types will fail the minweight test at the same node and the branch will stop. When (k>1), if a node's transmission ray fails the minweight test but the reflection ray passes, no further nodes in the branch will be able to cast a transmission ray; they will all fail the test as well. Therefore, the branch becomes a simple tree with random variable $Y_{k,n}$, the number of offspring of node k in generation n: (assuming case 1 from the code above)

$$\text{Prob}(Y_{k,n}=0) = 1-p$$

$$\text{Prob}(Y_{k,n}=1) = p$$

We will refer to this type of tree as singly-recursive because each node can generate, at most, one child for the next generation. The first node in the singly-recursive tree is what would have been the reflectively generated node in generation g of our branching process (see Figure 16). The number of nodes in the subtree is:

$$\text{E(\# nodes in a singly-recursive tree)} = \sum_i X_i = \sum_i E[X_0]*E[Y]^i = \sum_i E[X_0]*p^i = \sum_{i=1}^{depth} p^i$$

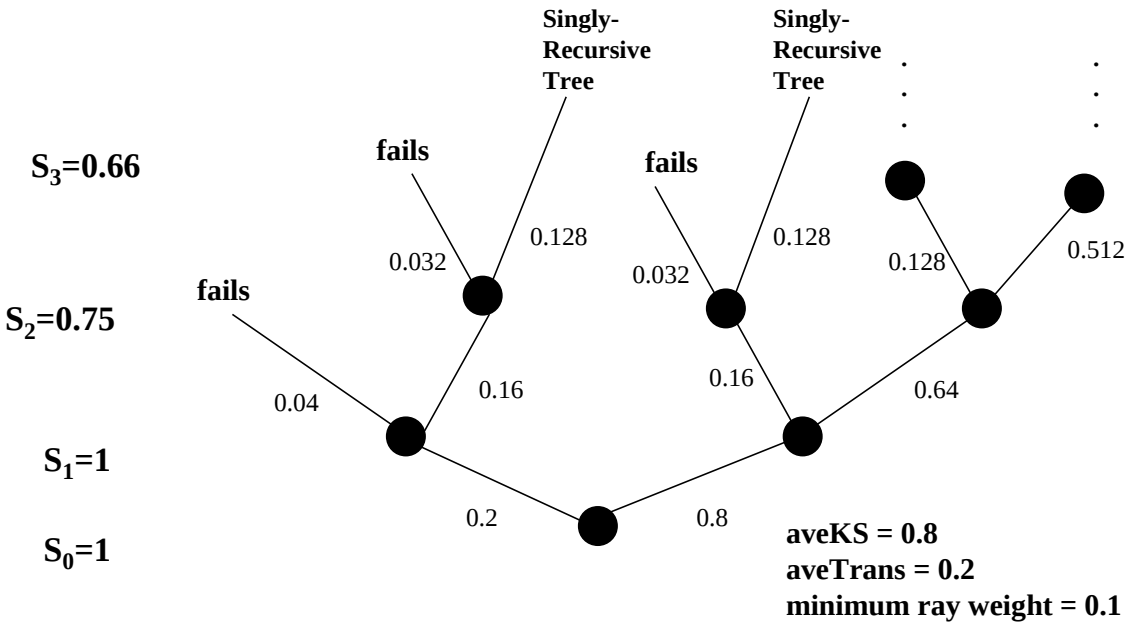

Figure 16: Example illustrating S_g and singly-recursive trees

How do we find the total number of branches in a generation that fail the minweight test? n_g is the number of rays cast from generation g-1 which fail the minweight test before being able to create nodes in generation g. The branches which have died out due to the minweight test by generation g are those whose most recent node was created by a transmission ray and who fail the test (r+kt > rMRD). If (r+kt == rMRD+1), no singly-recursive trees will be generated. If (r+(k-1)t <= rMRD) and (r+kt>rMRD), then one transmission ray put the weight over the top and a singly-recursive tree will be generated.

When we encounter a singly-recursive tree, to what depth to we allow it to extend? The upper bound would be the remainder of the user-set maximum ray depth. Note that the current generation will become the 0^{th} generation of the singly-recursive tree. It may be forced to die out before the user-set maximum depth by the minweight requirement. If one transmission ray pushed (r+kt) beyond MRD to generate the singly-recursive tree, then r+(k-1)t would be the current weight of the branch that becomes the singly-recursive tree's initial weight. The singly-recursive tree will fail the minweight test in (rMRD – (r+(k-1)t)) levels. Therefore, the singly-recursive tree's depth becomes the minimum of the user-set maximum depth and (rMRD – (r+(k-1)t)).

The number of nodes in the 0^{th} level of the singly-recursive tree is equal to the number of combinations of r and t for which (r+kt > MRD) and r+t = g. When a branch fails the minweight test, whether or not singly-recursive trees are generated, the number of branches that fail is recorded in n_g to subtract from S_g as defined above.

```
MRD = max(rMRD,tMRD)
k = max(log_aveKS (aveTrans), log_aveTrans(aveKS))
n_g = 0;
for (int i=0;i<=g;i++) {
      failweight = i + k*(g-i);
      if (failweight == MRD+1) {
            n_g += g!/(i!(g-i)!);
      }
      else if ((failweight - k) <= MRD) &&
            (failweight > MRD)) {
            n_g += (g-1)!/(i!((g-1)-i)!);
            E[X_0] = (g-1)!/(i!((g-1)-i)!);
            d = minimum((MRD-(failweight-k)), (userset-
      g+1));
            SinglyRecursiveNodes += sum_{g=1}^{d} E[X_{g-1}]* max P;
      }
}
```

This method breaks down, however, when one or more of the scene averages is zero. If aveKS > aveTrans, aveTrans would have to be zero for k to be infinite, all the branches in the process would be reflection subtrees, and $n_0 = X_0$. If only aveKS were zero, all the branches in the process would be transmission trees. If both the scene averages were zero, there should be zero nodes in our branching process. These cases are taken care of separately. For example, in the case when all branches are reflection subtrees:

```
if (q < minweight) {
      S_g = 0 for all g = 0 to user-set maximum depth
      d = minimum(rMRD,userset);
      E[X_0] = eyeray hits;
      SinglyRecursiveNodes = Σ_{g=1}^{d} E[X_{g-1}] * max P;
}
```

6.2.10 Modeling branching depth more accurately

We have shown that not all of the expected nodes in our branching process will necessarily be created. Some will fail the minweight test. The number that pass the minweight test in generation g is given by:

$$X_g = (2 * X_{g-1}) - n_g$$

We will handle the singly-recursive trees as processes separate from our branching process and add them back in later. Using our definition before that S_g gives the number of branches in generation g that could have survived the minweight test,

$S_g = (2 * S_{g-1}) - (2 * n_g)$

where $S_0 = E(X_0)$ and n_g is found using the code segment above

E(# of nodes in doubly-recursive tree) = $\sum_{g=1}^{user-setMRD} E[X_{g-1}]*(p+q)*\frac{S_g}{2*S_{g-1}}$

where $E(X_0)$ is fixed at the number of eyeray hits

We have from above,

Reflection ray hits = (E(# nodes in process) – $E(X_0)$) * $\frac{p}{p+q}$

Transmission ray hits = (E(# nodes in process) – $E(X_0)$) * $\frac{q}{p+q}$

Since we have handled our singly recursive trees separately, these equations now become,

Reflection ray hits = ((E(# nodes in doubly-recursive tree) – $E(X_0)$) * $\frac{p}{p+q}$) + E(# of nodes in singly-recursive reflection trees)

$= \sum_{g=1}^{MRD} E[X_{g-1}]* p *\frac{S_g}{2*S_{g-1}}$ +SR_r

Transmission ray hits = ((E(# nodes in doubly-recursive tree) – $E(X_0)$) * $\frac{q}{p+q}$) + E(# of nodes in singly-recursive transmission trees)

$$= \sum_{g=1}^{MRD} E[X_{g-1}] * q * \frac{S_g}{2 * S_{g-1}} + SR_t$$

Either all singly-recursive trees will be generated by reflection rays, or all will be generated by transmission rays.

$$\text{Either } SR_r = \sum_{g=0}^{MRD} n_g \text{ and } SR_t = 0$$

$$\text{or } SR_r = 0 \text{ and } SR_t = \sum_{g=0}^{MRD} n_g$$

We also have from above:

$$\text{Reflection ray misses} = \sum_{g=1}^{MRD} E[X_{g-1}] * \frac{1-p}{p+q} * r$$

$$\text{Transmission ray misses} = \sum_{g=1}^{MRD} E[X_{g-1}] * \frac{1-q}{p+q} * t$$

We must also add in the misses from singly-recursive trees. For a singly-recursive tree with:

$Prob(Y_{k,n} = 0) = 1-p$

$Prob(Y_{k,n} = 1) = p$

The number of misses for a given generation of the singly-recursive tree will be the number of nodes of the previous generation multiplied by the probability that a node produces a ray and the probability a ray produced will not generate another node:

$$E(\text{\# misses in a singly-recursive tree}) = \sum_{i=0}^{depth-1} E[X_i] * (1-p) * r$$

Using the result to obtain reflection and transmission ray misses:

Reflection ray misses = $\sum_{g=1}^{MRD} E[X_{g-1}] * \frac{1-p}{p+q} * r$ + (expected number of misses for singly-recursive reflection trees)

Transmission ray misses = $\sum_{g=1}^{MRD} E[X_{g-1}] * \frac{1-q}{p+q} * t$ + (expected number of misses for singly-recursive transmission trees)

6.3 Assumptions, Strengths, and Weaknesses of the Model

Assumptions

- **All objects are closed**
- **All objects are convex**
- **No light sources reside within objects**
- **Camera cannot reside within an object**
- **Camera cannot be completely enclosed by objects**

Strengths

- **Handles arbitrary scenes**
- **Handles arbitrary maximum recursion depths**

Weaknesses

- **Large bounding box error is magnified through model**
- **Relative object surface areas may not be well approximated by volumes**

The model we have developed involves many averages and approximations. Values for the probability of a random ray hitting an object, or the probability of a random object being reflective or transmissive cannot be found with any real degree of accuracy. What we have hoped to capture through the

model, however, is the relative proportions of these probabilities for different scenes. When a scene has a high density of objects, we expect it to have relatively more recursive ray-object intersections. Although the number predicted may not be accurate, the number should be greater than for a scene that is less dense.

Averaging scene reflectivity and transparency is another large source for error. However, if the scene contains enough objects, these values may be good approximations for general recursive behavior.

Using bounding boxes to represent objects creates error that depends on the "fit" of an object to its bounding box. How much empty space is left between the two? We compound this error when using volumes where surface areas may be more appropriate.

In addition, bounding boxes that have extremely small lengths along one axis with respect to the other two will create large error in the model. The surface areas of such bounding boxes will not be well approximated by their volumes. The recursion model does not work well for object groups whose relative surface areas are not well approximated by volumes because we have used volumes in our model where surface areas are more appropriate (e.g. densities of reflective, transmissive, or non-opaque objects).

The real strength to the model is its ability to handle a wide range of scenes. Although there are certain restrictions to where cameras and lights may be placed, these restrictions are not very stringent. Unless a closed room is being rendered, the restrictions will most likely not apply to the scene. Objects that have large surface areas but small volumes can always be broken up into smaller objects with a more desirable ratio.

A comparison of collected results with modeled results at each depth for the recursion model is given in section **9.5**.

7 Results

The model we have developed was used to predict rendering times for eleven scenes. A description of each scene can be found in the appendix, section **9.3**. Section **9.4** contains the baseTime values used for the basic operations during data collection. Keep in mind that the baseTime values for *getObjectsAndLights*, *ComputeEyeRay*, and *RayCast* were determined individually for each scene. All error results in this section are the percentage error of the prediction value from the collected value during rendering.

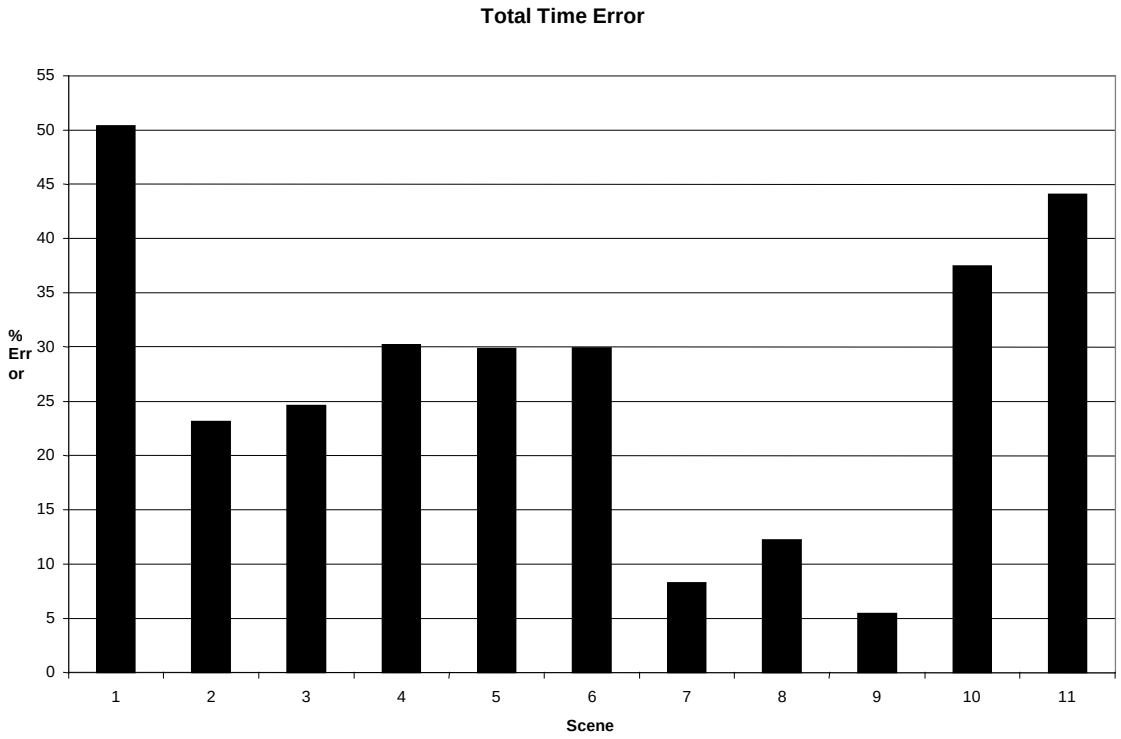

Figure 17: Prediction error of total time spent rendering by scene

Figure 17 displays the error of our prediction for the total time to render each scene. Scene 1 through 6 have large total time error because they suffer from high *RayCast()* baseTime error, as can be seen in Figure 18. The *RayCast()* baseTime for these scenes was predicted to be zero because the timing function used in not accurate enough to capture small time intervals. As a result, the timing function itself

takes up a large percentage of the measured time, artificially raising the baseTime a significant amount. Although the scenes were small enough that *RayCast()* did not account for the majority of the rendering time, it did account for a large percentage. Therefore, its time per call had a large effect on total rendering time.

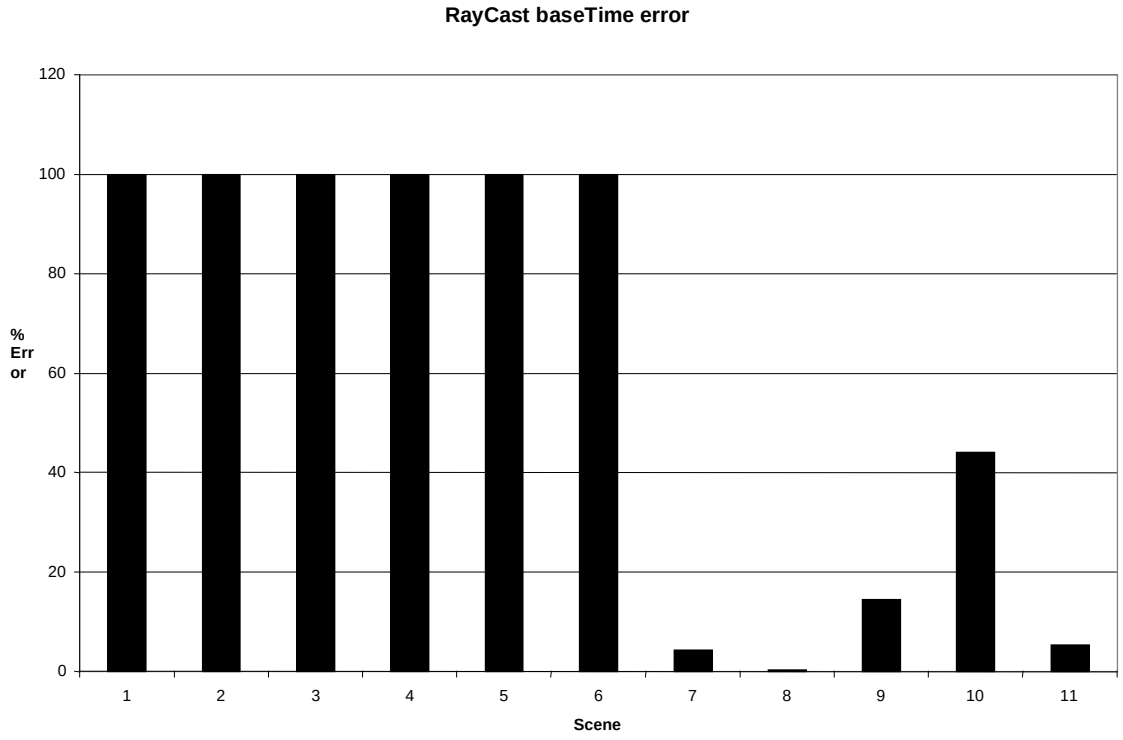

Figure 18: Prediction error of *RayCast()* baseTime by scene

Scene 1 also had high *setPixelImmed()* baseTime error, shown in Figure 19. In small scenes, such as scene 1, *setPixelImmed()* accounts for the majority of time spent in rendering, so its time per call also has a large effect on total rendering time. The effects of this error can be seen in the total time error for scene 1 in Figure 17, which is much higher than scenes 2 through 6.

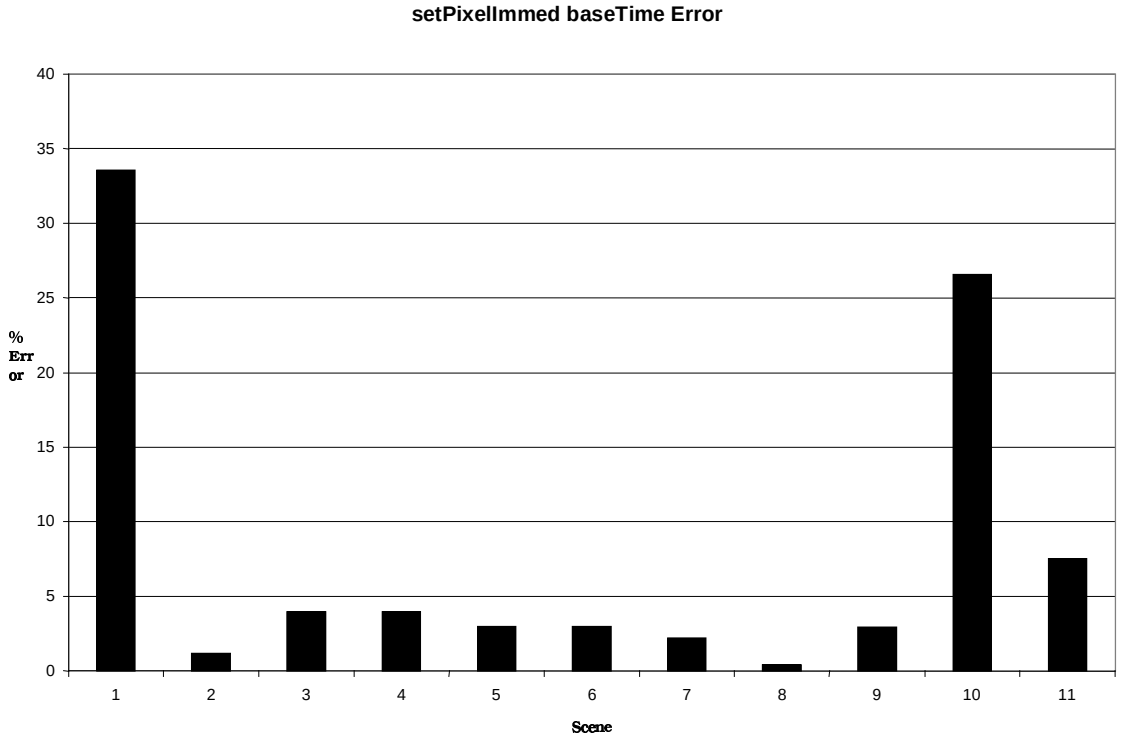

Figure 19: Prediction error of *setPixelImmed()* baseTime by scene

Scene 11 also had a high total rendering time error. The scene, comprised of cylinders and cubes, depicts a Greek temple. The cubes, making up the floor and roof, are large and flat. Due to their shape, the cubes' surface areas cannot be approximated well using their volumes in variable calculation. The cubes in the scenes were transparent, while the cylinders were reflective. The transparent cubes' inaccurate surface area approximations created large error in density of non-opaque objects in the scene, greatly affecting the number of expected shadow rays cast. The result was a 100% error in shadow rays predicted, as can be seen in Figure 23.

Scene 10 incurred high total rendering time error due to its large, highly reflective, highly transparent sphere. The large amount of bounding box volume error for spheres created error in all density calculations. This error had the greatest effect in calculating transmissive object density for transmission ray regeneration probability. All other objects in the scene are reflective, opaque cubes, which brought down volume error for reflective object density calculation. The volume of transmissive objects, however, was greatly overestimated since only the sphere contributed the to value. Figures 22, 25, and 27 demonstrate the high transmission ray prediction error for scene 10. The large reflection ray-object misses

error for scene 10, shown in Figure 26, is mostly likely a result of the layout of the scene. The cubes are tiled as a floor and adjoining wall, creating a contiguous surface that reflects more rays than would be expected simply by examining object density within the scene.

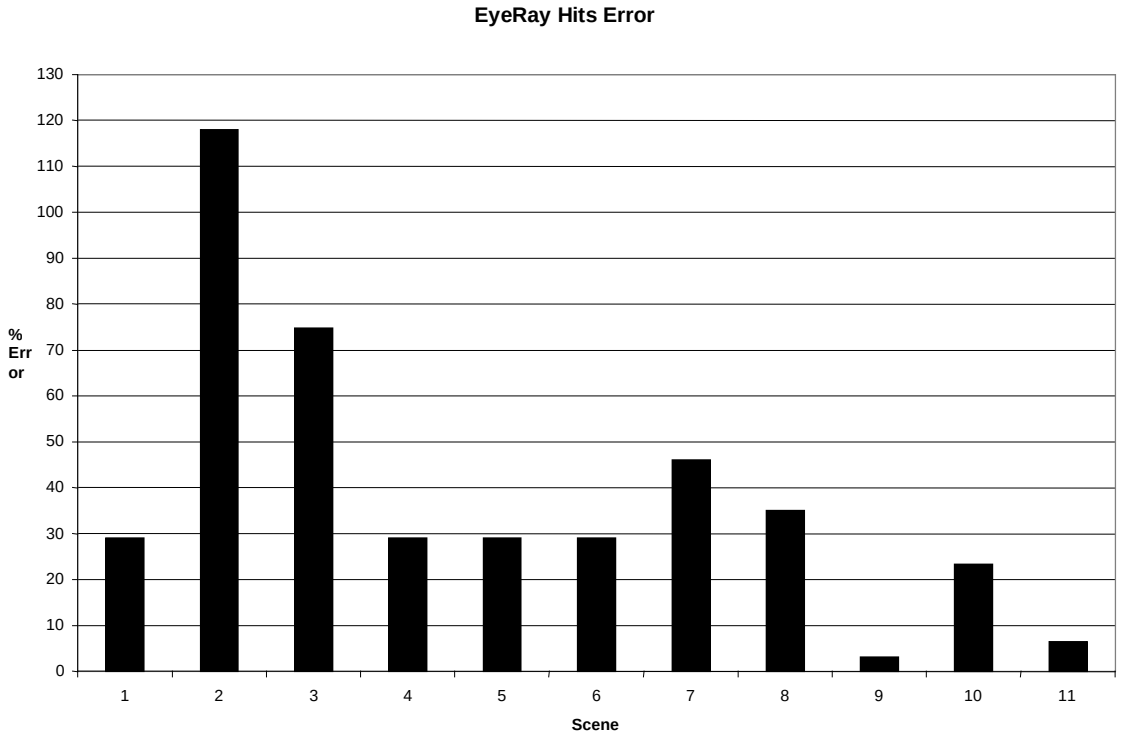

Figure 20: Prediction error of eyeray hits by scene

Examining Figure 20, it can be seen that scenes containing more spheres and cones, and less cubes and cylinders, incurred higher error in predicting eyeray hits. This is due to the fact that bounding box approximations are less accurate for spheres and cones than they are for cubes and cylinders. Scene 2, which consists entirely of spheres, incurred the greatest eyeray hit prediction error.

The error in eyeray hit prediction directly correlates with the error in reflection and transmission ray predictions, as can be seen in Figures 21, 22, 24, 25, 26, and 27. Recall that the number of eyeray hits is used to determine the number of nodes to start with in generation 0 of the reflection/transmission ray tree. The number of reflection/transmission ray tree nodes is then used to determine the number of nodes in generation 0 of the shadow trees. Similar relative error percentages between scenes can be seen for shadow rays in Figure 23 as for the reflection and transmission ray errors.

Scenes 9, 10, and 11, however, had higher errors for transmission ray prediction than would be expected form the eyeray hit prediction error, as seen in Figure 22. These scenes contained fewer transparent objects and/or lower object transparencies. What resulted was higher numbers of predicted transmission rays than were actually generated. The reason is that the transmission ray regeneration constant, empirically determined by the average constant found across the scenes on previous trial renderings, was set higher than it should have been for these scenes. Other scenes required higher transmission regeneration constants, pulling the average up. Scenes 9, 10, and 11, with fewer transparent objects and/or lower object transparencies, were more affected by the error in this constant.

The error in the transmission ray generation constant affected the transmission ray misses prediction for scene 9 very heavily, as can be seen in Figure 27. Far more misses were predicted than occurred. Why the transmission ray misses were more heavily affected than the hits is most likely due simply to the layout of the scene.

In general, reflection ray predictions fared better than transmission ray predictions, and both fared much better than shadow ray predictions. As seen in Figure 23, shadow ray predictions suffered most because the number of initial nodes in all shadow trees was based on the number of nodes in the reflection/transmission ray tree. Therefore, error from the reflection/transmission ray tree was carried over and compounded through the shadow ray tree model, adding to the unique error sources of shadow ray trees.

The relative errors of reflection and transmission ray prediction were most likely due to differing errors in regeneration constants. The constants' errors arose due to differing scene layouts and object reflection and transmission properties. Examining the model, we would have expected higher error in the reflection ray prediction because of reflection rays that bounce endlessly around inside of objects that are both reflective and transparent. Apparently, the empirical reflection ray generation constant captured this effect very well.

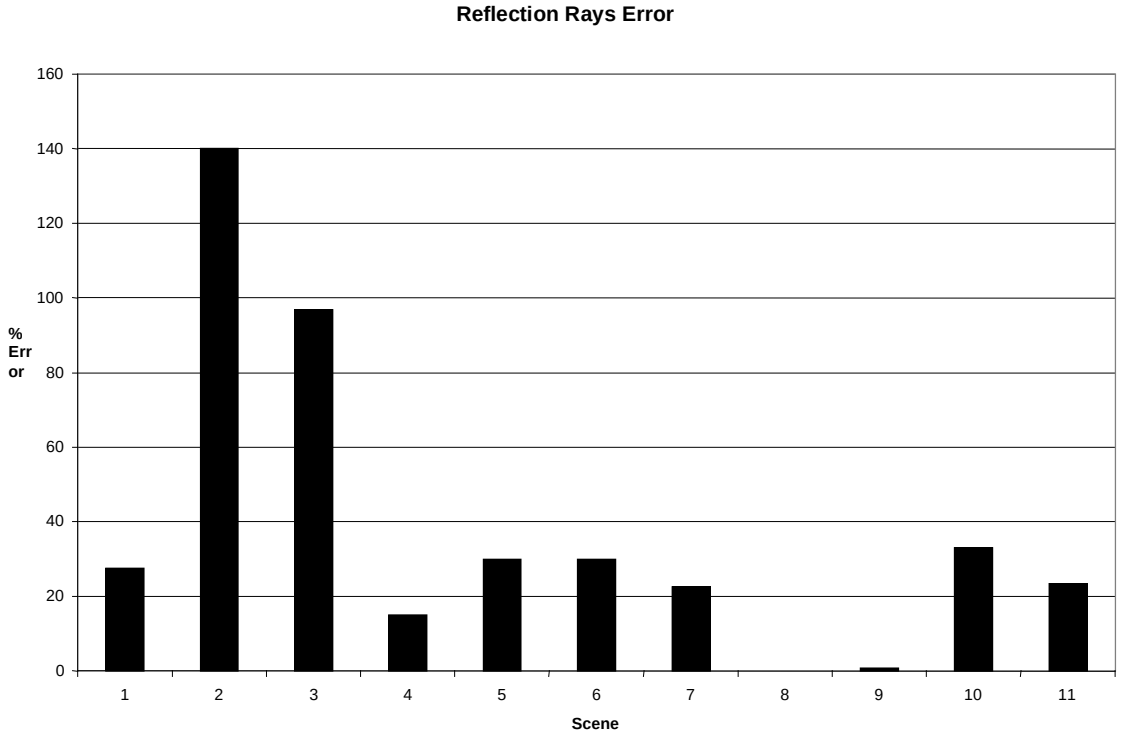

Figure 21: Prediction error of reflection rays cast by scene

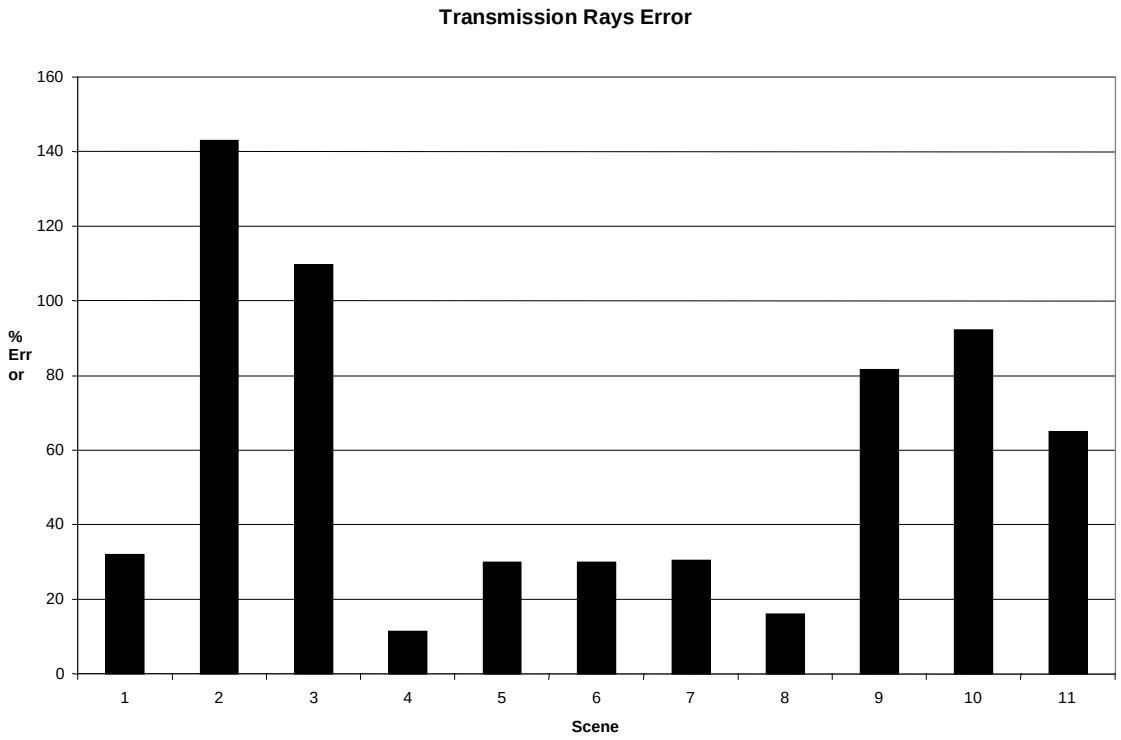

Figure 22: Prediction error of transmission rays cast by scene

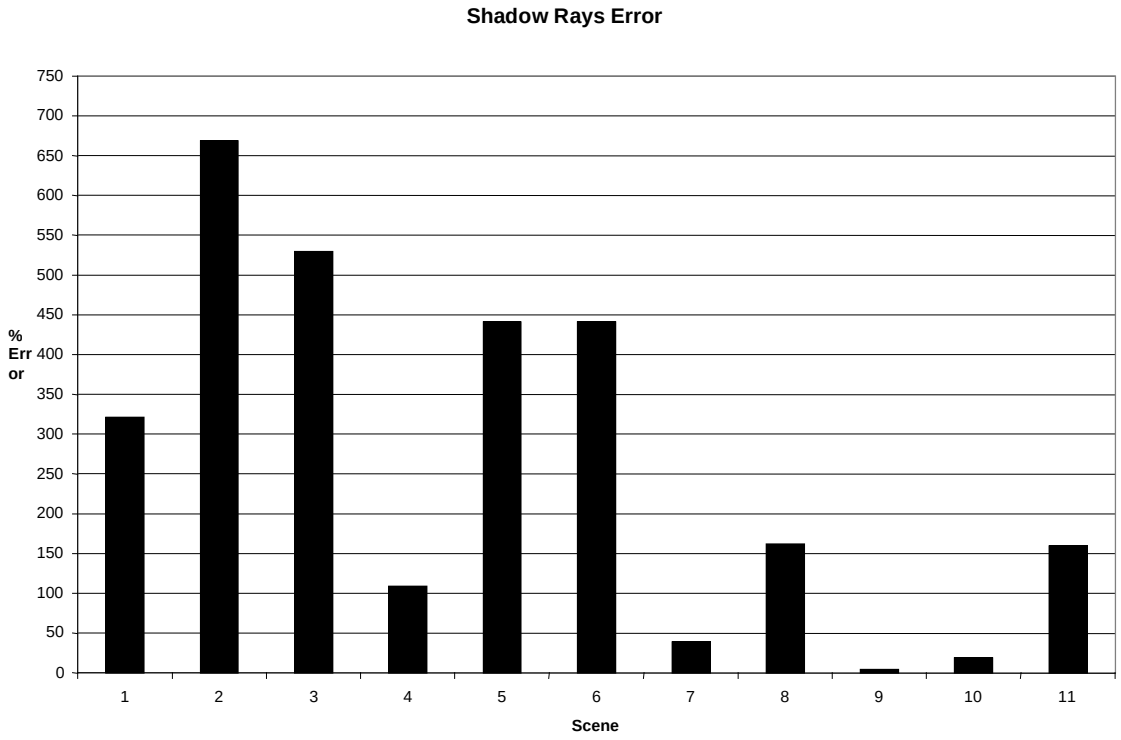

Figure 23: Prediction error of shadow rays cast by scene

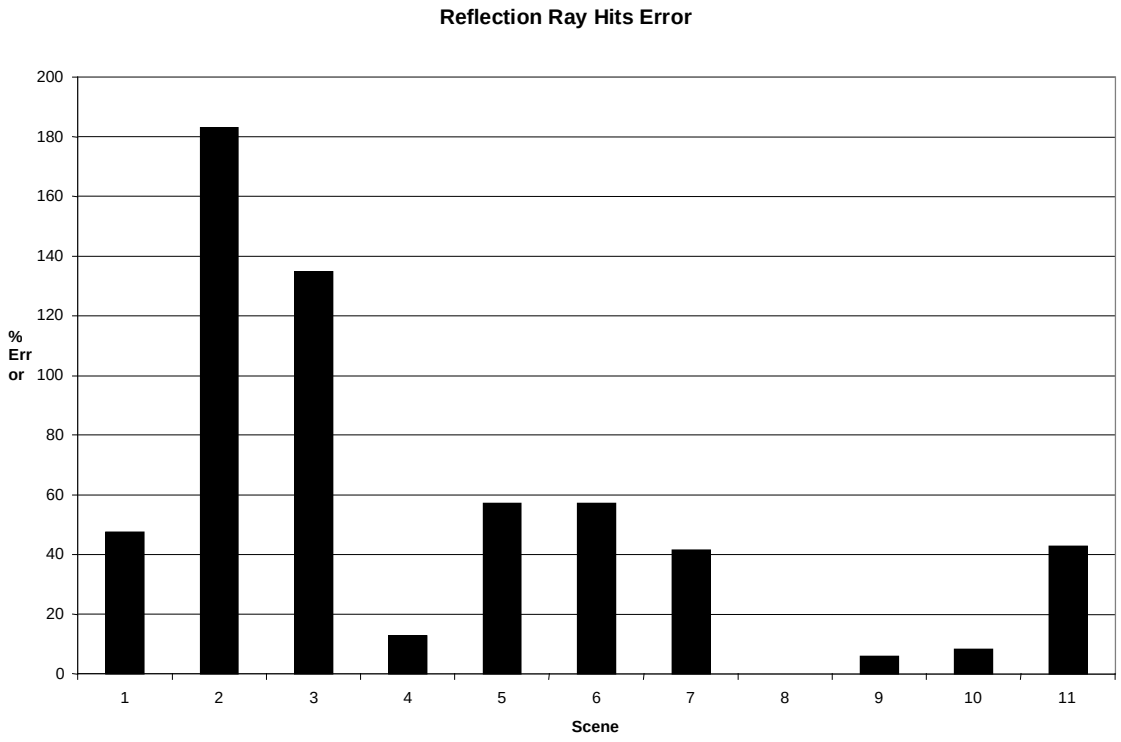

Figure 24: Prediction error of reflection ray-object hits by scene

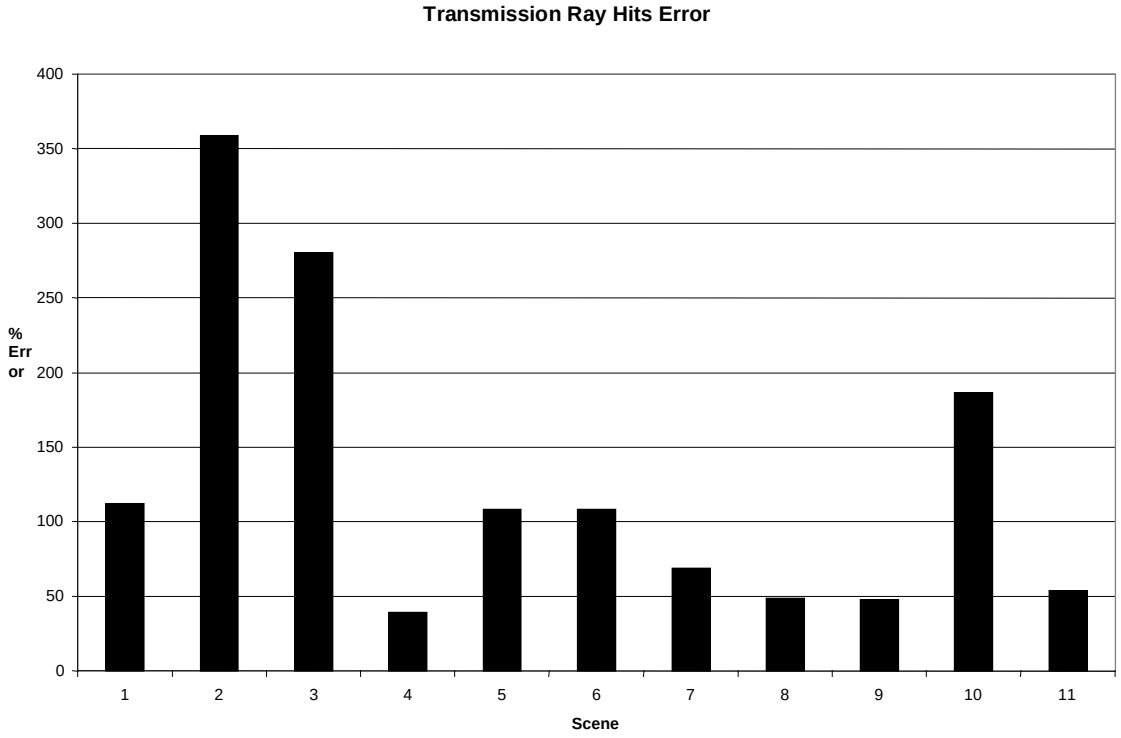

Figure 25: Prediction error of transmission ray-object hits by scene

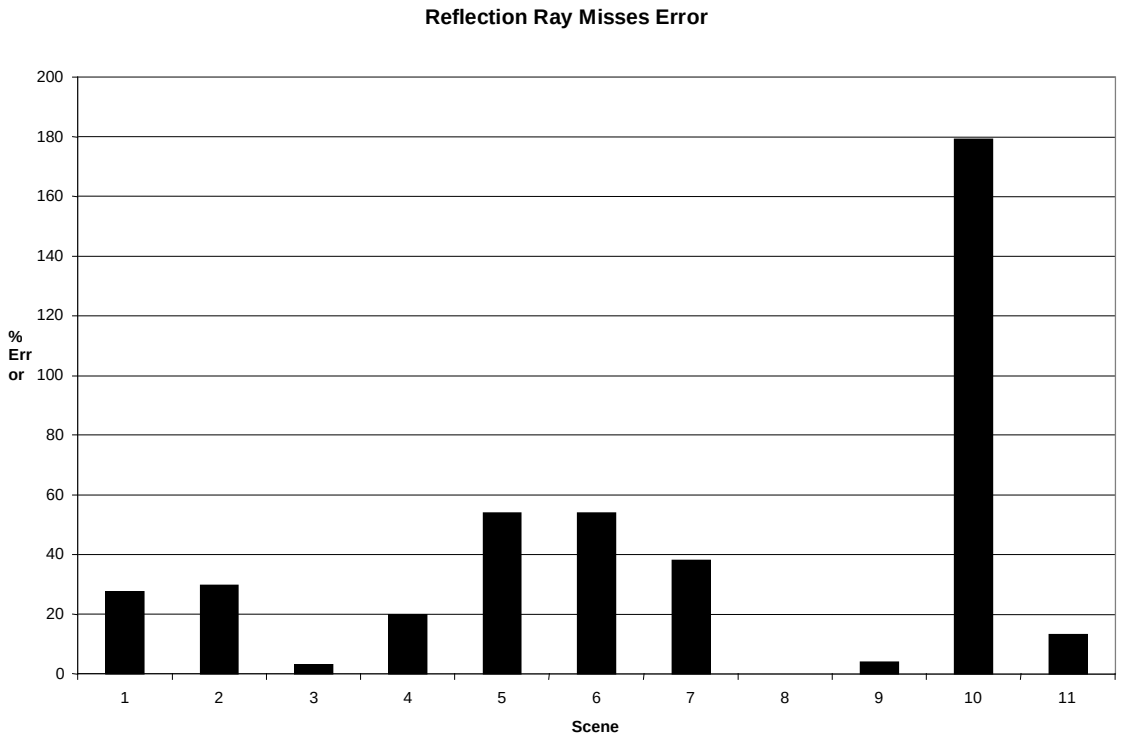

Figure 26: Prediction error of reflection ray-object misses by scene

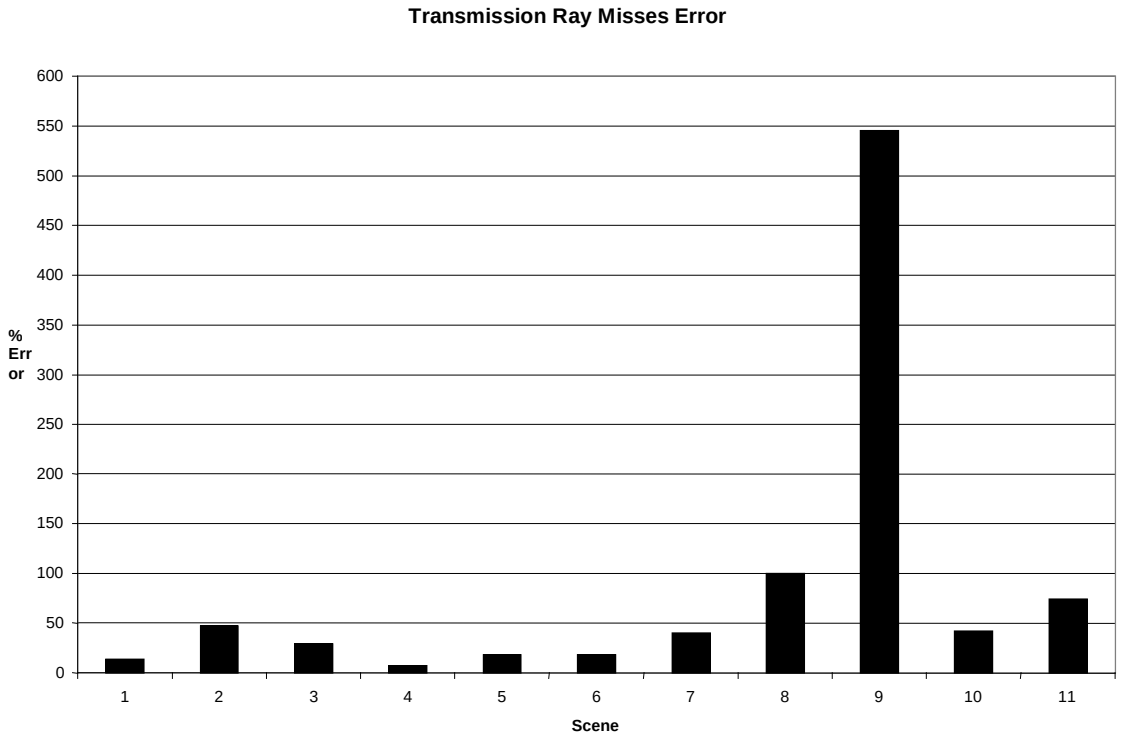

Figure 27: Prediction error of transmission ray-object misses by scene

8 Conclusions

The error in rendering time predictions for the sample set of scenes ranged from 5.5% to 50.4%. Error sources included object space bounding box error, screen space bounding box error, and approximating surface areas with volumes. Determining ray-object intersection probabilities was the greatest source for error as a result of the constants of proportionality.

One way to reduce error would be to collect many possible values for constants of proportionality in ray-object intersection probabilities. Each value would be used for the scene whose characteristics most closely matched those used to determine the constant empirically. Another way to reduce error would be to pre-compute object surface areas to be used in place of object bounding box volumes. Screen space bounding box error could be eliminated by casting all eyerays, then predicting recursive ray generation. The number of nodes in the 0^{th} generation of the reflection/transmission branching process would be accurate, although casting eyerays is a time consuming procedure. This method would only be useful in scenes that generate far more recursive rays than eyerays, making eyeray computation time small in comparison to total rendering time.

As it is enhanced, the prediction model will be a valuable tool in coordinating large rendering projects, as well as designing scenes to maximize quality while minimizing rendering time. The latter application is especially important for interactive ray traced applications which require a guaranteed frame rate of at least 20 frames/sec. To guarantee this frame rate, image quality must be sacrificed. Special rendering effects such as reflection and refraction may have to be cut out or fewer samples may have to be computed per pixel. To choose which piece of the computation to truncate or which algorithms and data structures to use, the application must have an idea not only of what the user values in image computation, but also how much time each computation would take. That way, the application can decide which aspects of computation are worth cutting out or changing slightly because they would save the most time while maximizing user preference and image quality.

9 Appendices

9.1 baseTime Reconstruction by Basic Operation

What follows is a list of how the baseTime is found for each of the eighteen basic operations.

getObjectsAndLights

The time spent varies greatly from scene to scene.

MakeJitter

The time spent is constant across calls.

ComputeEyeRay

The time spent varies greatly from scene to scene.

RayCast

The time spent varies greatly from scene to scene.

Shade

Calls take longer if the argument ray intersects an object.

BackgroundMap

The time spent is constant across calls.

ReflectionDirection

The time spent is constant across calls.

TransmissionDirection

The time spent is constant across calls.

ReflectionRadiance

Calls take longer if the current weight of the ray is above the user-set minimum ray weight.

TransmissionRadiance

Calls take longer if the current weight of the ray is above the user-set minimum ray weight.

Shadowing

The function is mostly made up of a loop. The number of times the loop is run is equal to the number of non-opaque objects between the point we are shadowing and the light given as an argument. We will predict this number in the next section.

AngularAttenuation

The time spent is constant across calls.

DistanceAttenuation

The time spent is constant across calls.

SpecularRadiance

The time spent is constant across calls.

DiffuseRadiance

The time spent is constant across calls.

AmbientRadiance

The time spent is constant across calls.

EmissionRadiance

The time spent is constant across calls.

setPixelImmed

The time spent is constant across calls.

9.2 Count Reconstruction by Basic Operation

What follows is a list of how the number of calls in each computation segment is found for the eighteen basic operations.

getObjectsAndLights

This function is called only once in the IMPORT segment of computation, regardless of the scene being rendered.

MakeJitter

If samples are jittered non-randomly, this function is called once for the scene to determine grid pattern of samples within each pixel. This call is considered part of the BUILD segment of computation.

If samples are jittered randomly, this function is called once for each pixel to determine sample positions. These calls are considered part of the COMPUTEEYERAY segment of computation.

ComputeEyeRay

This function is called once for each ray cast from the camera, through the image plane, into the scene. There is one call in the COMPUTEEYERAY segment of computation for each sample taken.

RayCast

This function is called once for each ray cast of every type. It is called for every eyeray in the QUERYEYERAY segment of computation. It is called for every shadow ray in the QUERYSHADOWRAY segment of computation. It is called for every reflection ray in the REFLECTION segment of computation. It is called for every transmission ray in the REFRACTION segment of computation.

Shade

This function is called once in the SHADINGMODEL segment of computation for each sample. *Shade* is called once in the REFLECTION segment of computation for each reflection ray cast, and once in the REFRACTION segment of computation for each transmission ray cast.

BackgroundMap

This function is called once for every call to *shade* whose ray argument has not intersected any object. That means it is called once for every ray from the camera through the image plane without a hit. These are part of the SHADINGMODEL segment of computation. *BackgroundMap* is also called once in the REFLECTION segment of computation for every reflection ray without a hit, and once in the REFRACTION segment of computation for every transmission ray without a hit.

ReflectionDirection

This function is called once in the REFLECTION segment of computation for every reflection ray cast.

TransmissionDirection

This function is called once in the REFRACTION segment of computation for every transmission ray cast.

ReflectionRadiance

This function is called once in the REFLECTION segment of computation for every call to *shade* whose ray argument has intersected an object.

TransmissionRadiance

This function is called once in the REFRACTION segment of computation for every call to *shade* whose ray argument has intersected an object.

Shadowing

In the COMPUTEEYERAY segment of computation, this function is called once for every light when *shade* is called with a hit. In the REFLECTION segment of computation, this function is called once for every light when a reflection ray hits an object. In the REFRACTION segment of computation, this function is called once for every light when a transmission ray hits an object.

AngularAttenuation

In the COMPUTEEYERAY segment of computation, this function is called once for every spot light when *shade* is called with a hit that is not in shadow. In the REFLECTION segment of computation, this function is called once for every spot light when a reflection ray hits an object and that hit is not in shadow. In the REFRACTION segment of computation, this function is called once for every spot light when a transmission ray hits an object and that hit is not in shadow.

DistanceAttenuation

In the COMPUTEEYERAY segment of computation, this function is called once for every local and spot light when *shade* is called with a hit that is not in shadow. In the REFLECTION segment of computation, this function is called once for every local and spot light when a reflection ray hits an object and that hit is not in shadow. In the REFRACTION segment of computation, this function is called once for every local and spot light when a transmission ray hits an object and that hit is not in shadow.

DiffuseRadiance

In the COMPUTEEYERAY segment of computation, this function is called once for every light when *shade* is called with a hit that is not in shadow. In the REFLECTION segment of computation, this function is called once for every light when a reflection ray hits an object and that hit is not in shadow. In the REFRACTION segment of computation, this function is called once for every light when a transmission ray hits an object and that hit is not in shadow.

SpecularRadiance

In the COMPUTEEYERAY segment of computation, this function is called once for every light when *shade* is called with a hit that is not in shadow. In the REFLECTION segment of computation, this function is called once for every light when a reflection ray hits an object and that hit is not in shadow. In the REFRACTION segment of computation, this function is called once for every light when a transmission ray hits an object and that hit is not in shadow.

AmbientRadiance

In the COMPUTEEYERAY segment of computation, this function is called once every time *shade* is called with a hit. In the REFLECTION segment of computation, this function is called once every time a reflection ray hits an object. In the REFRACTION segment of computation, this function is called once every time a transmission ray hits an object.

EmissionRadiance

In the COMPUTEEYERAY segment of computation, this function is called once every time *shade* is called with a hit. In the REFLECTION segment of computation, this function is called once every time a reflection ray hits an object. In the REFRACTION segment of computation, this function is called once every time a transmission ray hits an object.

setPixelImmed

This function is called once for every pixel to be displayed.

9.3 Scenes Used to Collect Data

All scenes were rendered with the following settings:

Number of Samples:	93564
Jittering:	non-random grid
Maximum Ray Depth:	4
Minimum Ray Weight:	0.01

Scene 1

The scene is comprised of six cubes of varying reflectivity and transparency, and three light sources. Objects are loosely distributed in space.

The total measured rendering time was 14.39 seconds.

Scene 2

The scene is comprised of six spheres of varying reflectivity and transparency, and three light sources. Objects are loosely distributed in space.

The total measured rendering time was 10.93 seconds.

Scene 3

The scene is comprised of two cubes, two spheres, and two cones of varying reflectivity and transparency, and three light sources. Objects are distributed with low density.

The total measured rendering time was 11.21 seconds.

Scene 4

The scene is comprised of six cubes and three light sources. The cubes had high reflectivity values and low transparency values. Objects are loosely distributed in space.

The total measured rendering time was 11.31 seconds.

Scene 5

The scene is comprised of six cubes and three light sources. The cubes had low reflectivity values and high transparency values. Objects are loosely distributed in space.

The total measured rendering time was 12.3 seconds.

Scene 6

The scene is comprised of six cubes and three light sources. The cubes had high reflectivity values and high transparency values. Objects loosely distributed in space.

The total measured rendering time was 12.41 seconds.

Scene 7

The scene is comprised of 43 objects of varying reflectivity and transparency, and one light source. Object types include spheres, cubes, cylinders, spheres, and face sets, and are closely packed in space.

The total measured rendering time was 54.7 seconds.

Scene 8

The scene is comprised of 29 objects of varying reflectivity and transparency, and one light source. Object types include spheres, cubes, cones, cylinders, and face sets. Objects are loosely distributed in space.

The total measured rendering time was 20.43 seconds.

Scene 9

The scene is comprised of 14 objects of varying reflectivity and transparency, and one light source. Object types include spheres, cubes, cones, and cylinders. Objects are closely packed and transparencies of objects are generally low.

The total measured rendering time was 17.36 seconds.

Scene 10

The scene is comprised of 19 objects and one light source. One of the objects is a large sphere that is both highly reflective and highly transparent. The other objects are opaque, non-reflective cubes. Objects are closely packed in space.

The total measured rendering time was 28.01 seconds.

Scene 11

The scene is comprised of 26 objects and one light source. Most of the objects are opaque, reflective cylinders. Three of the objects are transparent, reflective cubes. Objects are closely packed in space.

The total measured rendering time was 30.48 seconds.

9.4 baseTime Results

Here are the baseTimes for each basic operation. The times were found empirically using the scenes in section **9.3**. Time values are in seconds. Keep in mind that the baseTime values for *getObjectsAndLights*, *ComputeEyeRay*, and *RayCast* were determined individually for each scene.

```
MakeJitter_baseTime_reg = 0.000000
MakeJitter_baseTime_rand = 0.000000
Shade_baseTime = 0.000004
BackgroundMap_baseTime = 0.000003
ReflectionDirection_baseTime = 0.000002
TransmissionDirection_baseTime = 0.000001
ReflectionRadiance_baseTime = 0.000003
TransmissionRadiance_baseTime = 0.000001
Shadowing_loopTime = 0.000001
AngularAttenuation_baseTime = 0.000000
DistanceAttenuation_baseTime = 0.000000
DiffuseRadiance_baseTime = 0.000004
SpecularRadiance_baseTime = 0.000007
AmbientRadiance_baseTime = 0.000001
EmissionRadiance_baseTime = 0.000000
setPixelImm_baseTime = 0.000069
```

9.5 Recursive Ray Cast Data by Depth

Figures 28 through 39 present data on collected and predicted recursive ray casts for each scene. Each figure gives the numbers of reflection and transmission rays cast at each depth of the recursive process. The predicted numbers are higher in general due to the screen space bounding box error, which affects the 0^{th} generation of the recursive branching process.

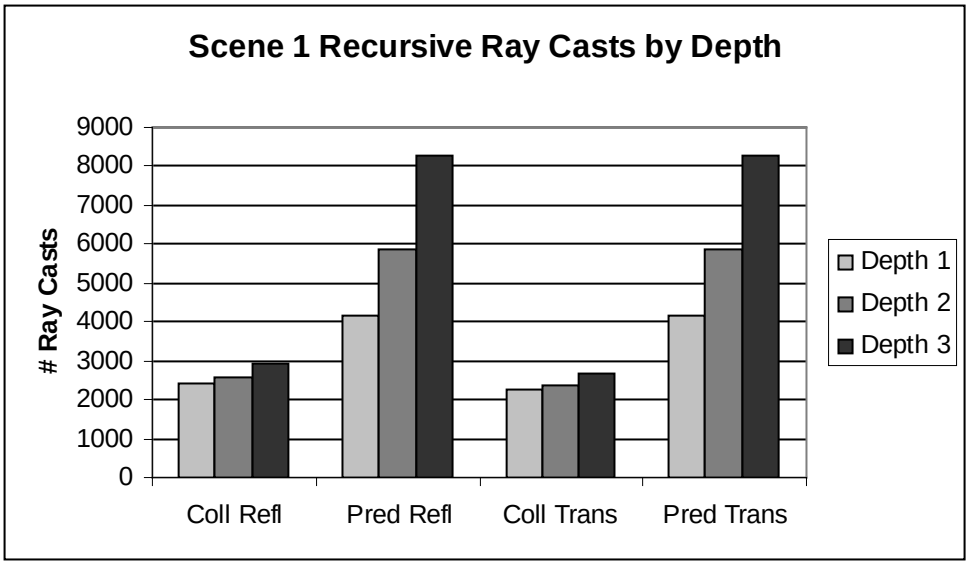

Figure 28: Scene 1 reflection and transmission ray casts

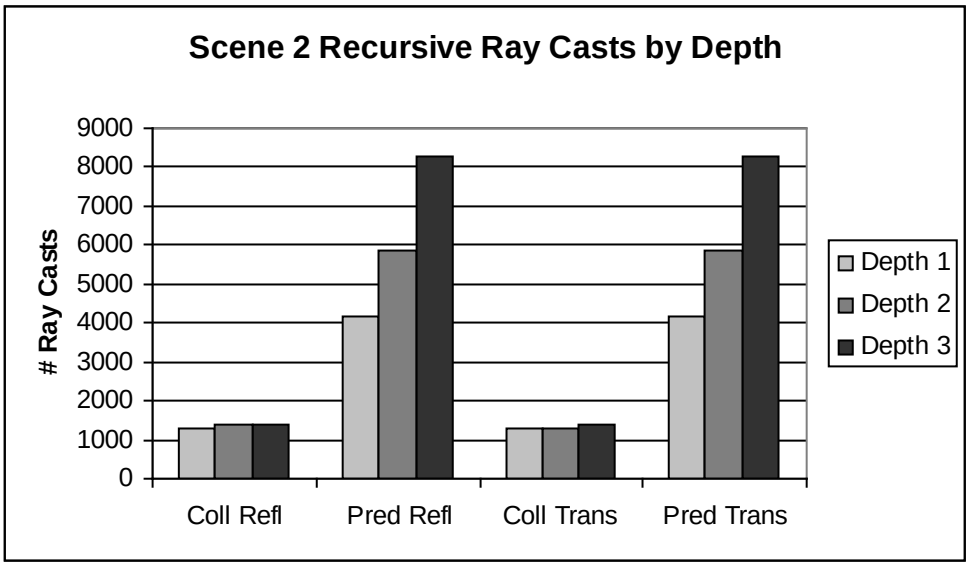

Figure 29: Scene 2 reflection and transmission ray casts

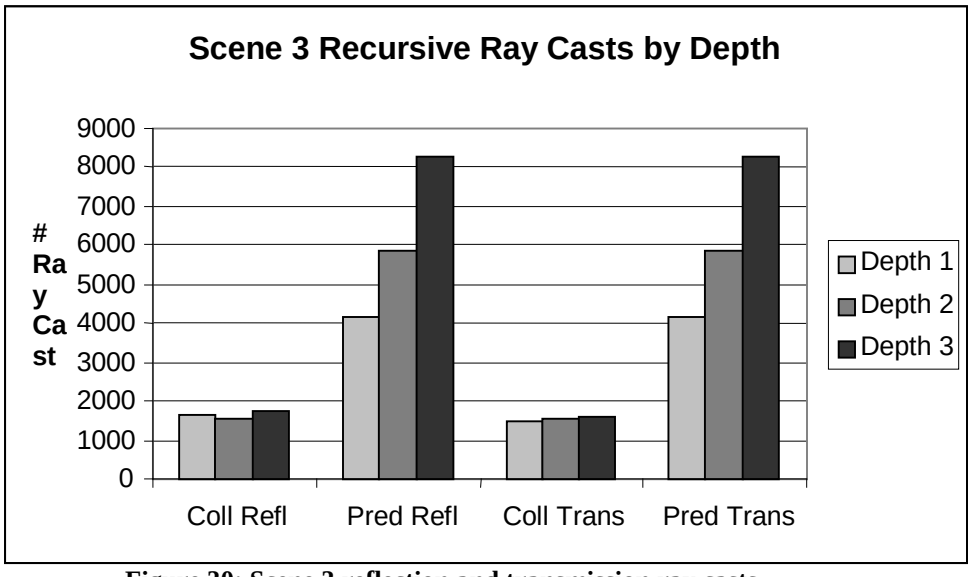

Figure 30: Scene 3 reflection and transmission ray casts

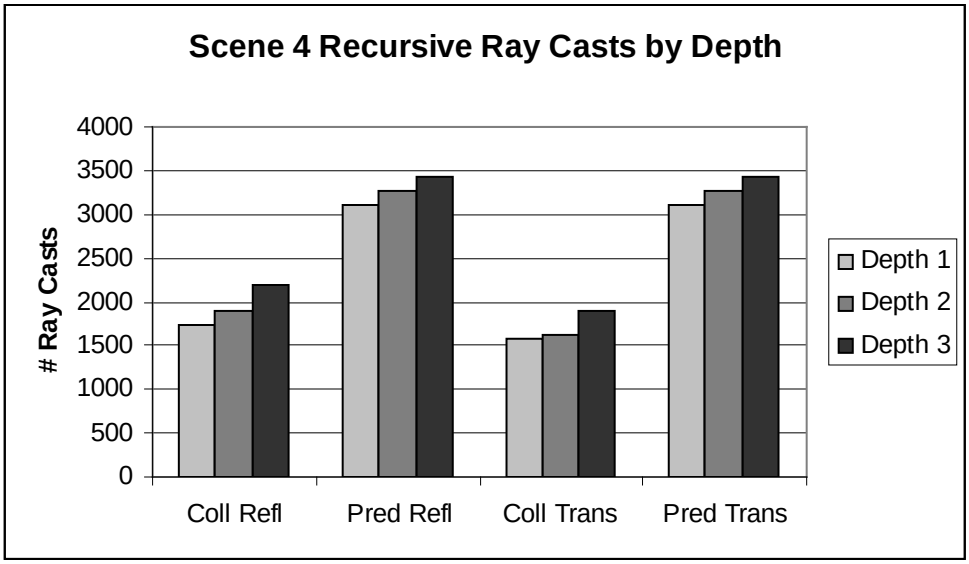

Figure 31: Scene 4 reflection and transmission ray casts

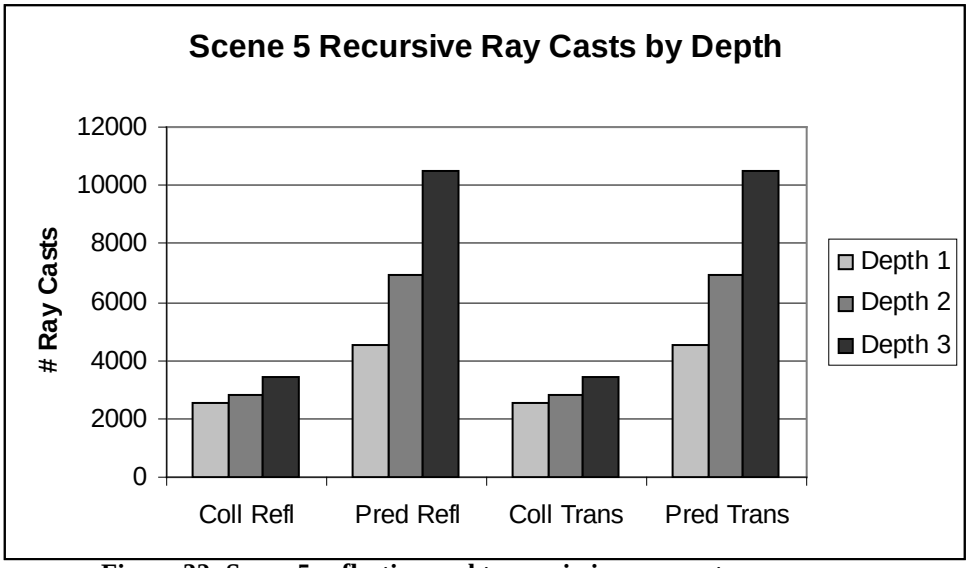

Figure 32: Scene 5 reflection and transmission ray casts

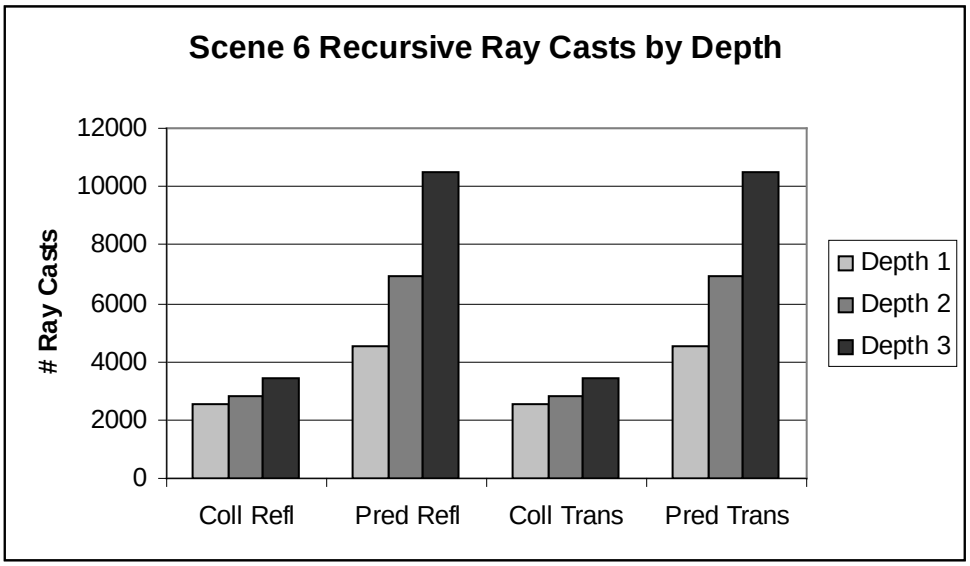

Figure 33: Scene 6 reflection and transmission ray casts

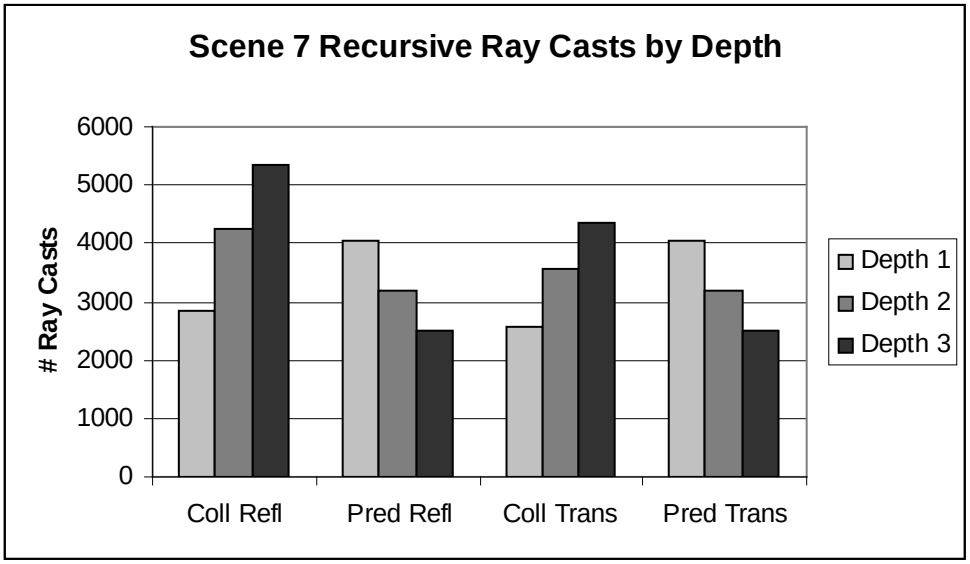

Figure 34: Scene 7 reflection and transmission ray casts

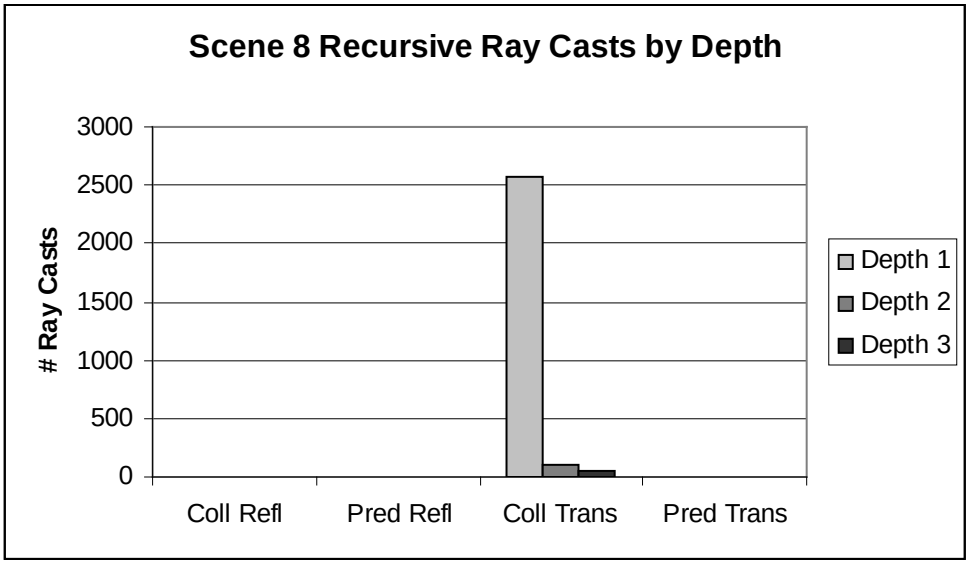

Figure 35: Scene 8 reflection and transmission ray casts

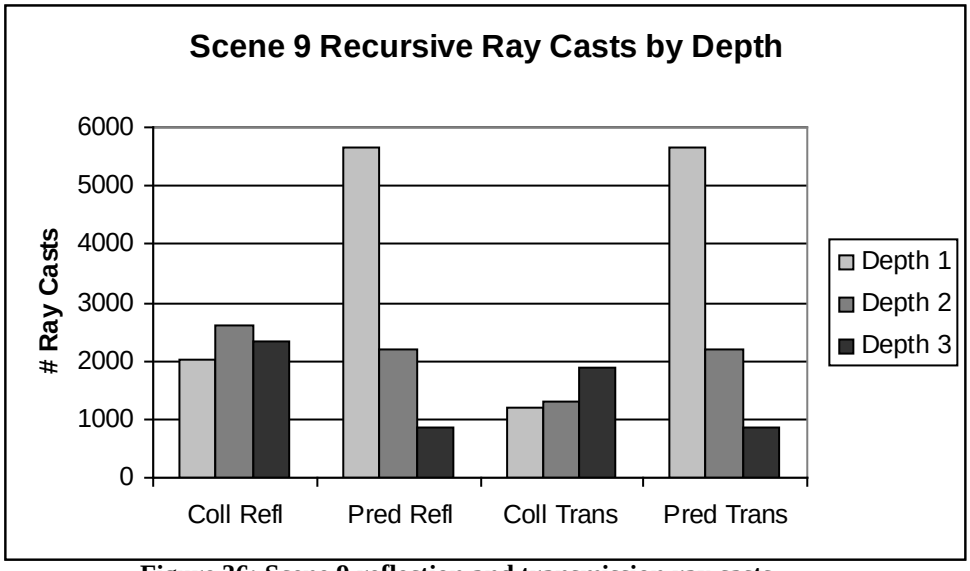

Figure 36: Scene 9 reflection and transmission ray casts

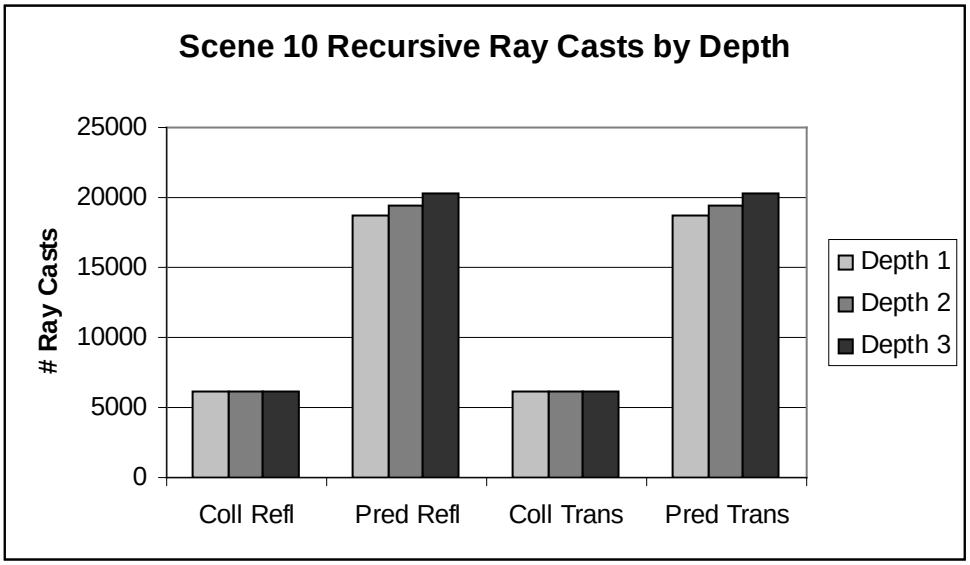

Figure 37: Scene 10 reflection and transmission ray casts

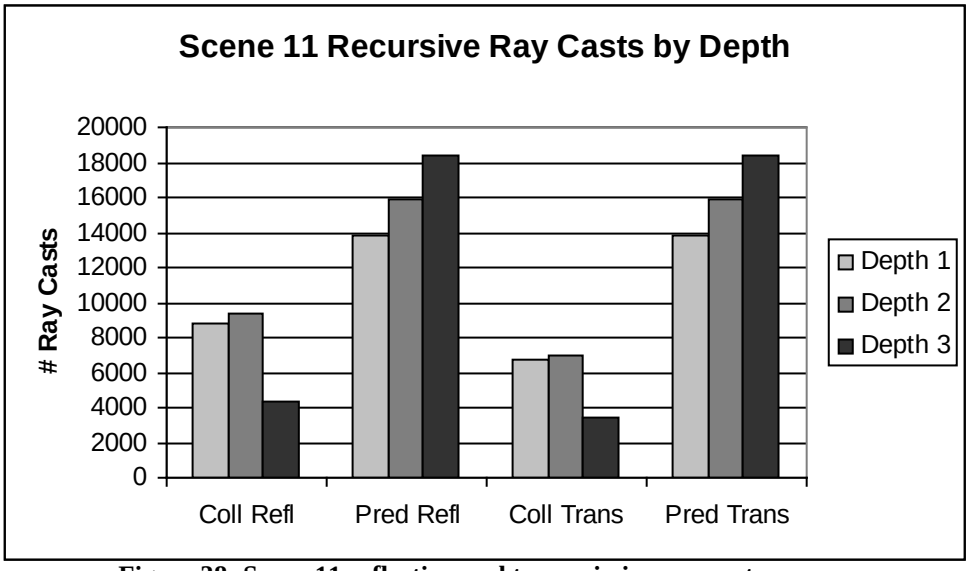

Figure 38: Scene 11 reflection and transmission ray casts

10 Bibliography

Arvo, James and David Kirk, "A Survey of Ray Tracing Acceleration Techniques," An Introduction to Ray Tracing, Andrew S. Glassner, ed., Academic Press, Boston, 1989, p 203.

Fujimoto, Akira, Takayuki Tanaka, and Kansei Iwata, Graphica Computer Corporation, "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications*, April, 1986, pp 16-26.

Haines, Eric, "Comparison of Kolb, Haines, and MTV Ray Tracers, Part I," January 2, 1990, RTNews, Vol. 3, Number 1, http://www.acm.org/tog/resources/RTNews/html/rtnv3n1.html#art10.