Spring 2014

Path tracing using lower level of detail for secondary rays

Ryan Irby Wolter
James Madison University

Path Tracing using Lower Level of Detail for Secondary Rays

A Project Presented to

the Faculty of the Undergraduate

College of Integrated Science and Engineering

James Madison University

---

in Partial Fulfillment of the Requirements

for the Degree of Bachelor of Science

---

by Ryan Irby Wolter

May 2014

Accepted by the faculty of the Department of Computer Science, James Madison University, in partial fulfillment of the requirements for the Degree of Bachelor of Science.

FACULTY COMMITTEE:

Project Advisor: David Bernstein, Ph. D.,
Professor, Computer Science

Reader: Chris Mayfield, Ph. D.,
Assistant Professor, Computer Science

Reader: Nathan Sprague, Ph. D.,
Assistant Professor, Computer Science

HONORS PROGRAM APPROVAL:

Barry Falk, Ph.D.,
Director, Honors Program

Table of Contents

List of Figures

Acknowledgements

Special thanks to Peter Shirley for his path tracing code supplied in his book, *Realistic Ray Tracing*, which served as the base for the renderer used in this thesis. Also thanks to the artists from _idst who modeled the Renaissance Courtyard, and the anonymous artists who modeled the VW Touareg, Potted Plant, and Virgin Mary with Baby Statue. Thanks to Dr. Chris Mayfield and Dr. Nathan Sprague for being readers on this thesis, and Dr. David Bernstein for advising on this project. Lastly, thanks to the JMU Honors Program for providing funding for this thesis.

Path Tracing using Lower Level of Detail Models to Increase Performance

Abstract

Path tracing is a computationally expensive method of three dimensional rendering that aims to accurately simulate the propagation of light. A large amount of time is typically spent calculating intersections between rays and the scene, which is composed of triangular meshes stored in some form of bounding volume. This time can be reduced by lowering the overall number of triangles in the scene. Path tracing works by casting rays from the camera into the scene, reflecting until they hit a light source. Secondary rays, or rays which occur after the first intersection, usually contribute less to the overall image, yet require much more time to calculate than primary rays. This thesis found that significant performance gains can be made by using lower level of detail (LOD) triangular meshes for secondary rays. While the lower LOD models are less accurate, they still provide a good approximation of the mesh for secondary rays. Scenes with 1.4 million faces could be rendered up to 10% faster using a 1/32 ratio level of detail for secondary rays. A study with 14 subjects who ranked images based on image quality showed they were unable to differentiate between low LOD and full LOD images.

1. Background

This section provides an overview of different methods of three dimensional rendering, as well as provide information on related works.

1.1 Introduction

Rendering is the process of creating 2-dimensional (2D) images from a digital 3-dimensional (3D) scene. A typical scene consists of lights, objects, and a camera which specifies from where in the scene the view should be made. There are many different methods of representing objects digitally, but one that is quite common is to use triangle meshes. Each corner of a triangle consists of a vertex. The flat plane that connects all three vertices is called a face. Any object's surface can be approximated using an arrangement of multiple triangles. Figure 1 shows a cube with 12 triangles, two triangles for each face. Thousands more of these triangles can be arranged in various ways to produce more complex objects.

Using more triangles, a 3D artist can create higher detailed representations of objects. Some objects, such as spheres, can never be perfectly modeled this way because faces are always flat, while spheres are perfectly smooth. However, using more faces can provide a better approximation. Figure 2 shows two different representations of a sphere side by side. The blue one (left) has only 80 faces, while the green one (right) has 1280 faces, and as a result looks

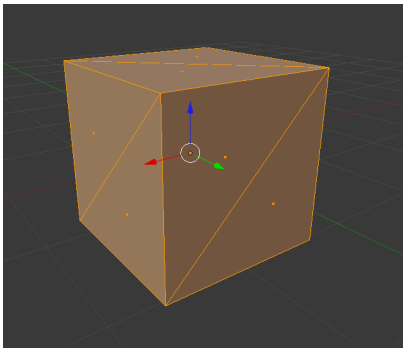

Figure 1: A cube with 12 triangles.

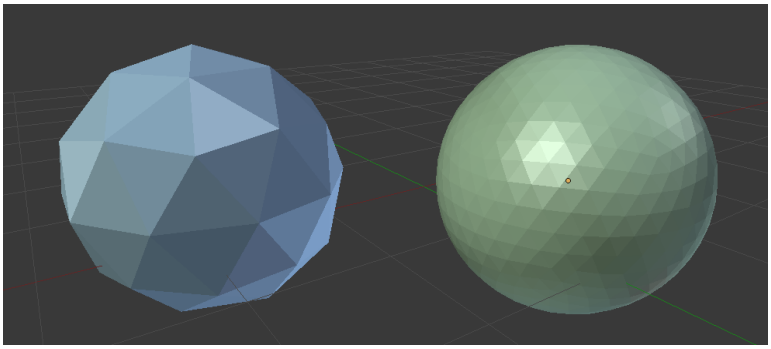

Figure 2: The blue sphere (left) has 80 faces, while the green sphere (right) has 1280. Notice how much more sphere like the green sphere appears, due to the higher level of detail mesh.

more realistic. An artist could make all of their models with hundreds of thousands of faces, but more faces require more memory to store and more time to process. Therefore most artists try to use as few faces as possible while still being accurate to they object the want to model.

There are many ways these objects, stored as triangle meshes, are rendered into images to be viewed. Traditional computer graphics are rendered using the *pipeline method*. A pipeline is a way of processing data sequentially from beginning to end, applying operations at each step. While rendering pipelines can vary depending on the implementation, they can usually be broken into three major steps. The first step is called vertex processing, where the vertices of the triangle meshes are transformed into position using rotations, translations (moving around), and scaling (stretching or shrinking) (Teschner n.d.). The positions are based on where objects are described to be in a scene, as well as the location of the camera. The second step is the primitive assembly stage, where the vertices are connected into faces as described by the triangle mesh (Teschner n.d.). Once connected, these faces are sent to the rasterizer, which determines what pixels the triangle occupies (Lobao 2009). These pixels will then be saved to an image or displayed on a screen, producing the final output. The rendering pipeline is a very fast method of rendering capable of real-time performance, meaning it can render enough images per second to display movement without flicker. The pipeline is fast because graphics cards employ tens or hundreds of special purpose processors designed to work in parallel (Hughes 2013). The pipeline also uses a very simple model of light, which does not model most reflections and often includes crude rigid shadows. While the pipeline method is fast, it does not naturally provide physically accurate lighting and shadows.

Ray tracing is an alternative method of rendering which attempts to model the propagation of light particles to simulate natural lighting. Light in nature has the properties of both waves and particles. For the purpose of this discussion, light can be described as a particle

called a photon. Photons have a position, direction of propagation, and a wavelength, which describes its color (Shirley 2003). Photons travel from a source like a light bulb out into the world, and are either absorbed or reflected by objects. The material properties of objects determines the likelihood of a photon being absorbed or reflected, as well as the direction it reflects. These materials are mathematically simulated in ray tracers to mimic the real world behavior exhibited by these objects. One of the limitations of simple ray tracing is that it can only simulate direct lighting, which is light that travels directly from a light source to an object. Direct lighting alone yields a very dark and unrealistic scene, and thus simple ray tracing does not provide a complete model of light.

Global illumination (GI) or indirect diffuse lighting occurs from the reflections of light photons of nearby illuminated objects. A good example of GI is holding a buttercup up to one's chin, and noticing added yellow coloration of their skin from the reflected light of the flower. Such lighting adds subtle details that are typically sacrificed for speed in real time applications, but can significantly increase the level of realism of the rendering. GI is very computationally expensive because by its nature all the light cast on a given point of a surface from every single other illuminated object in the scene must be summed (Snell 1997).

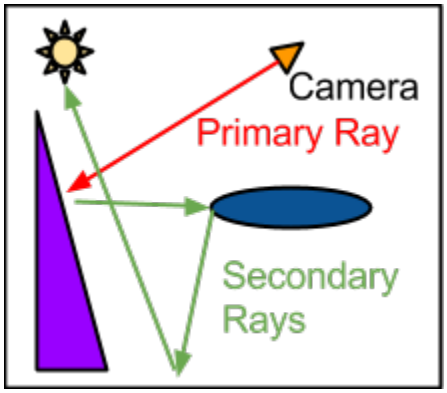

Figure 3: This image illustrates the propagation of light in a path tracing renderer. Notice the distinction between primary and secondary rays.

Path tracing is a form of ray tracing that models GI by tracing rays backwards from the camera into the scene, reflecting from one surface to the next until they finally hit a light source. This reflection path is demonstrated in Figure 3. The direction a ray reflects is determined in part by the materials it intersects with, and in part by randomness. The use of randomness to converge to a solution is called Monte Carlo integration. Monte Carlo integration works by tracing multiple rays and averaging their results to sample the light hitting a single point. Using only a few rays per pixel does not provide enough information to converge to a solution, thus causing a lot of noise, as seen in Figure 4. The solution is to use more than one ray for each pixel, increasing the sample size. An image which uses 160 rays per pixel is a 160 samples per pixel image. While more samples will decrease noise, they also take longer to produce a final image. Path tracing using Monte Carlo integration produces photo-realistic images, but requires a significant amount of time to produce a single image.

Researchers have spent much time creating efficient data structures and algorithms for global illumination to speed up the rendering process. One such structure is a bounding volume,

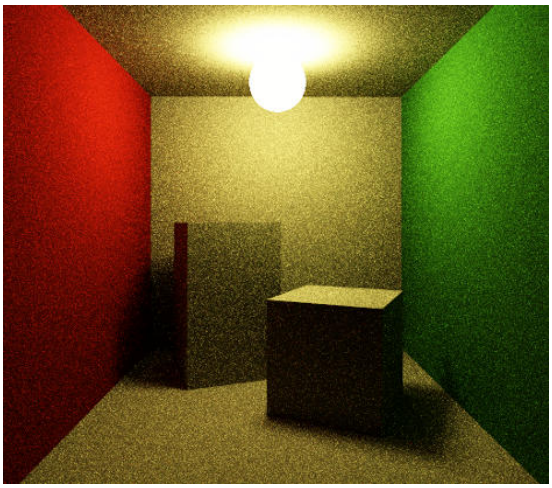

Figure 4: This "Cornell Box" was rendered with only a few samples and has severe speckling as a result. More samples would even out the speckles of this image (Penfold 2013).

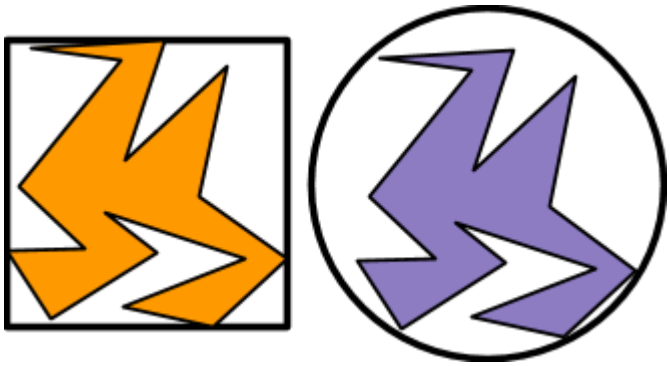

Figure 5: This figure shows different options for bounding a complex object, simplifying intersection tests.

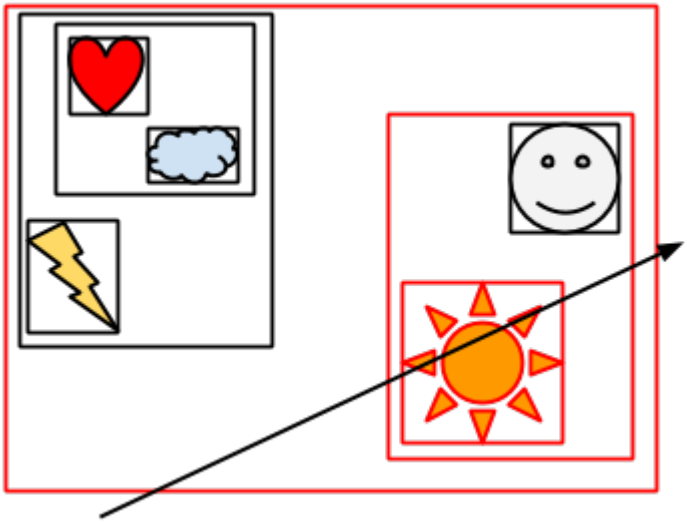

Figure 6: This image show an intersection between a ray and a HBV. All objects highlighted in red are checked for intersection with the ray, while all other objects are skipped.

which is any shape that can fully encompass an object. Figure 5 shows a complex object being surround by two different bounding volumes, a box and circle. Bounding volumes are simple shapes that provide an easy and fast way to test if an object contained within has no chance of being intersected by a ray. For example, instead of testing each of the faces of the object in Figure 5 for intersection with a ray, the ray can be tested against the box which is much simpler. If the ray does not intersect the bounding volume, then it also does not intersect the object, and no additional calculations need to be performed on this object. Larger bounding volumes can be used to surround clusters of smaller bounding volumes which allows for entire groups of

bounding volumes to be tested against a ray at once. The volumes inside are only tested if ray intersects the outermost bounding volume. Figure 6 shows how multiple objects can be grouped together inside nested bounding volumes. This data structure is called a hierarchical bounding volume (HBV).

The triangles of a mesh can be stored in these HBVs to speed up the ray tracing process. HBVs can be represented as trees, such as the one in Figure 7. This tree is a binary tree, meaning each bounding volume can only hold two objects. The top box contains two smaller boxes. These smaller boxes each contain additional boxes until eventually the objects themselves are stored. The more faces a mesh has, the more bounding volumes are needed to contain the faces, which also means more depth levels are required to store every triangle as shown in Figure 7. The speed it takes to find an intersection depends on the depth of the tree. Because objects are stored at the bottom of a tree, all bounding volumes that contain the object need to be tested before the object can be tested for an intersection. A tree with many objects will require many depth levels to store all the objects, and thus require more time to traverse to the bottom boxes. Overall, using a fewer number of triangles in a mesh can increase the speed of ray intersections.

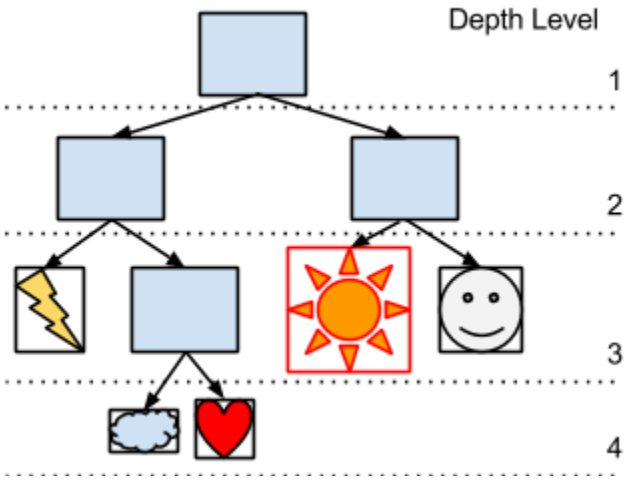

Figure 7: More depth levels are required to store larger numbers of objects.

As originally proposed by Clark, one way to lower the number of triangles in a mesh that need to be processed is to use multiple levels of detail to describe the same object, and select the lowest level possible for rendering based on certain parameters (Clark 1976). Figure 2 shows a low level of detail (LOD) model of a sphere in blue (left), and a higher LOD model of a sphere in green (right). One method of choosing the LOD is based on the distance an object is from the camera. Figure 8 shows that same blue sphere much further away. At this distance, the sphere begins to look smooth again. Pipeline renderers often use distance as one of the ways to determine which LOD to use (Lobao 2009). Close objects use high LOD, while far objects use low LOD.

Instead of using distance as the heuristic for picking an LOD, path tracers can simply use lower LODs for secondary ray intersections. Primary rays are defined as the rays that are sent from the camera before intersecting with an object. After intersection with an object, a primary ray becomes a secondary ray. We hypothesize that secondary rays have a much smaller effect on the final image than primary rays. Therefore less accuracy is needed for secondary rays. Instead of using distance to select an LOD, this thesis proposes using lower LODs for all secondary rays to speed up secondary ray intersections without significantly lowering image quality.

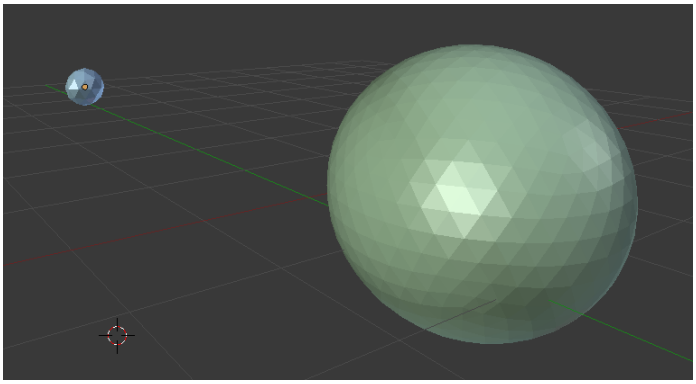

Figure 8: The blue sphere looks almost as round as the green one when looked at from a farther distance.

1.2 Related Work

The idea of using lower levels of detail to improve rendering performance began with Clark's method of defining "objects" in a hierarchy of detail (Clark 1976). He applied distance as the main heuristic in determining LOD levels. While Clark's method was the first to use LODs, many of the techniques are dated because they were based on the constraints of a 1976 computer and monitor. Yoon applies LODs to the ray tracing of massive (tens or hundreds of millions of triangles) models to achieve near real-time performance (Yoon et. al. 2006). His R-LOD algorithm includes other acceleration techniques such as ray coherence and cache coherence, which are not exploited in this thesis. Because of the mixture of ray coherence and cache coherence with LODs, it is difficult to pinpoint exactly how much Yoon's LODs actually increase performance on their own. Yoon also does not show images of diffuse ray interactions between complex models. However, Yoon's application of LOD techniques allowed for a performance increase of a factor of two. Such a technique is similar to that used by Xuemei, where he uses LOD to speed the rendering process (Lu 2007). One distinction here is the use of points rather than geometric primitives for rendering. With large datasets the points will be smaller than pixels, and are faster to render than triangles. Xuemei creates LODs by constructing tight octrees around groups of points, and giving those nodes the approximation of the points within. This approach is not as applicable to triangle meshes because triangles have both normals and back faces that need to be kept track of, while points do not have this information.

Johansson uses LOD techniques for standard pipeline rendering, but focuses in particular on the human perception of differences between LODs and original models. He conducted a study with 15 test subjects viewing two different animations, one with LOD and one without,

and asked the subjects to rate the quality difference between the animations (Johansson 2013). He found that most subjects were able to see at least a small deterioration in image quality, but none rated the deterioration as very large. With the LODs used, he saw an average of 83% faster rendering. Johansson did not use ray tracing in his renderer. Thus his metrics for choosing LODs were significantly different than those in this thesis, thus his results are not directly related. However his study serves as a guide for modeling other studies in graphics rendering.

Funkhouser presents additional heuristics for choosing LODs besides distance that can be updated dynamically to produce better results than static heuristics (Funkhouser 1993). His aim was to create steadier frame rates, which he achieved by allowing the amount of error to vary based on current frame rate. Because this thesis focuses on entirely static scenes, adaptive heuristics are unnecessary. Xia proposed methods of creating LODs on the fly progressively which could adapt to different views without significant image degradation, while improving performance (Xia et. al. 1997). However, Xia's methods are better suited to environments with a few highly complex objects in scientific visualization. As opposed to dynamic LODs as proposed by Xia, this thesis uses static LODs which are pre-computed and require less performance overhead but require more memory. Snell describes many different algorithms for providing global illumination to a scene, of which path tracing is just one (Snell 1997). Path tracing was chosen over other algorithms because it is conceptually simple, and easily adaptable to different acceleration data structures. Both Lauterbach and Thrane give comparisons of acceleration structures for increasing the speed of ray tracing (Lauterbach 2013), (Thrane 2005). Lauterbach mentions the benefits and drawbacks of HBVs against other data structures like K-d trees. HBVs are much simpler to implement, but often require more intersections to test. A

bounding box may require up to six ray-plane intersections, while a K-d node requires one. Similarly, K-d trees allow for early termination if an intersection is found, while HBVs must recurse through both child nodes (Lauterbach 2013).

2. Overview of Path Tracing

This chapter provides an overview of the techniques used in path tracing, including Monte Carlo integration, hierarchical bounding volumes, and jittering.

2.1 Monte Carlo Integration

Path tracing is defined by Shirley as using Monte Carlo methods to compute light (Shirley 2003). Each ray is a noisy estimate, and additional samples can improve the estimate. Geometrically, rays are sent from a camera into a scene, reflecting off objects based on their material properties. These rays are traced until they hit a light source, or a certain depth level threshold is met. Monte Carlo integration is a method by which integrals can be determined probabilistically (Edwards n.d.). Integrals can be described simply as taking all possible points in a given set and summing their associated differentiable areas. Of great significance to Monte Carlo integration is the use of probability density functions (PDF). These functions describe the likelihood of a random variable x taking on a given value between a certain range, and can be denoted

$$Probability(x \in [a, b]) = \int_a^b p(x)\,dx.$$

One example of $p(x)$ used in this thesis is based on picking points within a disk, such as a camera lens. Since we know the area of a disk is πr^2, we find $p(x)$ to be

$$p(x) = \frac{1}{\pi r^2}$$

such that the probability that x is in a certain subset, S_1 of the disk is

$$Probability(x \in S_1) = \int_{S_1} \frac{1}{\pi r^2}\,dA$$

where A is a measure of area.

The average value that a real function f of a one-dimensional random variable with underlying PDF p is called the expected value, denoted

$$E(f(x))=\int f(x)\,p(x)\,dx.$$

The expected value can be approximated by using a sum of independent random variables. This is called the estimated mean, and is given by

$$E(x)\approx\frac{1}{N}\sum_{i=1}^{N} x_i.$$

By the Law of Large Numbers,

$$Probability[E(x)=\lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N} x_i]=1,$$

which says that as the number of samples, N approaches infinity, the estimated means approaches the expected value. This forms the basis of Monte Carlo integration for computer graphics, where difficult integrals are solved using the estimated means of probability density functions. Shown below is the relationship between estimated means and the expected value of an integral where a random variable x maps, into a space, S. μ is simply a measure of area, similar to A in the disk example above, which expands the one-dimensional case into multiple dimensions.

$$E(f(x))=\int_{x\in S} f(x)\,p(x)\,du$$

$$\approx\frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

In path tracing, Monte Carlo integration is most often used in the materials functions, which determines the reflection direction, also called scatter direction. Difficult integrals, such as the light transport equation can be more easily approximated using Monte Carlo integration rather than directly solving.

2.2 Materials

Materials are described in the form of bidirectional reflectance distribution functions (BRDF). BRDFs are useful because they are all that is needed to characterize the properties of how a surface reflects light. A BRDF is a 4D function that takes the angle of incidence and the angle which the light hits the material, and returns the ratio of the surface radiance to the irradiance. This ratio helps explain the direction and intensity of light after reflection. While materials come in many forms, this thesis focuses on four primary material types. Diffuse, or Lambertian, objects are those which have an equal likelihood of reflection in all directions. Specular materials are those in which the reflection depends on the angle of incident, such as mirrors. Diffuse-Specular materials are hybrids that randomly select the reflection as diffuse or specular based on the angle of incidence. The last material is the luminaire material, which emits light.

2.3 Hierarchical Bounding Volumes

In order to store objects with different properties in the same data structure of statically typed languages like C++, they must implement the functions of an abstract parent class. In this

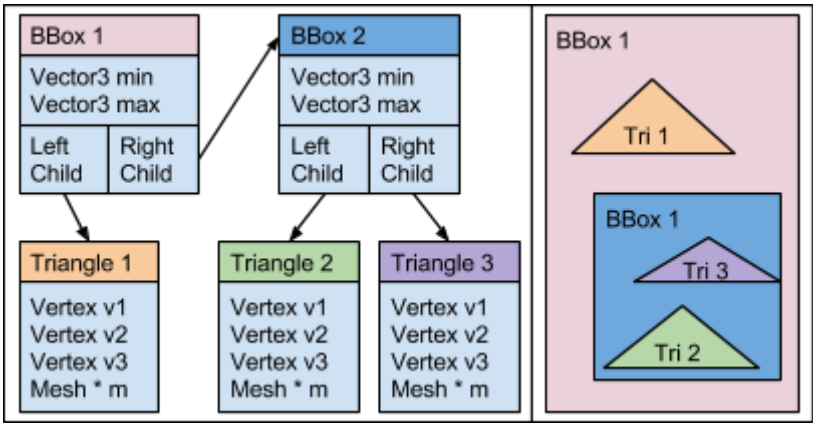

Figure 9: The basic structure of the Bounding Boxes and Triangles shown to the left, and their locations shown geometrically to the right. Arrows represent pointers. Vector3 is an x, y, z float triple.

case, that class is called `Surface`. This allows for each object to behave in a self-defined manner, while being treated the same way by the renderer. For example, the `hit()` method, that determines if a `Surface` has intersected with a ray is fundamentally different for a sphere than a triangle. However, the polymorphism abstracts this detail.

The bounding volume proposed by Shirley and used in this thesis is the axis-aligned bounding box hierarchy. Each node of the HBV consists of a left and right child pointer to other `Surfaces` (see Figure 9). The child can either point to another node, or it can be a leaf node, containing a pointer to a primitive such as a triangle or a sphere. The basic algorithm for ray-HBV traversal is as follows.

```
if (ray hits root box) then
      if (ray hist left subtree box) then
            recursively call this function for each -
            - child of the left subtree
      if (ray intersects right subtree box) then
            recursively call this function for each -
            - child of the right subtree

      if (an intersection is returned from each subtree) then
            return the closest of the two hits
      else if (only one intersection is returned)
            return that intersection
      else
            return false
else
      return false
```

The construction of the HBV is based on space, rather than perfect balance. The reason for this is that it allows for easy traversal of empty space. The algorithm basically splits the entire scene in half across each axis iteratively. The basic construction for this tree is as follows.

```
A is a list of all surfaces
N is the number of surfaces
      find the midpoint m of the bounding box of A along AXIS
      Partition A into lists with lengths k and (N-k) surrounding m
      left node = new node(A[0..k], (AXIS + 1) mod 3
      right node = new node(A[k+1...N-1], (AXIS + 1) mod 3
      Repeat until A cannot be partitioned any further.
```

Each object has its own mesh which stores all associated triangles within a HBV. These meshes are then treated as objects to be placed in the scene's global HBV.

2.4 Jittering

When using Monte Carlo techniques, higher numbers of samples produces results with less noise. In path tracing, this requires sending more rays from each pixel of the camera. With only one sample, a single ray is cast from the center of a pixel into the scene. When using multiple samples it is preferable to spread the rays out so that the rays are sent from all parts of the pixel, not just the center. By distributing the rays throughout a pixel, ridged edges are softened because the entire pixel is being sampled. Aliasing occurs when images have rough edges due to a lack of sampling within a pixel's bounds (see Figure 10).

One way to sample a pixel is to create a uniform grid. This approach, however, requires a cubic number of samples to be completely uniform, and it can produce artifacts. Picking

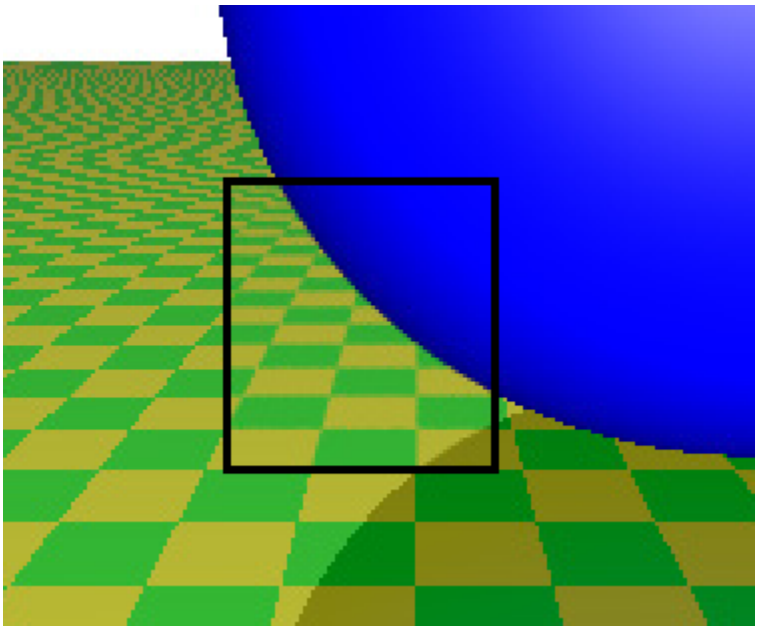

Figure 10: Inside the box the image has been jittered, while outside the box the image is aliased due to the lack of sampling (Willamette.edu n.d.).

random samples is less than favorable because the samples can be clustered instead of being evenly distributed across the pixel (see Figure 11). Jittering is a randomization technique used to sample an even distribution of points on a 2D plane (Shirley 2003). Here points are selected based on a pattern and offset by a slight amount of randomness. This guarantees a uniform distribution while still preventing artifacts caused by even grids.

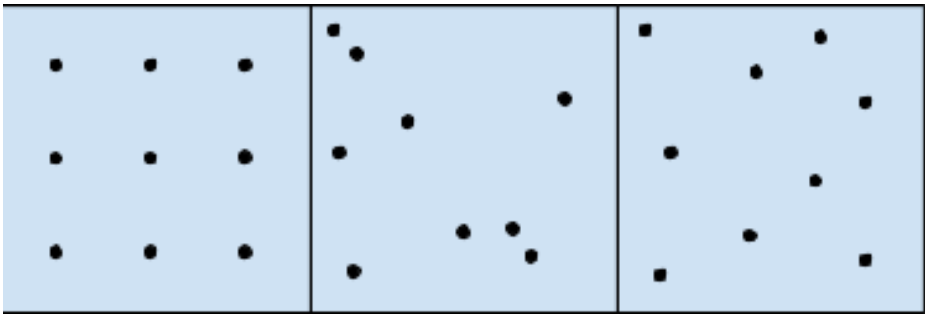

Figure 11: Grid sampling (left) has good overall coverage, but produces artifacts. The fully random sample (middle) has clusters of points together, and large unsampled spaces. The jittered sample (right) has good coverage while still enough randomness to prevent artifacts.

3. Implementation

This thesis uses most of the functions of Shirley's code, which include `Vector` and `Matrix` libraries, `Materials`, the HBV discussed earlier, and Surfaces such as triangles and spheres. While the code provided by Shirley had most of the functionality needed to perform path tracing, many additional components were required to implement multiple LODs.

3.1 Mesh Class

Significant effort was spent recreating the `Mesh` class. It previously held arrays of vertices of the triangles that comprised the mesh. Here is the original specification of the `Mesh` class.

```
class Mesh
{
public:
   ~Mesh() {}
   Material* getMaterial(int index);

   // data members
   Material**  mptr;
   Vector3*    verts;
   VertexUV*   vertUVs;
   VertexN*    vertNs;
   VertexUVN*  vertUVNs;
};
```

The `Mesh` class now contains an array of HBVs, simply called trees, which store all the triangles associated with the mesh for a particular LOD. It was also updated to be a member of the `Surface` class, and implement the Surface virtual functions.

```
class Mesh : public Surface
{
public:
      Material* getMaterial(int index);
      bool shadowHit(const Ray& r, float tmin, float tmax, float time);
      BBox boundingBox( float time0, float time1 );
      bool hit(const Ray& r, float tmin, float tmax, float time,
                  SurfaceHitRecord& rec);
      // data members
      Surface ** trees;
      string name;
      Material**  mptr;
};
```

The vertex references are no longer needed because of changes to the `Triangle` class described in section 4.3. The trees pointer is declared to be an array of Surface pointers, which in this case will be other HBVs.

```
trees = new Surface *[NUM_LODS];
```

The field trees[0] represents a full LOD ratio, while trees[1] contains a ½ ratio LOD, and so forth. The ratio that a particular level contains is not hard coded, but is instead derived from the LOD imported by the OBJFile class. While multiple LODs could have been used and stored in this Mesh, only the full LOD and one other LOD were used at a time for this implementation. This is because the tests only required one lower LOD per execution. The Mesh class also has a pointer for materials that is shared among all LODs.

The `Ray` class has been updated to include an integer called `lod` that describes which LOD to use when calculating intersections. The `Mesh` class then implements the virtual methods of the `Surface` class so that it selects the LOD based on the ray's `lod` value. An example of the `shadowHit()` function from the `Mesh` class is shown below.

```
bool shadowHit(const Ray& r, float tmin, float tmax, float time) {
      return trees[r.lod]->shadowHit(r, tmin, tmax, time);}
```

3.2 OBJ Importer

One important function that was not provided by Shirley was the ability to load high polygon count scenes into memory. We selected the Wavefront .obj file format for this thesis to store scene data because it is text-based, which allows for easier parsing, and is also widely used. A majority of the parsing was handled by the tinyobjloader, written by Syoyo. However, it was modified to be able to create triangles and materials specified by Shirley. A .obj file is structured as a text file that holds vertex, normal, and face information. The following example is a basic object in the .obj specification.

```
o Mesh_Name
v 1.123 4.425 9.004
      ...
vt 0.42 -.112
      ...
vn 5.23 -11.44 0.042
      ...
usemtl Material_1
f vertex1/normal1/uv1 vertex2/normal2/uv2/ vertex3/normal3/uv3 ...
```

The o prefix declares a new object with the name Mesh_name. A vertex is defined with the prefix v followed by three float values representing its x, y, and z position. The prefix vt denotes a texture vertex pair at u, v. Finally vn denotes a normal in the direction x, y, z. Using f creates a new face. A face can have any number of vertices, but all faces in this program are triangles. Wavefront files store faces using integer references to the vertex data. Usemtl declares which material to use for the object. Materials are declared in a separate .mtl file, which has the following syntax:

```
newmtl my_mtl
Ns 96.078431
Ka 0.000000 0.000000 0.000000
Kd 0.800000 0.800000 0.800000
Ks 0.449020 0.449020 0.449020
```

The prefix newmtl declares a new material with the given name. The prefix Ns is the Phong

exponent which explains how focused specular reflections are. Ka, Kd, and Ks, each represent colors for ambient, diffuse, and specular components, respectively. To map these Wavefront values to those usable in the path tracer requires declaring materials that match the Wavefront definitions.

```
// This creates a texture of the given red, blue, and green color
// components of the Kd values
SimpleTexture * texture = new SimpleTexture(rgb(material.diffuse[0],
      material.diffuse[1], material.diffuse[2]));

// Similar for the specular textures Ks value
SimpleTexture * spec = new SimpleTexture(rgb(f[0], f[1], f[2]));

// This creates a new diffuse material
DiffuseMaterial * diff = new DiffuseMaterial(texture);

// ns is the phong exponent
float ns = material.shininess + material.shininess *
      material.shininess / 1000;

// A texture can be created for ns values, but in this case
// all ns values are the same, so a SimpleTexture is used
SimpleTexture nstexture = new SimpleTexture(rgb(ns, ns, ns));

// This creates a new Phong material, which is a specular material
PhongMetalMaterial * phong = new PhongMetalMaterial(spec, nstexture);

// This creates the DiffSpecMaterials
// For materials with very high ns value (ns > 1998), the specular
weighting
// is increased.
if (ns > 1998)
      mesh->mptr[0] = new DiffSpecMaterial(diff, phong, ns / 3200);
else
      mesh->mptr[0] = new DiffSpecMaterial(diff, phong);
```

The diffuse-specular material randomly evaluates either the diffuse or Phong materials for color and scatter direction. The optional third parameter describes what weight to give each component. The likelihood of a material being chosen is based on the angle of incidence and the optional weighting. A weighting of 1.0 would cause the Phong exponent to be picked every time, while a weighting of 0.0 would cause the diffuse material to be picked every time. Blender, an open source modeling tool, is used in this thesis to export .obj files, as described in

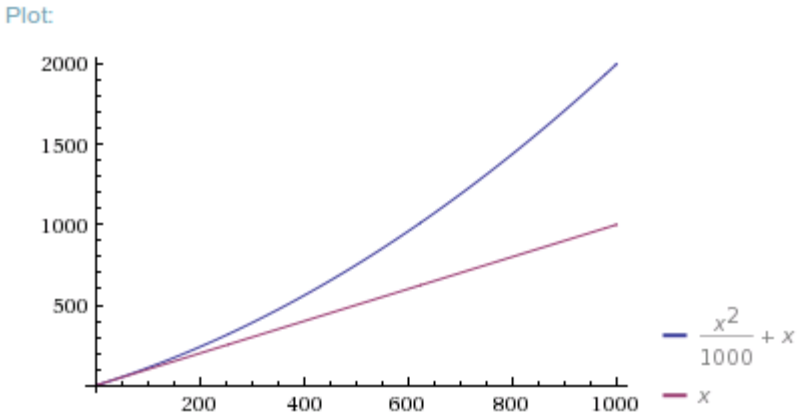

Figure 12: The red curve shows original ns mappings, while the blue curve shows the new mappings which allow for higher reflecting objects. Plotted with Wolfram Alpha.

section 4.2. One limitation of Blender is that the Phong value, ns in this code excerpt, has upper limit of 1000 when exporting .obj files, which is not large enough to produce perfect mirrors using the `PhongMetalMaterial` given by Shirley. Thus, the ns value was changed to grow exponentially so that values between 0 and 2000 can be used without lower ns objects becoming too shiny. Figure 12 illustrates the new mapping.

The `OBJfile` was also expanded to import lower LOD scenes. Each scene was saved as individual .obj files, one for each of the LOD ratios chosen. The full LOD scene is loaded first and added to a map which associates object names as strings with a pointer to the Mesh. Next a lower LOD is loaded in a similar manner, however materials are ignored. Instead of creating new meshes, the lower LOD is added to the original full mesh that shares its name, which is found by searching the map. Thus a completed mesh will store its full detailed model from the original import, one or more LODs from other .obj files, as well as a material pointer that is shared between all LODs.

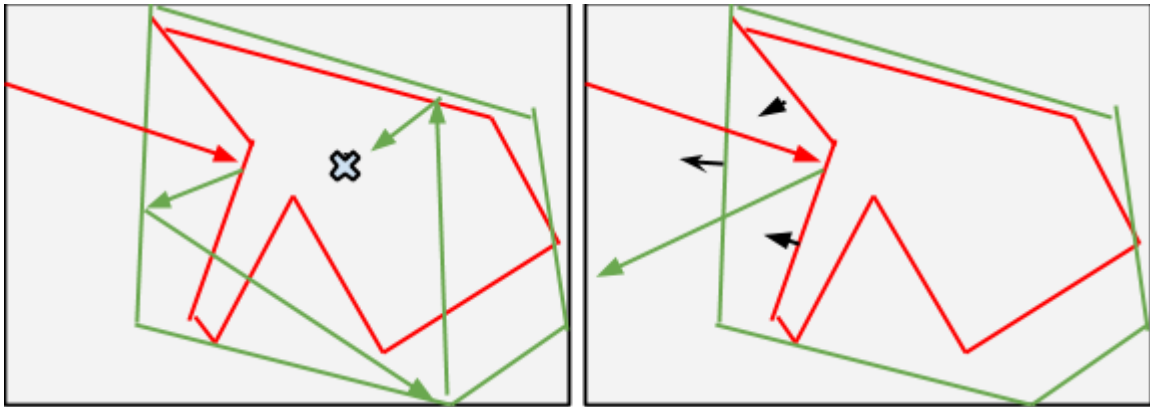

Figure 13: Red represents a full LOD, and green represents a low LOD. Primary (red) arrows can only intersect the full model, and secondary rays (green) can only intersect the lower LOD. On the left, no back face checking allows for a ray to be trapped inside a lower LOD if that LOD is larger than the original. The right image has normals which describe the direction that faces out, so rays will pass through faces if they intersect from behind.

3.3 Triangle Back Face

Shirley's original implementation of triangle intersection did not test for back facing triangle intersections. However, when switching between a full LOD and a low LOD, the low LOD could have triangles which closely cover the face of the original mesh. An intersection at this switch would cause the object to hit its own LOD for secondary rays from within the low LOD's mesh, as seen in Figure 13. This problem was solved by computing the normal of the triangle's face using the cross product of its edges, then taking the dot product of the resulting normal and the ray's direction. The dot product indicates the angle between the vectors, and if this value is greater than zero, the ray hit the triangle from behind. This technique can be seen in the following code.

```
inline bool
triangleIntersect(Ray r, const Vector3& p0, const Vector3& p1, const
Vector3& p2,
      float tmin, float tmax, float& t, float& beta, float& gamma)
{
      // Store the vectors as independant float values
      // Vector p0 - p1
      float A = p0.x() - p1.x();
      float B = p0.y() - p1.y();
      float C = p0.z() - p1.z();
      // Vector p0 - p2
      float D = p0.x() - p2.x();
      float E = p0.y() - p2.y();
      float F = p0.z() - p2.z();
      // Store the ray direction in local varables
      float G = r.direction().x();
      float H = r.direction().y();
      float I = r.direction().z();

      // Take the cross product of the vectors of the Triangle
      // to the normal direction of the face
      float one = B*F - C*E;
      float two = C*D - A*F;
      float three =A*E - B*D;
      // Take the dot product of the normal direction and the
      // ray direction. If it is greater than 0, the triangle
      // was hit from behind by the ray.
      float dot = one*G + two*H + three*I;
      if (dot > 0)
            return false;
...
```

The `Triangle` classes were also changed to not share common vertices in memory because the specification of the .obj file format makes finding shared vertices a non-trivial task. Instead, a triangle contains its own instance of each vertex, increasing memory costs. Using shared vertices takes approximately 55% less space than the method used, but it requires an additional dereference operation to access each vertex (Shirley 2003). For this thesis, the savings in space from using shared vertices as Shirley proposes was simply not worth the complication and extra code required, especially when considering the high availability of memory in modern computers.

3.4 Multi-threading

Multi-threading is a technique that allows for the concurrent execution of code. This is easy to implement when threads do not rely on each other to perform tasks. In other words, multi-threading is very useful in situations where threads can process a large amount of data without waiting for other threads to catch up, or where very little modification of shared data is necessary. Modern processors typically have more than one core which allows for multiple threads to run at the same time. In path tracing, implementing multi-threading is very simple because no data is being modified and threads do not need to wait on each other to process the next chunk of data. There are many different ways that multi-threading could be implemented in this program, for example:

- Each thread receives a portion of the image, and computes the lighting for each sample in that portion until completion. The portions of each thread are then combined to make a full image.
- Each thread processes every pixel of the image, but only for a portion of the total number of samples. Then the samples are averaged together to create the final image.
- A thread receives a small portion of the image, such as one column, and processes lighting for that portion for only one sample. Once done, it is assigned a new small portion. At the end, all samples are averaged to create the final image.

Multi-threading was implemented in the third way because it prevents any threads from being idle. Once a thread finishes a column, it is assigned the next available column that is not already being rendered by another thread. Then once every column has been rendered, the column number resets to 0, and the current sample number is incremented. A simple mutex prevents the integer that keeps track of the next available column from being corrupted by another thread while being read. This method is considered superior to the first two because it prevents a

situation where one thread may have a section of the image that is very slow to render, causing it to take longer to finish. The other threads would finish their sections, and then have to wait until the last one terminates. This would add several minutes to rendering time where the processor is not being used to its full potential. The third method prevents this situation by ensuring that when a thread finishes its assigned portion, it must find a new portion to execute instead of waiting for other threads to terminate.

3.5 Basic Algorithm

The basic algorithm for this implementation of Path tracing is as follows.

```
for (int i = 0; i < number_of_samples; i++) // Monte Carlo
{
      for each pixel // Multi-threading
      {
            get the jittered ray from the camera()
            test for intersection with the scene using full LOD
            if (intersection found)
            {
                  use the material properties of the mesh -
                  - intersected to determine the color of -
                  - the point, as well as light intensity -
                  - and scatter direction
                  Recursively test secondary ray similarly, instead -
                  - using lower LODs to a given depth
            }
            else return background
      }
}
```

Multi-threading was implemented in the *for each pixel* section, where threads take a portion of the image to render at a time. Jittering is used to sample each pixel. Each sample is tested for intersection with the scene, recursively reflecting a certain number of times while measuring color and light intensity.

4. Study Design

This chapter provides an overview of the scenes tested in this thesis, as well as the test parameters and equipment used in rendering.

4.1 Test Scenes

Two scenes were tested for this thesis. The first, referred to in this thesis as the Boxed Car scene (see Figure 24), is similar to the famous "Cornell Box" (see Figure 14), which is well known in the computer graphics community for being a simple but effective demonstration of indirect lighting. The Boxed Car scene consists of a closed box with one red wall, one green wall, and a light source in the center ceiling. In the middle is a high polygon car model flanked by two smaller boxes on each side. The car is a Volkswagen Touareg, with 386,870 triangles. The purpose of this scene was to test the visual properties of renderer. In particular, this scene tested if diffuse lighting was being accurately modeled in the reflection of light from the walls

Figure 14: The famous "Cornell Box", which in its simplicity demonstrates indirect diffuse lighting in the green and red reflections on the smaller boxes. Rendered with POV-Ray.

onto the smaller boxes. Also tested were specular reflections, such as those from the windows or the reflection of the car's mirror onto its door.

The second scene, referred to in this thesis as the Renaissance scene, is a large renaissance courtyard with a Virgin Mary statue next to the same Touareg model (see Figure 25). The Renaissance Courtyard has 975,370 triangles. The Mary statue has 35,408 triangles and the potted plant has 3,177 triangles. All together, this scene has 1,445,954 faces. This scene consists entirely of high polygon meshes, and is well suited to test the performance of the renderer. It also tested the accuracy of the renderer at different levels of detail based on diffuse lighting, specular reflections and soft shadows.

4.2 Test Parameters

The tests were all performed with five levels of depth for intersections, meaning that after five reflections the path was terminated. This parameter has been found to approximate the lighting of the scene well. A constant max depth was used because it forced the same number of reflections per sample. All tests were executed with 16 threads on Intel Core i7-3770 3.4 GHz processors with 16Gb ram. They were rendered at 1600 samples per pixel. The Boxed Car was rendered at 720x720 resolution, while the Renaissance scene was rendered at 720x980 resolution. Both scenes were run once at each of the LODs. Because the tests were run for a large number samples, the difference in time between two runs of the same scene was nearly zero. A 1600 sample test of the Renaissance scene was run two times with identical parameters. Both ran for over seven hours and finished within three seconds of each other. Thus using only one run of each LOD was sufficient for measuring performance.

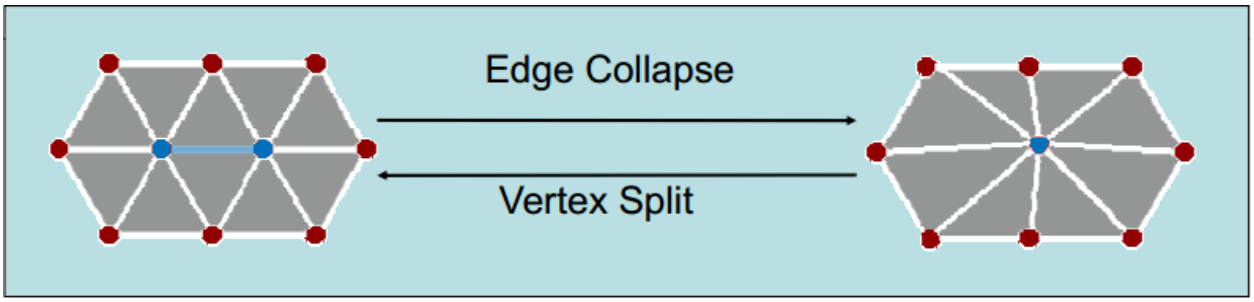

Figure 15: The blue edge is collapsed, with the two blue verticies joining to become one. This one collapse removes two triangles. The Vertex Split arrows shows the reversal of the Edge Collapse operation.

The scenes were created in Blender, an open source modeling tool. The lower LODs of the scenes were created by performing the decimate function on the original scenes, which creates a lower LOD model of any ratio between 0 and 1. According to the Blender documentation, the decimate function uses a few different techniques for reducing polygon count (Blender.org 2014). One method is Edge Collapse, which is a simple but powerful method where two adjacent vertices are merged, reducing the face count (Ben-Chen et al. 2010). This is visualized in Figure 15. Another method, UnSubdivide, works by reducing detail on grid-heavy meshes. Planar, the last method, reduces detail on very flat surfaces. The decimate tool in Blender works well for most meshes, however, it is dependent on how objects are grouped within a scene. For example, the entire floor of the Renaissance scene consists of only 256 triangles, while each post of the banister has over 2000 polygons. Because of this, a flat decimation of the entire scene using the same LOD ratio is not possible without significantly altering the scene. Many objects in the scene, such as the walls and the floor were exempt from decimation to prevent errors such as those seen in Figures 20 and 21. It is appropriate to hand pick these low polygon objects for exclusion because their lower LODs would be more accurately modeled by 3D artists than the decimate script. In addition, they already had a low impact on the overall performance of the renderer. The levels of detail used for decimation were 1, ½, ¼, ⅛, 1/16, and 1/32. This results in the following values for each scene.

LOD Ratio	Boxed Car (# of triangles)	Renaissance (# of triangles)
1	385569	1445954
½	190941	744827
¼	95583	394285
⅛	47879	219037
1/16	24020	131392
1/32	12121	87971

4.3 Expected Results

The speed increase expected using this method depends on how low of a ratio LOD was used for secondary rays. It is difficult to find the exact average case time complexity of ray-bounding volume intersections according to Walter, but it is safe to say that it is between $O(\log(n))$ and $O(n)$ (Walter et. al. 1997). Yoon explains that performance in ray tracing using hierarchical data structures is logarithmic in the number of primitives for a given resolution (Yoon et. al. 2006). As a conservative estimate, a log(n) time complexity is assumed for this thesis. Also assumed for this estimation is that the hierarchical bounding volume used is well-balanced. Given these assumptions, a one million triangle mesh would require about 20 depth levels in a binary tree to store. That same mesh's lower LOD using a 1/32 ratio would have 31,250 triangles, and require about 15 depth levels to store. All triangles would be stored at leaf nodes in the mesh, and thus require at least the same number of ray-box intersections as depth levels. This means that the full LOD mesh would take about 20 intersections to find a match under this simplified model, while the 1/32 ratio mesh would only require 15 intersections. This would result in a 25% speed increase for ray intersections. Using the profiling tool OProfile on the path tracing renderer designed for this thesis, an estimated 40% of the programs total runtime is spent in ray-bounding volume intersections. This means that an

expected speed increase using 1/32 ratio LODs is about 10%, or 25% of 40%. Once again, these calculations are highly simplified, and serve only to demonstrate potential speed increases using lower LODs. Parameters like ray direction, scene layout, and actual bounding volume construction vary the number of intersections wildly, making only approximate estimations possible.

4.4 Subjective Test

In addition to testing for performance improvements, this thesis proposes that the images produced using lower LOD meshes would be perceived as accurate as those using full LOD. This was tested in a study with 20 subjects. There were four rounds of tests. Each test had three images posted side by side in a random order. The first test was the Boxed Car scene, with all three images being the same full LOD images. The second was of the 1, 1/8, and 1/32 ratio Boxed Car images. Similarly, test three was the Renaissance scene with full detail images. Test four was of the 1, 1/8, and 1/32 ratio images. Each viewer rated the three images in order from best to worst for all four tests based on image quality. The results were then analyzed to determine if a correlation existed between LOD used and perceived image quality.

5. Results

This chapter discusses the speed of the renderer, as well as the subjective image quality comparison of the outputs at various LOD levels.

5.1 Performance

A table of the results for each test is listed below.

LOD Ratio	Boxed Car (Time in minutes)	Renaissance (Time in minutes)
1	122.51	445.31
½	123.57	454.80
¼	121.39	443.68
⅛	119.02	431.43
1/16	116.24	416.49
1/32	114.76	402.86

LOD Ratio	Boxed Car (% increase)	Renaissance (% increase)
1	0%	0%
½	-0.1%	-2.1%
¼	0.1%	0.3%
⅛	2.8%	3.1%
1/16	5.1%	6.5%
1/32	6.3%	9.5%

The difference in time is nearly 10% for the Renaissance scene, and 6% for the Boxed Car between the full and 1/32 ratio tests. This confirms that using lower LOD meshes for secondary reflections increases performance. At the ½ ratio for both scenes, the rendering actually took a bit longer, showing a small amount of overhead with this technique. Overall performance on the Renaissance scene matches what was expected, while the Boxed Car is slightly less. This is

because the Boxed Car scene only has the car which can be decimated, and a large portion of the secondary reflections are with walls, which cannot have a lower LOD. This resulted in less potential speed increase for the Boxed Car scene.

5.2 Subjective Image Quality

The results of the subjective tests show that viewers were unable to perceive any difference between the LODs used. Based on the responses given, no correlation between LOD and perceived image quality was detected. This suggests that subjects were unable to perceive any difference between the images. Figures 16 and 17 show the Boxed Car scene at full and 1/32 LOD, and Figures 18 and 19 show the Renaissance scene at full and 1/32 LOD. Figures 21 and 22 show a zoomed in image of the differences in the Renaissance scene.

6. Concluding Remarks

This chapter discusses some of the limitations of this thesis, proposes potential items for further work, and provides a conclusion of the results.

6.1 Limitations

This program relies on Blender to generate LODs for an entire scene, which is cumbersome and does not allow for the direct import of preexisting .obj scenes. The use of a progressive LOD construction algorithm during runtime, such as the one proposed by Xia, could help solve these issues by automatically creating LODs instead of using static precomputed LODs (Xia et. al. 1997). This program also relies on having very large meshes as the baseline from which to create lower LODs. Such meshes may be overly complex even for primary ray intersections, but are nonetheless preserved in full detail for baseline speed comparison, which could lead to misleading results.

6.2 Further Work

This thesis only used lower LODs for secondary ray intersections. It also always used the same LOD for the entire run, without any other heuristics to determine what LOD to use. One possible extension would be to apply an adaptive LOD selection which chooses LODs based on heuristics such as distance from the camera, or reflection depth. Objects that are farther away do not need to be rendered with as high of a quality LOD, even for primary intersections. Similarly, selection could be based on type of material intersected. Materials that are more reflective could use a higher LOD to preserve the quality of the reflection, while diffuse materials could use a lower LOD because the random scatter hides differences in detail.

Another possibility would be to try the tests with even lower LOD ratios, or potentially artist crafted LODs. An artist would be able to lower the polygon count with better preservation of the original model than the decimate function in Blender. Other forms of LOD creation could be used as well, including a progressive LOD model like Xia used in her ray tracing algorithm. The idea of using the lowest possible LOD could be taken to the extreme by approximating objects with geometric primitives, such as spheres or boxes, which would be faster to test intersections against than a low LOD polygon mesh.

Lastly, further work could be done on testing the subjective quality of images. A more in-depth study would provide more definitive results than the results of this thesis.

6.3. Conclusion

Through the use of lower LODs in secondary reflections, the path tracing renderer programed for this thesis achieved nearly a 10% increase in speed when using a 1/32 ratio LOD. This speed increase matches what was expected using the assumption that fewer depth levels in binary trees would increase performance. Despite using lower LODs for secondary rays, subjects were unable to discern differences between the full images and the 1/32 ratio LOD ratio images.

Appendix of Images

Figure 16: The Boxed Car scene with car at full detail.

Figure 17: The Boxed Car scene with 1/32 LOD used for secondary ray intersections.

Figure 18: Renaissance scene without using lower LODs for rendering secondary rays.

Figure 19: Renaissance scene with 1/32 ratio LOD used for secondary rays. Notice a slight difference in the reflection of the mirror on the window.

Figure 20: Renaissance scene without floor decimation.

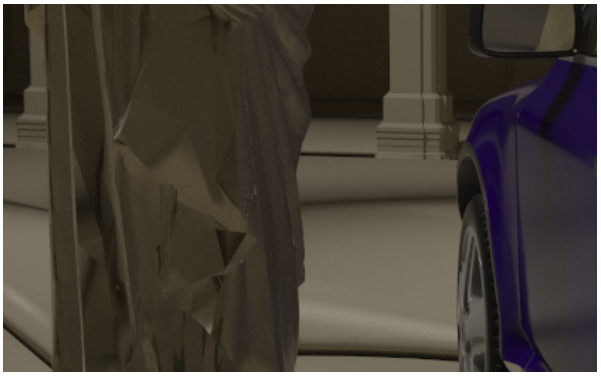

Figure 21: Renaissance scene with floor decimation. Notice the shadowing error that occurs behind the statue as a result of trying to decimate objects with too few polygons. Objects like this floor were manually excluded from decimation to prevent this.

Figure 22: 1/32 LOD zoom of the mirror.

Figure 23: Full LOD zoom of the mirror.

Figure 24: Full sized image of the Boxed Car scene with 1600 samples and full LOD.

Figure 25: Full size image of the Renaissance scene at 1600 samples, with full LOD.

References

1. Ben-Chen, Mirela, and Andy Lai Lin. "Mesh Simplification." lecture., Stanford University, 2010. http://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/08_Simplification.pdf.
2. Blender.org, “Blender 2.65 Release Notes: Modifiers.” Accessed March 28, 2014. http://wiki.blender.org/index.php/Dev:Ref/Release_Notes/2.65/Modifiers.
3. Clark, J. “Hierarchical geometric models for visible surface algorithms”. *Communications of the ACM*, 19(10):547–554, 1976
4. Edwards, Dave. "Monte Carlo Integration." manuscript., University of Utah, http://www.cs.utah.edu/~edwards/research/mcIntegration.pdf. n.d.
5. Funkhouser, Thomas A. and Séquin, Carlo H. “Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments.” *In Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '93). ACM, New York, NY, USA, 247-254. 1993.
6. Hughes, John F. *Computer Graphics: Principles and Practice*. Boston, MA, USA: Addison-Wesley Professional, 2013.
7. Johansson, Patrik. *Perceptually Modulated Level of Detail in Real Time Graphics*. master, Royal Institute of Technology, 2013. http://www.diva-portal.org/smash/get/diva2:680183/FULLTEXT01.pdf.
8. Lauterbach, Christian. “Interactive Ray Tracing of Dynamic Scenes using BVHs.” Lecture. University of North Carolina at Chapel Hill, 2006.
9. Lobao, Alexandre Santos, Evangelista, Bruno Pereira and Grootjans, Riemer. *Beginning XNA 3.0 Game Programming: From Novice to Professional* (1st ed.). Apress, Berkely, CA, USA. 2009.

10. Lu, XueMei. “Interactive Hierarchical Level of Detail Level Selection Algorithm for Point Based Rendering.” Diss. Kyungwon University, Springer Berlin Heidelberg, 2007.
11. Penfold, Dom. Wootracer, “Path Tracer 1.” Last modified April 26, 2013. Accessed April 1, 2014. http://woo4.me/wootracer/path-tracing-1/.
12. Shirley, Peter and Morley, R. Keith. *Realistic Ray Tracing*. A K Peters, Limited, 2003.
13. Snell, Q. O. and Gustafson, J. L. “Parallel hierarchical global illumination.” In IEEE Int High Performance Distributed Computing Symp, pages 12–19, 1997.
14. Teschner, Matthias. “Image Processing and Computer Graphics: Rendering Pipeline.” lecture., University of Freiburg. n.d. http://cg.informatik.uni-freiburg.de/course_notes/graphics_01_pipeline.pdf.
15. Thrane, Niels and Simonsen, Lars Ole. “A comparison of Acceleration Structures for GPU Assisted Ray Tracing.” Master's Thesis. Aarhus Universitet, Datalogisk Institut, 2005.
16. Walter, Bruce, and Peter Shirley. “Cost Analysis of a Ray Tracing Algorithm.” working paper., Cornell University, 1997. http://www.graphics.cornell.edu/~bjw/mca.pdf.
17. Willamette.edu, "Anit-Aliasing POV-Ray Examples." Accessed April 8, 2014. http://www.willamette.edu/~gorr/classes/GeneralGraphics/AntiAliasing/.
18. Xia, Julie C. , El-Sana, Jihad and Varshney, Amitabh. “Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models.” *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (April 1997), 171-183. 1997.
19. Yoon, SE, Lauterbach, C, Manocha, D, “R-LODs: fast LOD-based ray tracing of massive models.” *The Visual Computer 22* (9), 772-784, 2006.