2014

Divide and Conquer G-Buffer Ray Tracing

Daniel Stokes
Eastern Washington University

Divide and Conquer G-Buffer Ray Tracing

A Thesis

Presented To

Eastern Washington University

Cheney, Washington

In Partial Fulfillment of the Requirements

for the Degree

Master of Science in Computer Science

By

Daniel Stokes

Winter 2014

Thesis of Daniel Stokes Approved by

DR. R. WILLIAM CLARK, GRADUATE STUDY COMMITTEE	DATE
DR. PAUL H. SCHIMPF, GRADUATE STUDY COMMITTEE	DATE

Contents

List of Figures

1 Introduction

Real time computer graphics focuses on producing successive images fast enough to give the appearance of motion. As real time computer graphics continue to improve, some effects remain difficult to reproduce in a constantly changing, dynamic, scene. Current real time techniques have a difficult time dealing with effects caused by light interacting (reflecting and refracting) with multiple surfaces before reaching the viewer. There are various techniques to simulate these effects in real time, but they either do not work for dynamic scenes, or only work on specific surfaces or scenes.

When creating a still image or single frames of a video, other more physically accurate rendering techniques are often used. These techniques are often quite slow, making them impractical for use in real time rendering. One such technique is ray tracing. Ray tracing works by following light from a light source and tracking its interactions with the scene on its way to the viewer, and is able to much more naturally handle effects such as reflections and refractions. In addition to being slow, ray tracing can be unattractive for real time use as attempting to switch over to ray tracing completely from existing real time techniques would require abandoning a large body of existing real time techniques.

In the hope of merging the speed of real time rendering with the accuracy of ray tracing, a hybrid rendering technique has been emerging. This technique goes by a few different names, including hybrid rendering and G-buffer ray tracing. Since hybrid rendering is an ambiguous term, this thesis uses the term G-buffer ray tracing. In regard to using ray tracing in games (a real time graphics application)Jacco Bikker[1] points out a few open issues. One being that many rays are needed for some effects, and that “finding efficient schemes to reduce the number of rays will let ray traced games use more realistic graphics”. Another problem Bikker notes is the transition from rasterized rendering to ray tracing, as most game developers are mostly familiar with the former. This thesis aims to improve upon these two issues by using G-buffer ray tracing to reduce the number of rays required to render reflections, and to allow compatibility with many rasterized rendering techniques.

Additionally, this thesis explores an alternative ray tracing algorithm known as divide and conquer ray tracing for use on dynamic scenes to evaluate its effectiveness for real time use. Scenes are usually processed before ray tracing to gather information to improve the ray tracing performance. If the scene changes, this preprocessing step needs to be redone or updated, making dynamic scenes more difficult to process. Divide and conquer ray tracing attempts to make ray tracing dynamic scenes faster and simpler.

2 Background

2.1 Rasterized Rendering

Rasterized rendering is an algorithm for producing 2D images of a 3D scene by applying transformations to the vertices of polygons and then filling in the area of those resulting polygons. Vertex positions are typically transformed by 4x4 matrices to match a virtual camera's point of view and to project the 3D positions into a 2D space. These 2D coordinates are then linearly mapped to screen pixels to produce the final output.

Once all the transformations have been applied to the vertices of a scene, the polygon they form, usually a triangle, needs to be filled in. This is done using an algorithm known as scan conversion. Scan conversion fills in the pixels of a polygon one row, or scan line, at a time. To determine the start and end of a scan line, the edges of the polygon are found by linearly interpolating spatial and other data between two vertices. While position is the most important data to interpolate, other data such as color and surface normals can also be interpolated between vertices. Once the edges of the polygon are found, the beginnings and ends of the scan line can be found within those edges. Then it is simply a matter of linearly interpolating the edge data from the beginning and end of the scanline. This process is commonly done with triangles, rather than more complex polygons, as they simplify the algorithm and have fewer special cases than other polygons. When drawing multiple polygons, care must be taken that only visible pixels are drawn to the screen. For example, if one triangle overlaps another, any pixels behind the first triangle should not be visible. This is known as hidden surface removal. The most common hidden surface removal technique for rasterized rendering is to store a pixel's distance from the camera, known as depth, into a memory buffer commonly referred to as a depth buffer. When another pixel is being drawn to the same screen location, its depth is compared to the existing depth, and the pixel that is closest to the camera is kept.

Rasterized rendering has become the dominant rendering algorithm for use in real time applications, such as video games, because of its speed. It is a computationally fast algorithm that, especially with dedicated hardware support in the form of graphics processing units (GPUs), can easily render graphics in real-time (more than 30 frames per second). To access the GPU's functionality, graphics application programming interfaces such as Direct3D and OpenGL are commonly used.

2.2 Ray Tracing

Ray tracing is an algorithm for rendering a 3D scene that attempts to more closely emulate the physical behaviour of light than rasterized rendering [2]. Light sources emit photons that interact (reflect and refract) with and illuminate the objects they collide with. Some of these photons eventually make it to the viewer's eye. Ray tracing focuses on simulating photons with rays. To focus only on the rays that make it to the camera (the virtual eye), rays are processed backward starting at the camera instead of the light sources. These rays fired from the camera are known as primary rays. When a primary ray intersects with an object in the scene, it determines the lighting and shading at that point, and returns it to the camera. To create a more realistic image, secondary rays are often recursively cast from the point of the primary ray's intersection with scene geometry. These secondary rays may again collide with a scene object, calculate the lighting and shading at that point, and return the value to the primary ray. The primary ray can then mix the color at the intersection point on the first object with the reflected color before sending the mixed color back to the camera. Secondary rays are commonly used to create effects such as reflection, refraction, and shadows. While ray tracing often produces more photo-realistic results than rasterized rendering, it requires a great deal of computation to find the intersection points of rays with the scene, making it too slow for real time use.

How the primary rays are cast determine various camera properties. For example, the number of rays used affects the resolution of the camera, and the angle at which the rays are cast from the camera affects the camera's field of view. Secondary rays, as well as primary rays, are responsible for finding the closest scene intersection along the ray (providing hidden surface removal), determining a color value at that intersection, and possibly creating new rays at that intersection.

To improve the performance of ray tracing, scenes are often preprocessed and stored into special data structures commonly referred to as acceleration structures. These acceleration structures often represent the scene in a tree like structure allowing a ray to process a logarithmic number of scene elements instead of all scene elements to find an intersection. Two popular acceleration structures are the kd-tree [3] and the bounding volume hierarchy [4]. While acceleration structures offer significant performance improvements, they are not well suited for real time applications. Real time applications often have dynamic scene elements that can be added, moved, or removed over time. Whenever a change happens to the scene, the acceleration structure becomes invalid. Some acceleration structures, such as bounding volume hierarchies, can be updated, others, such as the kd-tree, need to be rebuilt from scratch. To reduce the cost of rebuilding or updating the acceleration structure, two acceleration structures can be used. One acceleration structure can store static scene data, such as terrain, while the other stores the dynamic data, such as animated characters. This reduces the amount of overhead for updating by leaving the acceleration structure with the static scene data untouched.

2.3 Programmable Shaders

Graphics processing units (GPUs) process graphics data in parallel using dedicated hardware. Modern GPUs allow certain parts of the graphics pipeline, such as vertex and pixel processing, to be fully programmable. The ability to program these pipeline stages offers a lot of possibilities for graphics programmer. The programs that are run by the graphics card for these pipeline stages are called shaders. In OpenGL, the graphics library used for this thesis, shaders are written in a language called OpenGL Shader Language, or more commonly GLSL. GLSL is a language syntactically similar to C, but has functionality, such as 3D math operations, built in to handle graphics processing.

2.4 G-Buffer

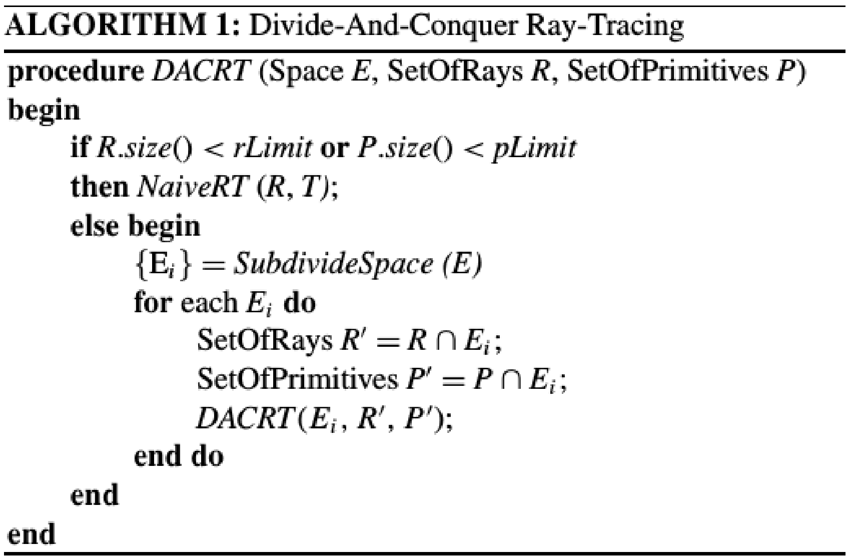
To improve performance of rendering 3D images with post-processed enhancements, Saito and Takahashi [5] proposed caching geometric information between renders. This geometric surface data was put into geometric buffers, known as a G-buffer. Each image of the G-buffer stored a different geometric property, such as position or surface normals. While the G-buffer was stored as multiple 2D arrays processed by the CPU, Saito and Takahashi still referred to each 2D array as an image. In order to fill the G-buffer, the geometry was processed as usual for rasterized rendering, and then the desired processed geometric properties were recorded in the G-buffer for each resulting pixel instead of recording the typical shaded color for the pixel.

With today's GPUs and graphics APIs, implementing a G-buffer in hardware is simple to do. The data of the G-buffer, a 2D array of geometric properties, is most easily stored on the GPU as an image, with the geometric properties being stored in the images color channels (red, green, blue, and alpha). This is commonly done using multiple render targets (MRTs) which allow multiple images to be written to in parallel. In OpenGL, MRTs are setup by attaching multiple images to a framebuffer object, and then controlling the output to each image with a shader. Writing to a specific render target in the shader is as easy as writing to a specially named variable. One problem that occurs with writing G-buffer data into images using MRTs is the limit of how many render targets a GPU can write to at once. This limit is commonly four render targets. Fortunately the work done in this thesis does not run into this problem, but care should still be taken to limit the amount of G-buffer space used to make this technique easier to combine with other techniques that may require additional space in the G-buffer.

G-buffers have made a resurgence lately with their use in screen space lighting techniques such as deferred shading [6], inferred lighting [7], and light prepass rendering [8]. These techniques start by filling a G-buffer with information needed to calculate lighting, such as surface normals and material colors, and then render lighting using the G-buffer. This approach offers performance benefits by avoiding calculating lighting on pixels that do not make it into the final image and, with some slight approximations, by only calculating lighting on pixels within range of the light source. This thesis aims to take advantage of G-buffers in a similar way allowing smoother integration with existing rasterized rendering pipelines.

2.5 Divide and Conquer Ray Tracing

The initial implementation of the ray tracer for this thesis used naive ray tracing. In other words, it tested every triangle with every ray. This requires many unnecessary and slow ray-triangle intersection tests and scales very poorly to large numbers of rays and triangles. As mentioned previously, acceleration structures are a common method used to improve performance and scalability. However, this thesis explores an alternative solution. Divide and conquer ray tracing aims to achieve similar performance as using an acceleration structure with out the use of one by using a divide and conquer algorithm [9]. Removing the need to maintain an acceleration structure makes divide and conquer ray tracing appealing for interactive use where the acceleration structure would require constant updates.



```
ALGORITHM 1: Divide-And-Conquer Ray-Tracing
procedure DACRT (Space E, SetOfRays R, SetOfPrimitives P)
begin
    if R.size() < rLimit or P.size() < pLimit
    then NaiveRT (R, T);
    else begin
        {E_i} = SubdivideSpace (E)
        for each E_i do
            SetOfRays R' = R ∩ E_i;
            SetOfPrimitives P' = P ∩ E_i;
            DACRT(E_i, R', P');
        end do
    end
end
```

Figure 1: The divide and conquer ray tracing algorithm as proposed by Benjamin Mora [9]

Mora's divide and conquer ray tracing algorithm (outlined in Figure 1) is organized into a recursive function that accepts as input a set of rays and a set of primitives (e.g. triangles). The first step in this function is to subdivide the list of primitives into disjoint subsets. Then the set of rays is tested against a bounding volume of each triangle subset in a step called ray filtering. The rays that intersect the bounding volume are passed along with the triangle subset associated with the bounding volume into another recursive call of the function. The base case for the recursion occurs when the list of rays or primitives becomes sufficiently small. At this point the remaining rays and primitives are tested naively against each other for intersections.

In Mora's implementation the primitives are triangles and the triangle set is split into two disjoint subsets. The choice of how to split the set is similar to those choices available to kd-tree construction. The simplest option is to choose an axis, find the median triangle along that axis, and partition the triangles around that median in a quicksort like fashion with triangles appearing before the median moved to the front of the list, and triangles appearing after the median moved to the back of the list. The axis to split on is chosen by finding the axis with the longest dimension. Ray filtering is often done by finding the axis aligned bounding box of the triangles of a given subset, and testing rays against that axis aligned bounding box.

Improvements can be made on Mora's initial implementation as pointed out by Áfra [10]. Afra focuses on using single instruction multiple data (SIMD) instruction sets, such as SSE and AVX, and optimal cache usage to improve performance. SSE and AVX are expansions to the x86 assembly instructions that make use of large registers that can hold an entire vector. Using SSE and AVX gives programmers the ability to perform operations on vectors in a single CPU instruction. Another use for SSE and AVX is to process multiple items at a time from a stream of data (e.g. rays and triangles). A simple example of this would be if two arrays needed to be added together with their results stored in a third array. Multiple elements from both the first and second array could be loaded into two CPU registers, and then a single instruction can add the individual elements in those to registers at once. The results are then moved from the result register into the third array. To improve cache performance,Áfra proposes storing rays and moving the entire ray in memory when necessary, as opposed to Mora's implementation where indexes into the ray list were reordered. This allows the rays to always be accessed linearly for better cache performance. Áfra also describes compact ways to represent the most vital information for rays and triangles to ease memory transfers and to better fit the data into SSE registers.

3 Related Work

3.1 GPU Ray Tracing

A popular area of research to improve the performance of ray tracing is to leverage the power of GPUs. GPU Ray tracing often makes use of APIs such as OpenCL or CUDA to write a ray tracer that can be run on the GPU. Two recently published theses from this university explored GPU ray tracing[11][12]. GPU ray tracing offers significant performance gains to CPU ray tracing, but requires algorithms specialized to work on GPUs. Continued research in GPU ray tracing could yield results that are useful for some games, but it does not offer an easy transition from current real time rendering techniques.

3.2 G-Buffer Ray Tracing

An approach to real time ray tracing similar to the one presented in this thesis was published last year by Sabin et al.[13]. Sabin et al. propose using G-Buffers to mitigate the cost of primary rays. This allows for more resources to be used on the secondary rays responsible for the lighting effects that allow ray tracing to achieve more photo realistic results than rasterization. Sabin et al. go on to further propose heuristics for prioritizing rays that will have the most visual impact on the scene. This allows for better visual effects with fewer rays. The approach proposed in this thesis differs in its handling of secondary rays, by attempting to reduce the number of rays needed to complete a frame rather than dropping the ones that are deemed less important. These two approaches to handling secondary rays are not mutually exclusive, and may be usable together to yield even better performance by further reducing the number of rays needed. Among other topics, Cabeleira[14] also discusses using a G-buffer to create real-time reflections. However, his work did not give much attention to dynamic scenes, other than to mention using an acceleration structure that supports updates such as a bounding volume hierarchy.

4 Renderer Details

4.1 Skipping Primary Rays

Ray tracing's primary rays need to accomplish four things: determine camera properties, find the closest point of intersection with scene geometry, determine a color value at the point of intersection, and determine the origin and direction of any secondary rays. The first three can be easily handled by rasterized rendering using transforms (e.g. perspective transform) to handle camera properties, depth buffering to handle finding the pixel closest to the camera along each ray, and using shaders to determine a color value for each pixel. This leaves only the task of determining the origin and direction for a secondary ray. This last step can be accomplished through the use of G-buffers. The data necessary to create secondary rays at each intersection point can be stored in the G-buffer as each pixel in the G-buffer can be seen as an intersection of a primary ray with the scene. What data is stored into the G-buffer and how it is processed is detailed in the following sections.

4.2 Design Overview

The renderer used in this thesis combines rasterized rendering on the GPU using OpenGL with ray tracing done on the CPU. The GPU is used to create a G-buffer, create an R-buffer, and to render the final output. The R-buffer is a color buffer that represents the color that a reflected ray from a given pixel would return. The values in the R-buffer assume that surfaces are perfect mirrors, and therefore may need to be mixed based on surface properties to produce a final color value. The CPU is responsible for processing the data stored in the G-buffer, performing the ray tracing, and gathering ray tracing results into a format the GPU can use to create the R-buffer. This process requires the scene to be rendered twice, and requires one transfer of data from the GPU to the CPU, and one transfer of data from the CPU to the GPU. An overview of the steps taken to produce the result of a single frame is given below.

1. Render the scene into the G-buffer to record surface properties
2. Transfer G-buffer data from the GPU to the CPU
3. Process G-buffer data into rays
4. Cast rays into the scene to find intersections
5. Collect data about the intersections
6. Process intersection data into a GPU friendly format
7. Transfer intersection data from the CPU to the GPU
8. Use intersection data to draw reflected color values into R-buffer
9. Render scene using the R-buffer to lookup reflected color values to mix with the material.

How the surface of an object looks, including how reflective the surface is, is defined by a material. Materials are a collection of data representing surface properties such as diffuse color. Material shaders make use of this data to perform calculations that determine the final color of a pixel on that surface. There is a many-to-many relationship between materials and shaders as materials are used as input for the shader's calculations. In this thesis two shaders are used, one to write out surface properties to the G-buffer, and another that calculates the actual color of the surface. This second shader is used to fill the R-buffer as well as producing the results of the final render to achieve visual consistency between when an object is viewed directly and when it is viewed in a reflection.

4.3 G-buffers

The G-buffer is created by setting up render targets (i.e. images) for data we want to collect, and writing that data to the render targets by rendering the scene with a shader dedicated to this rendering pass. To define a ray, the origin and direction of the ray are needed. The origin of a secondary ray is position of the intersection that created it. The direction for a reflected ray is found by reflecting the vector from the camera to the pixel about the pixel's normal using equal angle reflection. Since the positions are stored in view space, this reflected vector can be found easily with the formula $P - 2(PN)N$ with P being the view space position and N being the surface normal. This reflection calculation can be done on the GPU and stored into the G-buffer, or the normal can be stored in the G-buffer with the reflection calculation being done on the GPU when the G-buffer data gets processed. While doing the reflection calculation on the GPU would be faster, most techniques that use G-buffers also need the normal stored in the G-buffer. By calculating the reflected vector on the CPU, we can make use of the normal data that may already be saved for a different use. The implementation for this thesis stores the normal in the G-buffer for better compatibility with other techniques that use the G-buffer. One more piece of information needs to be stored in the G-buffer, a reflectivity bit. The reflectivity bit is used to determine if ray tracing needs to be done for that pixel. If the reflectivity bit is false, the G-buffer data for that pixel is ignored when processing the data for ray tracing.

With the G-buffer being stored as an image, it is important to look at OpenGL image format specifications [15]. The image format is used to describe how data is stored into an image. The image format name can be broken into three parts. First is the list of components stored by the image. These can also be seen as the color channels the format supports, though arbitrary data can be stored into an image that may not represent colors. The next part of the image format name is how many bits are used to store each component. Using more bits per component offers more precision, but at the cost of slower reads and writes to the image. The third part of the image format name describes the data type of the data stored in the image. If only the components and size are listed in the name, then the data is stored as an unsigned normalized integer. A suffix of "F" denotes the image data is stored as a floating point number. While storing data as floats into an image can be convenient, floating point image formats are often slow to read and write to, making integer formats preferred if the data can be represented in one of the integer formats. OpenGL also supports some special image formats that do not strictly follow this naming convention. An example of this is the RGB10_A2 format. This format uses 10 bits for the red, green, and blue components, but only 2 bits for the alpha component. More details about image formats can be found on the OpenGL wiki [15].

For the G-buffer used in this thesis, the surface normals are stored at 16bits per channel into an RGBA16 image format. The only data currently stored in the alpha channel of the normal image is the reflectivity bit, leaving plenty of room to store other flags and data. Originally a RGB10_A2 image format was used, but the low precision normals caused rendering artifacts in ray tracing results (Figure 2). The increased precision to remove the artifacts is worth the trade off in GPU performance for this thesis as the rendering tends to be CPU bound. In other words, the GPU is

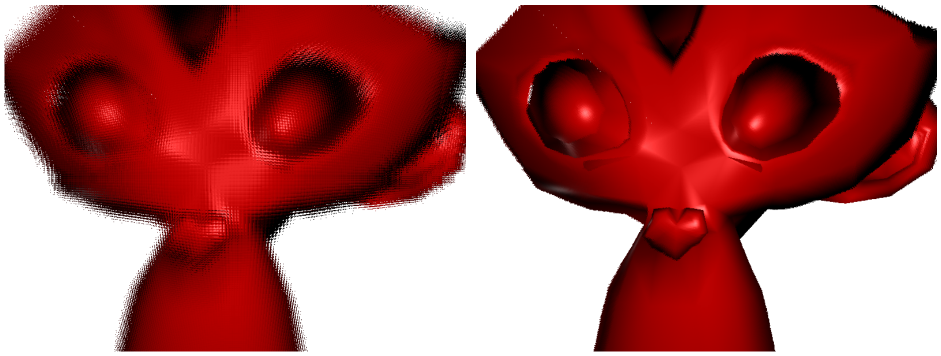


Figure 2: These images of Suzzane reflecting off of the interior surface of a sphere with normals stored in 30 bits (left) and 48 bits (right) highlights artifacts caused by storing normals in a low precision image format.

waiting for things to do as the CPU is finishing its tasks, such as ray tracing. The view space positions are stored in an RGB16F image format. The positions could be stored in a normalized integer format by dividing the positions by the dimensions of the scene, however the floating point format is easier to work with and the GPU performance is not critical for this thesis. Alternatively, rather than saving out the view space positions of pixels, the view space positions can be reconstructed from the depth buffer as described in [16]. This frees up an entire render target that can either be used for other data, or omitted to improve GPU performance. However, this requires more work to be done on the CPU as it will need to handle the reconstruction of the position data.

4.4 Ray Tracer

After the G-buffer is filled, it needs to be transferred from the GPU to the CPU. With OpenGL this can be accomplished with a glReadPixels call. Once the data is moved to the CPU, it can be processed for the ray tracer. This stage produces two lists, the normals and positions read from the G-buffer. Only pixels that require ray tracing (those with their reflectivity bit set) make it into these lists. At this point the normals, which are stored in an unsigned image format, are brought back into the -1 to 1 range using linear remapping. Although an integral image format is used for the normals, OpenGL can handle the conversion to floating point numbers when doing the memory transfer.

Along with the rays, triangles also need to be processed before ray tracing. The scene's triangles are reprocessed every frame. This means the ray tracer makes no assumptions about the previous state or existence of triangles from previous frames, allowing it to handle changes to the mesh information during run time. This process could potentially be improved by caching known static triangles in between frames. When the triangles are processed a bounding box, a box that completely encloses the triangle as tightly as possible, is found along with the triangle's centroid, the average of the positions of the three vertices that make up the triangle. These pieces of information are used later for the ray tracing.

The data for each ray and triangle are stored into two separate structures each as illustrated in Figure 3. The Ray struct contains the information necessary to perform the part of the divide and conquer ray tracing that requires the most movement of the rays in memory. This helps to decrease the cost of constantly moving the rays around in memory. A second struct, RayResult, contains the information related to the intersection of a ray with a triangle. The appropriate RayResult is found by storing an id on the Ray that acts as an index into the array of RayResult structs. Similarly the Triangle struct acts as a minimal representation for a single triangle, and contains the axis aligned bounding box of the triangle and an id to match it with the appropriate FullTriangle struct.

The ray tracing is done using a divide and conquer approach based on Mora's implementation [9] with a few ideas borrowed from Afra's implementation [10]. For each recursive iteration of the divide and conquer function, the set of triangles is split into two subsets, the rays are filtered against those subsets, and a recursive call to the divide and conquer function is made for each subset and the rays that passed the

```
struct Ray {
    float origin[3];
    float direction[3];
    float
    unsigned int id;
};

struct RayResult  {
    float uv[2];
    unsigned int tri_id;
    float origin[3];
    float direction[3];
    float t;
};
```

```
struct Triangle {
    float aabb[2][3];
    unsigned int id;
};

struct FullTriangle {
    float vertex0[3];
    float vertex1[3];
    float vertex2[3];
    float normal0[3];
    float normal1[3];
    float normal2[3]
    float texcoord0[2];
    float texcoord1[2];
    float texcoord2[2];
    float edge0[3];
    float edge1[3];
    float centroid[3];
    Material *material;
};
```

Figure 3: The structures used to hold ray tracing data

filtering step. The splitting method used in thesis is a simple median cut. The axis with the largest range of values is found, and the median triangle along that axis (based on the triangle centroids) is chosen. The centroid of that median triangle is used as the point along the chosen axis to split. The median triangle is found using nth_element function from the C++ standard library which finds the median in linear time and conveniently sorts the set of triangles such that the median is in the middle of the set, the values below the median are in the first half of the set, and the values above the median are in the second half of the set. This allows the set, stored as an array, to be treated as two disjoint subspaces. Next the axis aligned bounding boxes, a bounding box that only has planes perpendicular to the world axis, for each subspace are found. The set of rays passed into the divide and conquer function are then tested against the axis aligned bounding box. The rays that pass the intersection test with a given subspace are passed along with the triangles of that subspace into another iteration of the divide and conquer function.

The recursion stops when either the size of the set of triangles or the size of the set of rays falls below a threshold. The value of 8 was used for both thresholds as suggested by Áfra [10]. At this point each triangle is tested against each ray using the classic Möller Trumbore ray-triangle intersection [17]. When an intersection is found, the results are recorded in the corresponding ray's RayResult structure.

```
struct DrawRay {
    float origin[3];
    float position[3];
    float normal[3];
    float texcoord[2];
    float pad[5];
};
```

Figure 4: The struct that is passed to the rasterizer to render as points.

After all the necessary ray-triangle intersection tests have been completed, the results for each ray are processed in order to gather the information necessary for the rasterized renderer to fill the R-buffer with the result. In this structure (Figure 4), two different position values are stored (origin and position). The origin is the origin of the ray and is used to determine where to draw the result in the R-buffer, whereas the position is the position the ray intersected with the triangle and is used for shading. Normal and texcoord are also from the intersection point and are used for shading. The position, normal, and texcoord are all found by using the barycentric coordinates computed from the Möller Trumbore ray-triangle intersection. Extra bytes of padding are added to the end of the struct as some GPUs, especially those made by AMD, prefer drawing data to be aligned to 32 bytes. The ray results are converted into this format, skipping any results that did not have an intersection, and sorted into buckets by material. This bucketing is done by using a C++ standard library map keyed by material pointers. Once all the results have been processed into buckets, the buckets are emptied into an array with all results of the same material being in contiguous memory. This allows the rasterized renderer to handle all results of the same material at once.

4.5 Rasterized Renderer

The rasterized renderer is responsible for getting the ray tracing results into the R-buffer. When working with manipulating color values, it is important to know the difference between linear (not gamma corrected) and non-linear (gamma corrected) color spaces. GPU Gems 3 Chapter 24[18] does a great job explaining the problem, and how to fix it. In short, images are often stored in a non-linear color space to appear correctly on monitors. However, using non-linear colors can cause errors in shader calculations. This means any non-linear color input must be converted to a linear color space before using it for any calculations, and any output should be converted back to a non-linear color space to display correctly. To allow these conversions to happen on the GPU, the R-buffer data is stored in a GL_SRGB8 image format.Values read from an sRGB image format are automatically converted to a linear color space, and any values written to an sRGB image format are automatically converted to a non-linear color space.

To handle the transfer of data from the CPU to the GPU, a special buffer is created. This is a buffer that allows vertex attributes to be uploaded to a GPU and is called a vertex buffer object in OpenGL. Each ray result is uploaded as a vertex so that it can be drawn as a single pixel in the R-buffer. To avoid needing to recreate the vertex buffer object (often a slow operation), it is initialized to a size equal to the number of pixels in the G-buffer (the R-buffer is the same size). This leaves enough room in the vertex buffer object in the case that all pixels from the G-buffer require reflection. Every frame the ray data is loaded into the front of the vertex buffer object using a glBufferSubData call. For each material present in the ray tracing results, the shader is set up and the portion of the vertex buffer object that corresponds to that material is rendered. This is accomplished by taking advantage of glDrawArray's ability to choose a start and end point, and is why the ray tracing results were ordered into contiguous groups based on material.

Once the R-buffer is filled, the scene is rendered making use of the R-buffer. The final shader can use the position of the pixel it is currently processing to look up the appropriate color value from the R-buffer, and mix it with the final color based on the material properties.

4.6 Optimizations

4.6.1 Pixel Buffer Objects

To ease the overhead of transferring image data between the CPU and GPU, OpenGL's pixel buffer objects[19] can be used. Pixel buffer objects allow for asynchronous transfer of image data between the CPU and GPU. This means that the CPU can continue to do work while image data is being transferred, unlike glReadPixels which blocks until the transfer is complete. In order to take advantage of this asynchronous behavior, the CPU actually needs something to do while the data transfer occurs. If image data is immediately needed, the data can be written into one pixel buffer object while reading from a second pixel buffer object. During the next frame, the roles are swapped and the first pixel buffer object is read from while the second one is written to. Using two pixel buffer objects in this way causes a minor problem on the first frame when nothing has been written into either pixel buffer object. This means there is no data to read while one of the pixel buffer objects is filled. To handle this, either the CPU can wait for the first transfer to complete, or simply use the empty buffer (in the case of G-buffer ray tracing this means no reflections on the first frame).

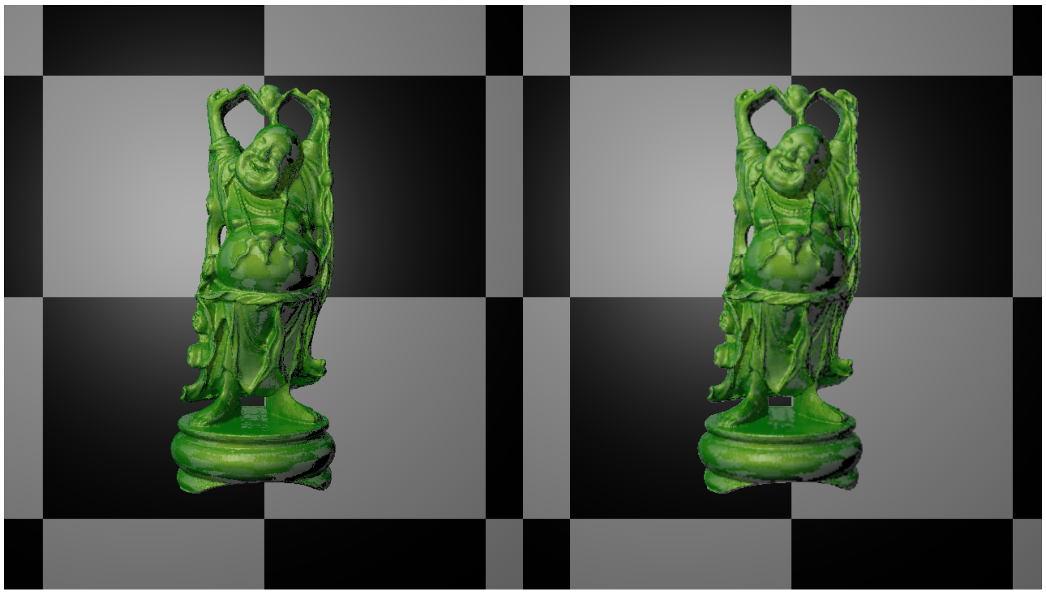


Figure 5: The Stanford Happy Buddha model rendered with a G-buffer resolution of 100% (left) and 50% (right)

4.6.2 Mixed Resolution Rendering

One of the most computationally expensive parts of ray tracing is the ray-triangle intersection testing. The number of ray-triangle intersection tests needed is controlled in part by the number of rays being traced. The number of rays needed is in turn controlled by the number of reflective pixels in the G-buffer. It is possible to render the G-buffer at a lower resolution to reduce the number of pixels in the G-buffer and the number of rays produced from the G-buffer. Because the R-buffer is highly dependent on the G-buffer data, it is created at the same resolution as the G-buffer. By lowering the resolutions of the G-buffer and R-buffer, some blurring in the reflections can occur as the lower resolution images are now effectively stretched across larger surfaces. Furthermore, the sampling required to stretch the lower resolution textures across a larger surface can cause artifacts in areas that have discontinuities, such as the edges of objects. This is caused by some of the data from one side of the discontinuity bleeding into the other, and could potentially be fixed by a smarter approach to sampling the lower resolution buffers, such as the discontinuity sensitive filtering used by Kircher et al[7].

4.6.3 Interlacing

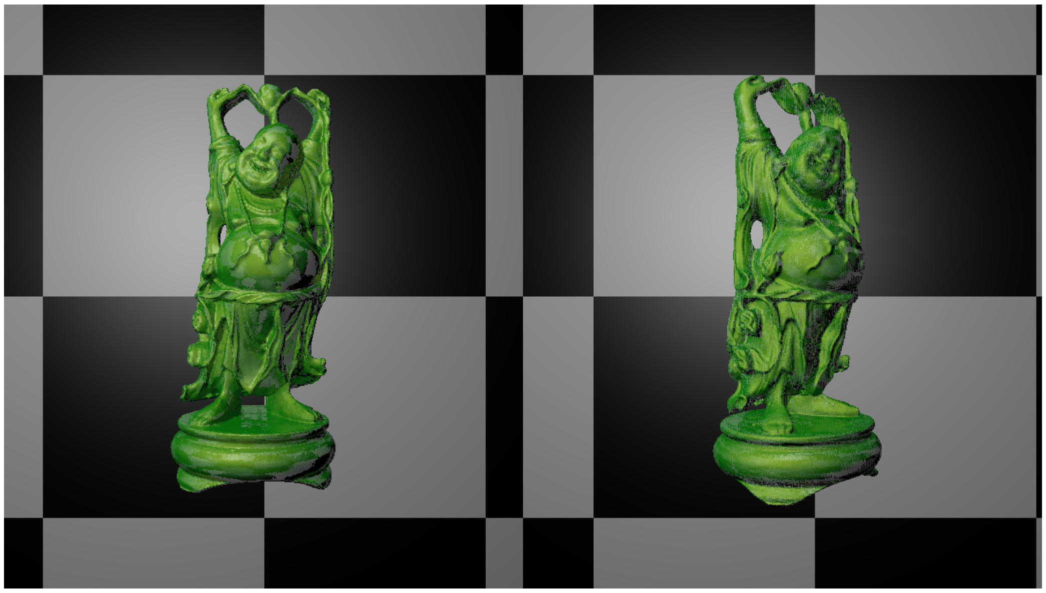


Figure 6: The Stanford Happy Buddha model rendered with out interlacing (left) and with (right) interlacing across two frames

The number of rays that require tracing per frame can be further reduced by only processing a portion of the rays per frame. This can be accomplished by interlacing the ray tracing results over multiple frames. The rendering into the R-buffer can be interlaced by not clearing the R-buffer between draws, and only processing every n^{th} ray and storing its result into the R-buffer, where n is the number of frames to interlace between. As an example, on one frame half of the rays could be processed with their results stored into the R-buffer, with the other half of the pixels in the R-buffer remaining in their previous state. On the next frame the other half of the rays can be processed with their results stored into the R-buffer, and this time leaving the results from the first half of the rays in their previous state. This can have a significant impact on performance, but at the cost of temporal artifacts. These artifacts appear as banding in areas that have changed significantly between frames. These artifacts become less noticeable as the frame rate improves and there is less difference between two frames.

5 Results

OS	Arch Linux 64bit using the 3.12.9 Linux kernel
Compiler	gcc 4.8.2
CPU	Intel Core i5-2410M CPU @ 2.30GHz
GPU	AMD Radeon HD 6330m
Memory	6GB DDR3

Figure 7: System information of the environment used for testing

Testing was performed using the system outlined in Figure 7, and all images were rendered at a resolution of 1280x720. Four images demonstrating the Happy Buddha model at four mesh resolutions from the Stanford 3D Scanning Repository[20] with no interlacing and a full resolution G-buffer can be found in Figure 11 and Figure 12.

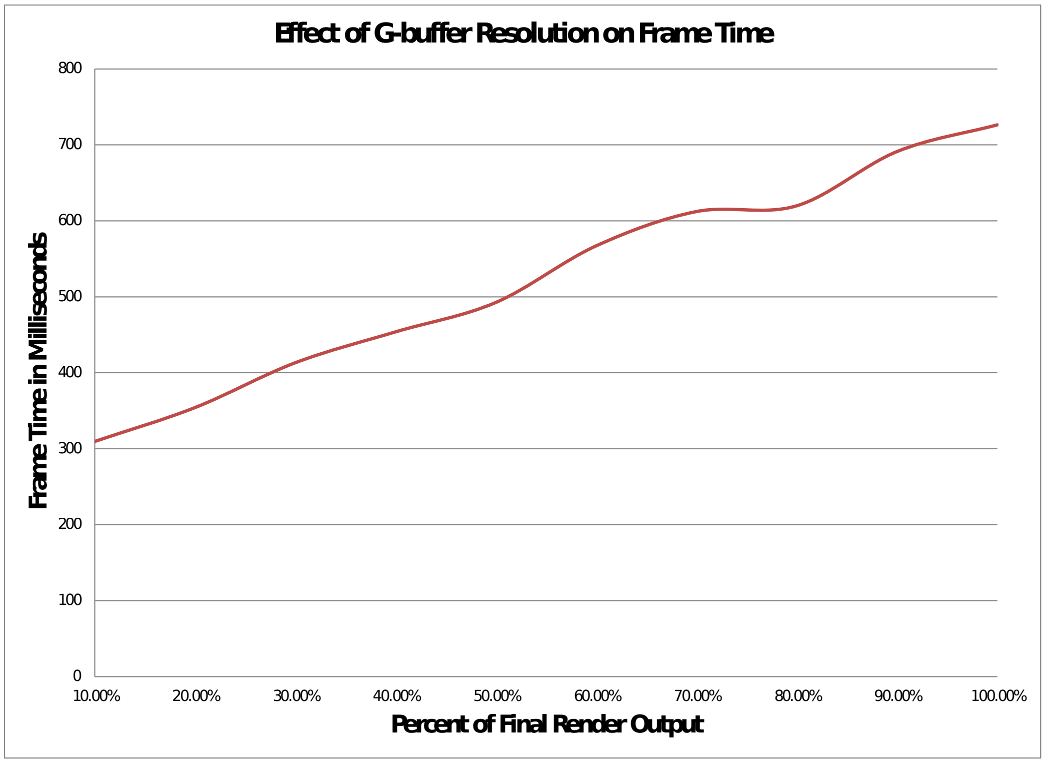


Figure 8: The effect on frame time of changing the G-buffer resolution on a scene of 290,633 triangles

Figure 8 shows the effect of changing the G-buffer resolution on frame time. The percentage indicates the ratio of the G-buffer resolution to the final image resolution (1280x720 for these images). This means a G-buffer resolution of 50% contains half the number of pixels as the final output, and also uses half the number of rays of a G-buffer resolution of 100%. This data suggests that a linear relationship exists between the number of rays (or pixels in the G-buffer) and the frame time. Furthermore this demonstrates that it is possible to trade quality (G-buffer resolution) for performance (frame time).

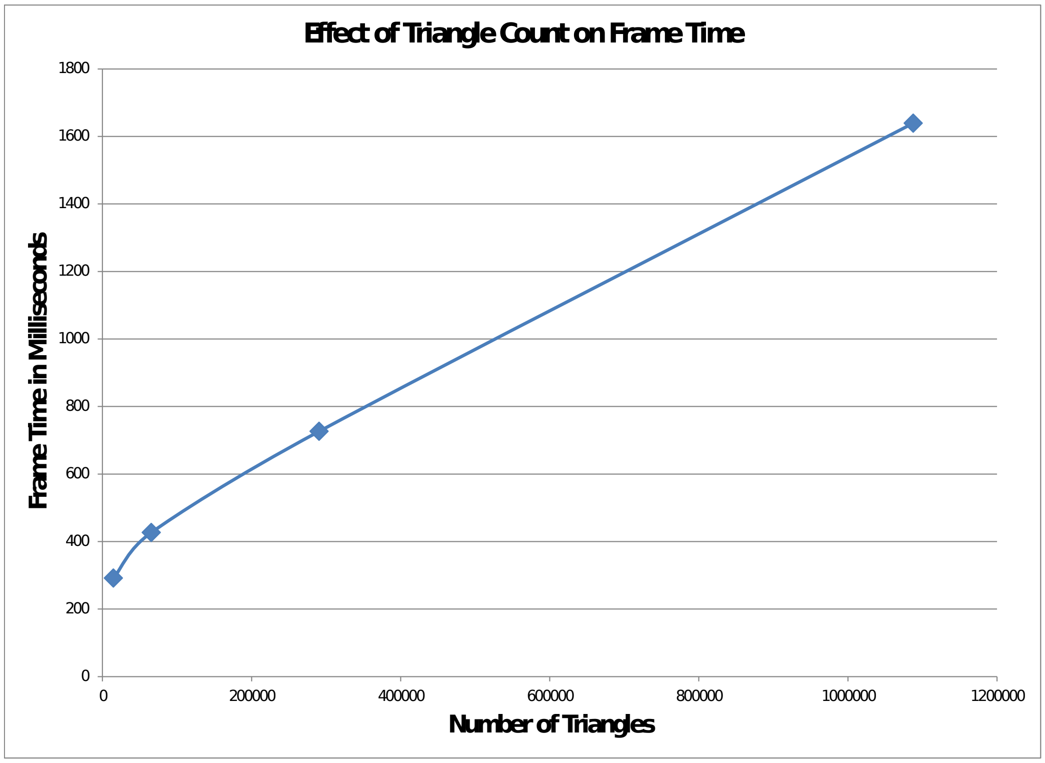


Figure 9: The number of triangles in the scene also impacts the ray tracing time

The number of triangles in the scene also has a significant impact on the performance of the ray tracing as seen in Figure 9. While there are not enough data points to draw a definitive conclusion, it appears the number of triangles also has a linear effect on the frame time once a certain threshold of triangles is reached. The performance of ray tracing an acceleration structure often scales logarithmically to the number of triangles. This means using an acceleration structure for static scene elements could improve the performance of this technique for larger scenes with many static elements. However, the scenes used for testing were almost entirely dynamic, with only the twelve triangles making up the background being static, meaning acceleration structures would likely not improve the performance of the test scene. It is important to note that high triangle counts already cause performance problems in rasterized rendering, meaning artists already try to make efficient use of triangles to improve performance.

Figure 10 shows the effect of two different optimizations on a scene containing the 14,765 triangle Buddha model with a full resolution G-buffer. Pixel buffer objects (PBOs) improved the performance of transferring data from the GPU to the CPU, but most of the time spent creating the R-buffer is in the ray tracing itself. This is why adding interlacing has a larger impact on the frame time, as it cuts the number of rays used in half.

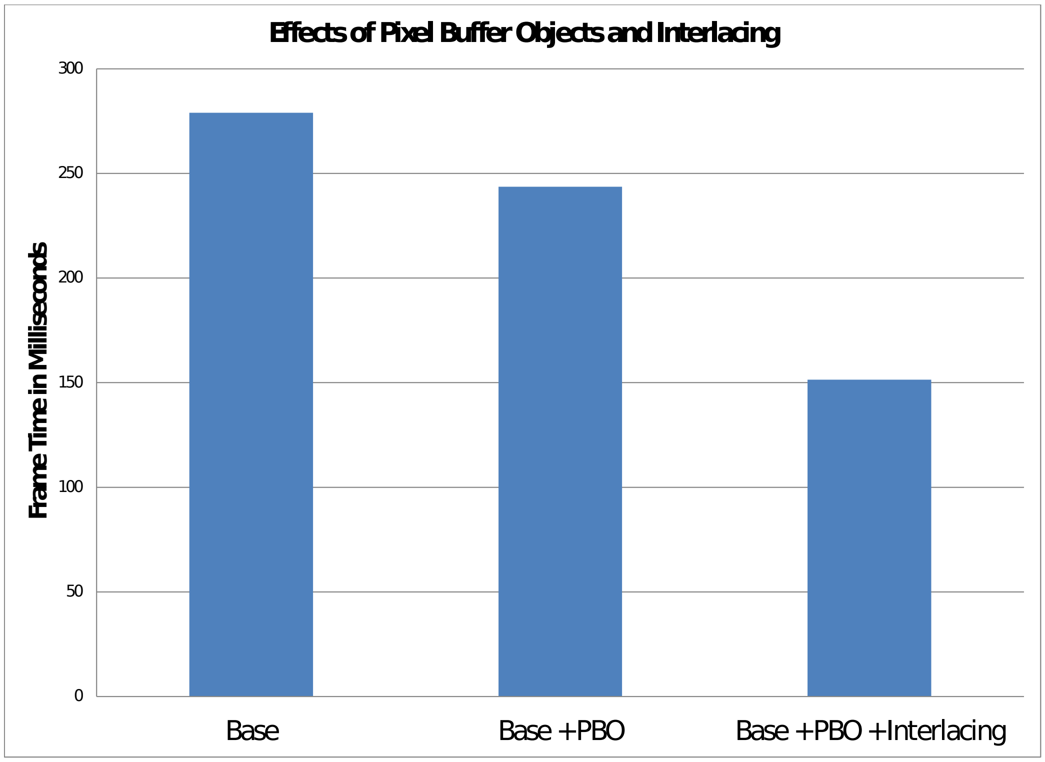


Figure 10: The effects on frame time of using pixel buffer objects and interlacing on the approximately 15,000 triangle Buddha model

6 Conclusion

This thesis aimed to provide insight into a means of combining rasterization and ray tracing in the hopes of achieving real time dynamic reflections on arbitrary meshes in the near future. Specifically, the goals of this thesis were to reduce the number of rays needed to ray trace a scene, determine if this solution can produce ray traced reflections at better than 30 frames per second, and to evaluate the ease of integrating this solution into existing real time applications.

The first goal was certainly met through the use of G-buffer ray tracing. By removing the primary rays, an image with a resolution of 1280x720 avoids tracing over 900,000 rays (1 ray per pixel). The number of rays can be further reduced by reducing the resolution of the G-buffer and with interlacing. G-buffer ray tracing clearly works for reducing the number of rays required to add reflections to a scene. Unfortunately the implementation in this thesis was not fast enough achieve the desired minimum frame rate of 30 frames per second on complicated scenes. However, this is likely in large part due to insufficient optimizations. When comparing the performance of the divide and conquer ray tracing done in this thesis to the performance of Afra's[10] implementation, there is nearly an order of magnitude of difference in millions of rays per second on a single thread. While part of the performance difference comes from the difference in CPUs, there is obviously room for more optimization. Furthermore, separating static and dynamic geometry could offer significant performance improvements. The dynamic geometry could continue to use divide and conquer ray tracing, whereas the static geometry could be preprocessed into an optimized acceleration structure. This would add more complexity to the implementation, but the performance benefit would likely be a worthwhile trade off. Even with these optimizations, G-buffer divide and conquer ray tracing may still not be enough for real time needs, but it is certainly a step in the right direction. The third goal, easy integration with existing solutions has been met. This solution works well with existing shaders and many real time rendering techniques, though it may require some extra effort to work with screen space lighting techniques such as deferred shading and inferred lighting.

While the techniques in this thesis do not reach the desired goal of dynamic real time reflections on arbitrary meshes, they do show a potential road map to achieving it. The use of a G-buffer and divide and conquer ray tracing provide a way to drastically reduce the number of rays needed and allow a way to handle dynamic geometry, and show promise for use in real time applications such as video games. By turning down the quality of the scene and the reflections, it is possible to achieve ray traced reflection in real time (greater than 30 fps).

7 Future Work

7.1 Screen Space Lighting

Throughout this thesis, decisions were made for compatibility for screen space lighting techniques, such as storing normals instead of reflected ray directions in a G-buffer. However, the current implementation of the rasterizer uses traditional forward rendering; it calculates lighting per material. The methods used in this thesis can be expanded to work with screen space lighting. The following proposed solutions assume deferred shading is used. The first possible solution is to draw the rays directly into the G-buffer instead of creating a separate color buffer with the reflection results. This makes it difficult to blend reflections with the material color, but should work for perfect mirrors. Instead, it may be better to follow the entire render process when creating the R-buffer. This means creating a G-buffer from the ray traced results and compositing lighting into it for a final color buffer.

7.2 Multi-Threading

Another area of improvement is the use of multi-threading to improve performance. The ray tracing in this thesis is done on a single CPU thread, using more CPU threads would better utilize modern CPUs and effectively throw more hardware at the problem. Unfortunately the divide and conquer ray tracing algorithm may not parallelize as well as standard ray tracing with an acceleration structure as hinted at in [10]. In order to parallelize this thesis' ray tracer, the rays could be divided amongst available threads and each thread could perform the divide and conquer ray tracing algorithm on its own subset of rays. Unfortunately, this would require copying the triangle list to each thread, or using extensive thread synchronization, since the triangle data is rearranged in memory. A subset of triangles could be determined for each core by following the divide and conquer algorithm to create a number of subspaces equal to the number of available threads. Another possibility is for each recursive call to the divide and conquer ray tracing function to be performed on a new thread until all available threads are used. At that point the threads would continue as usual and stop creating new threads.

7.3 Multiple Bounces

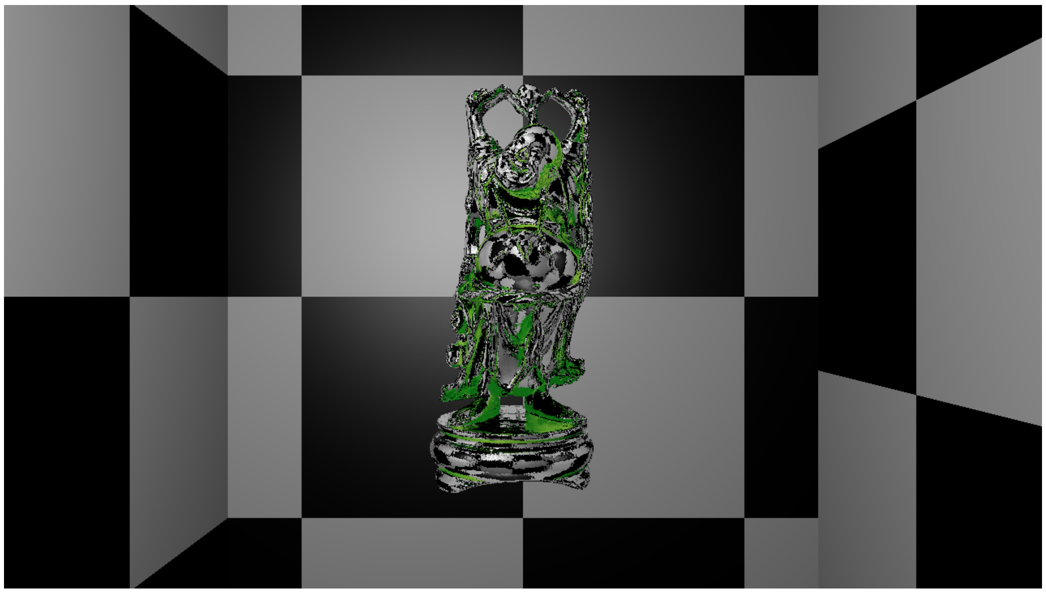


Figure 13: With out multiple bounces in the ray tracing, self reflections are not rendered correctly)

Currently reflections containing reflective objects do not work well, as can be seen in Figure 13 where the Buddha model's material was set to be 100% reflective and parts of the Buddha model are in the reflections. Furthermore, transparent objects aren't rendered correctly in reflections. To fix these issues, new rays need to be created when a reflective or transparent (i.e. refractive) object are hit during ray tracing. To handle these multiple collisions, or bounces, the R-buffer will need to be rendered multiple times per frame. When performing the ray tracing, a new ray can be created and added to a set for the next bounce. In this way multiple sets of rays can be created, one for each bounce. Each set of rays is then rendered into the R-buffer starting with the last bounce, and rendering each ray set in reverse order to the first bounce using the previous result as input to the current bounce.

References

[1] J. Bikker, "Real-time ray tracing through the eyes of a game developer," in *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, ser. RT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/RT.2007.4342584

[2] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, Jun. 1980. [Online]. Available: http://doi.acm.org/10.1145/358876.358882

[3] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: http://doi.acm.org/10.1145/361002.361007

[4] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Trans. Graph.*, vol. 26, no. 1, Jan. 2007. [Online]. Available: http://doi.acm.org/10.1145/1189762.1206075

[5] T. Saito and T. Takahashi, "Comprehensible rendering of 3-d shapes," in *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '90. New York, NY, USA: ACM, 1990, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/97879.97901

[6] M. Hargreaves, Shawn Harris, "Deferred shading," https://developer.nvidia.com/sites/default/files/akamai/gamedev/docs/6800_Leagues_Deferred_Shading.pdf, accessed: 01/03/2013.

[7] S. Kircher and A. Lawrance, "Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects," in *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, ser. Sandbox '09. New York, NY, USA: ACM, 2009, pp. 39–45. [Online]. Available: http://doi.acm.org/10.1145/1581073.1581080

[8] W. Engel, "Light pre-pass renderer," http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html, accessed: 01/03/2013.

[9] B. Mora, "Naive ray-tracing: A divide-and-conquer approach," *ACM Trans. Graph.*, vol. 30, no. 5, pp. 117:1–117:12, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/2019627.2019636

[10] A. T. Áfra, "Incoherent ray tracing without acceleration structures." in *Eurographics (Short Papers)*, C. Andújar and E. Puppo, Eds. Eurographics Association, 2012, pp. 97–100. [Online]. Available: http://dblp.uni-trier.de/db/conf/eurographics/eg-short2012.html#Afra12

[11] T. Pitkin, "Gpu ray tracing with cuda," Master's thesis, Eastern Washington University, 2013.

[12] C. Soss, "Ray traced rendering using gpgpu devices," Master's thesis, Eastern Washington University, 2013.

[13] T. L. Sabino, P. Andrade, E. W. Gonzales Clua, A. Montenegro, and P. Pagliosa, "A hybrid gpu rasterized and ray traced rendering pipeline for real time rendering of per pixel effects," in *Proceedings of the 11th international conference on Entertainment Computing*, ser. ICEC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 292–305. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33542-6_25

[14] J. P. G. Cabeleira, "Combining rasterization and ray tracing techniques to approximate global illumination in real-time," Master's thesis, Instituto Superior Técnico, Lisbon, Portugal, 2010.

[15] O. Wiki, "Image format," http://www.opengl.org/wiki/Image_Format, accessed: 02/06/2014.

[16] M. Pettineo, "Scintillating snippets: Reconstructing position from depth," http://mynameismjp.wordpress.com/2009/03/10/reconstructing-position-from-depth/, accessed: 02/06/2014.

[17] T. Möller and B. Trumbore, "Fast, minimum storage ray-triangle intersection," *J. Graph. Tools*, vol. 2, no. 1, pp. 21–28, Oct. 1997. [Online]. Available: http://dx.doi.org/10.1080/10867651.1997.10487468

[18] L. d. E. Gritz, "The importance of being linear," http://http.developer.nvidia.com/GPUGems3/gpugems3_ch24.html, accessed: 02/27/2014.

[19] T. K. G. Inc., "Arb_pixel_buffer_object," http://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt, accessed: 02/13/2014.

[20] S. University, "The stanford 3d scanning repository," graphics.stanford.edu/data/3Dscanrep/, accessed: 02/26/2014.

[21] "Open asset import library," http://assimp.sourceforge.net/, accessed: 02/13/2014.

[22] "Freeimage," http://freeimage.sourceforge.net/, accessed: 02/26/2014.

[23] "freeglut," http://freeglut.sourceforge.net/, accessed: 02/13/2014.

[24] "Eigen," http://eigen.tuxfamily.org/, accessed: 02/13/2014.

Libraries Used

The following libraries were used to facilitate development of the renderer:

Assimp 3.0.1270 [21]
Assimp is an open source library for loading 3D model formats. It loads various well known formats into a uniform structure. It was chosen for its ease of use and wide support of formats. Assimp also offers many post-processing options to make the imported data easier to use and more uniform.

FreeImage 3.15.4 [22]
FreeImage is used to load image files into byte arrays that are passed into OpenGL as textures.

Freeglut 2.8.1 [23]
FreeGLUT is an open source alternative to the popular but depricated GLUT library. It simplifies the process of creating a window and setting up an OpenGL context. It was chosen for its simplicity and many easy to find examples.

Eigen 3.2.0 [24]
Eigen is a linear algebra library with many features beyond simple 3D math. It uses expression templates and SIMD optimizations to provide a fast and easy to use feature set. It was chosen for its performance to handle the performance sensitive ray tracing.

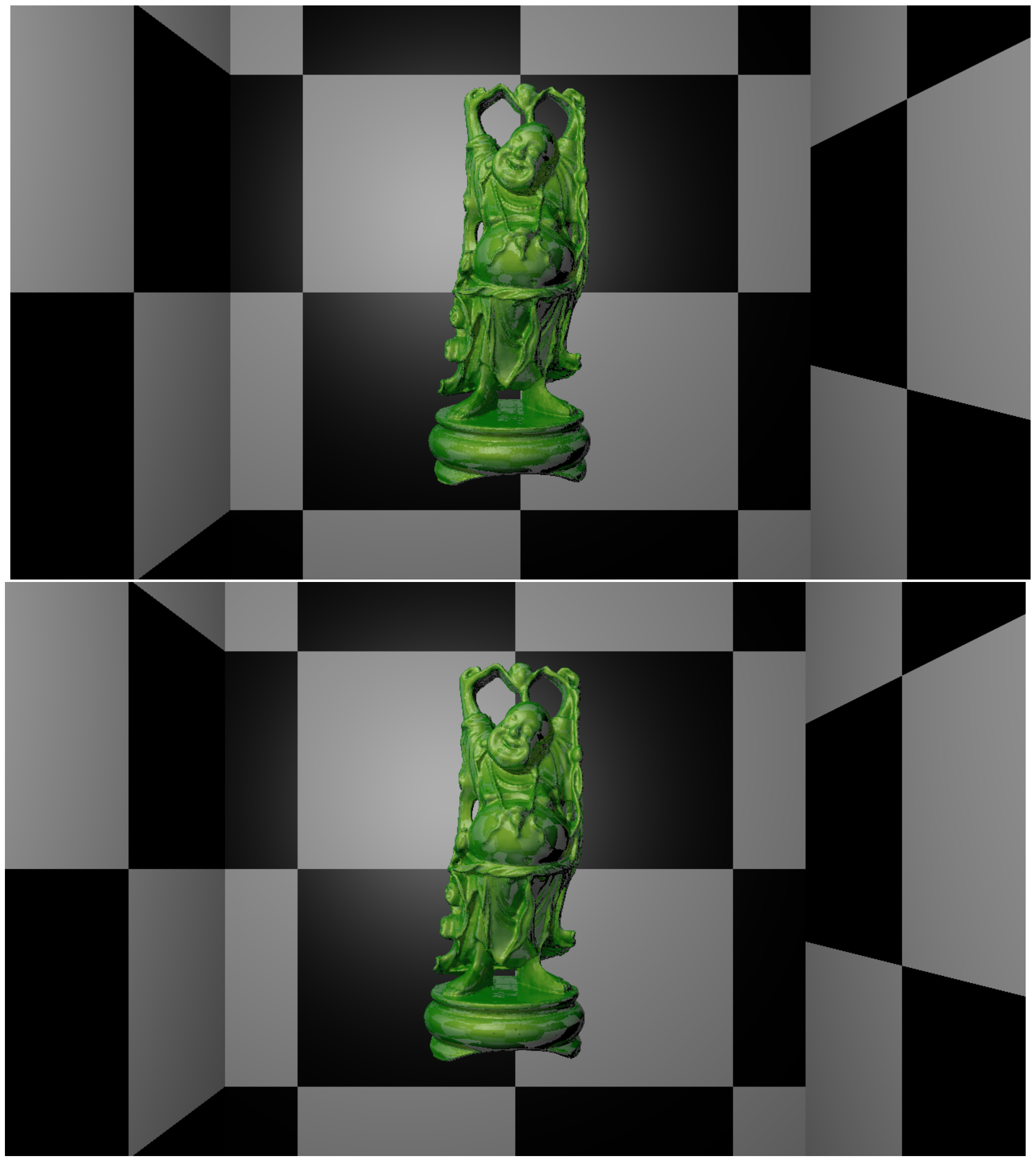


Figure 11: The Stanford Happy Buddha model with 1,087,716 triangles (top) and 290,633 triangles (bottom)

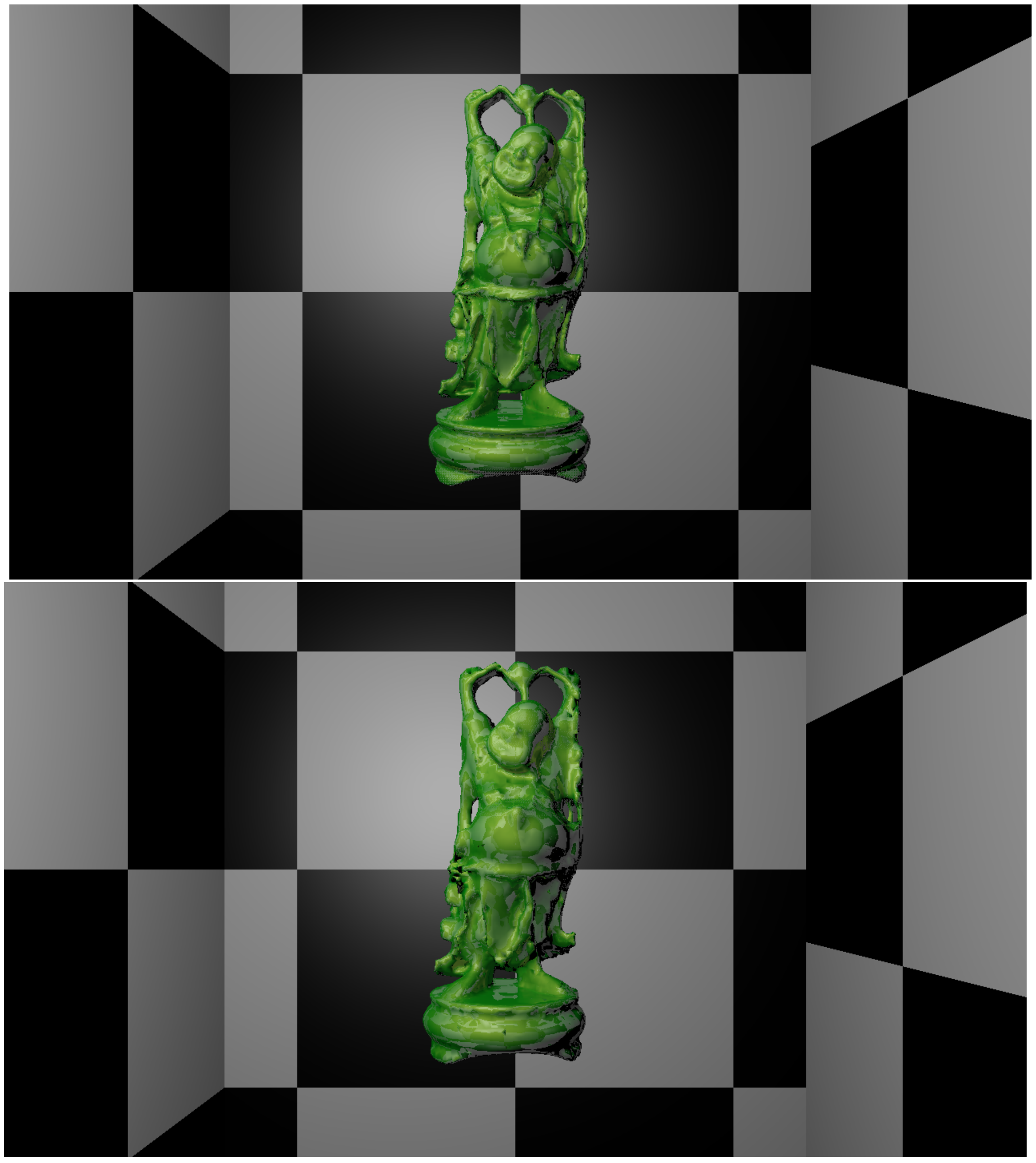


Figure 12: The Stanford Happy Buddha model with 65,590 triangles (top) and 14,765 triangles (bottom)

Vita
Daniel Stokes

Education

- Associate of Arts, 2009, Spokane Falls Community College
- Bachelor of Science in Computer Science, 2012, Eastern Washington University

Honors and Awards

- Graduated with Honors, 2009, Spokane Falls Community College
- Graduated Summa Cum Laude, 2012, Eastern Washington University

Work Experience

- Computer Science Graduate Assistant, 2012-2014, Eastern Washington University

  Taught Computer Literacy Applications course

- Google Summer of Code Student, 2013, Blender Foundation

  Fixed bugs and implemented a level of detail system for the Blender Game Engine

- Computer Science Lab Tutor, 2010-2012, Eastern Washington University

  Assisted students with topics including basic programming, data structures, Python, Java, and C

- Google Summer of Code Student, 2011, Blender Foundation

  Improved the user interface and implemented various small features into the Blender Game Engine including improved font support, vertex buffer objects, and collision masking