Design and Comparison of Parallel Ray-Tracing Algorithms

BY

Nicholas K. Faulconer

of

Wheaton College

in Partial Fulfillment of the Requirements

for

Graduation with Departmental Honors

in Computer Science

Norton, Massachussetts

May 14, 2012

Abstract

The graphical capabilities of computers have advanced to the point where life-like visuals are the norm, and moviegoers and video gamers are constantly demanding increasingly realistic scenes. Much of the photorealism in computer-generated images comes from the chosen lighting algorithm, the series of instructions that tell a computer how to apply lighting effects to a three-dimensional scene. While numerous lighting algorithms have been designed and implemented, ray-tracing, a lighting algorithm that accurately emulates the real-world behavior of light, remains a popular choice for developers seeking especially realistic visuals. However, the sheer amount of time and processing power ray-tracing requires has limited its applications to movies and other offline applications in which individual frames are slowly rendered and then compiled to create the illusion of motion, just like individual pictures on a reel of film. This paradigm has changed in recent years, partly due to improvements in hardware but mostly due to cleverly-designed variations to the ray-tracing algorithm capable of generating photorealistic images in real time, at the rapid rate the user needs them on consumer-grade machines. In general though, these algorithms are designed for high-end machines and dedicated gaming systems, limiting their availability to the average user. Our new ray-tracing algorithm, Frosty, distinguishes itself by providing accurate and fast rendering on low-end PCs, which for the purposes of this paper is defined as a dual-core machine with no dedicated GPU. The new algorithm is shown to be a fast but accurate implementation of ray-tracing that works in parallel to generate photorealistic scenes on such low-end systems.

Acknowledgments

I would like to thank my professors: Mike Gousie for putting up with my nonsense over the past year, Mark LeBlanc for convincing me to write a thesis in the first place, Tom Armstrong for the use of his equipment, and Josh Stenger for his invaluable support in revising this document to make it more accessible to readers.

I would also like to thank Dave Schloss and Josh Campbell for the encouragement and support they provided. I'd like to thank Björn Craig-Muller, James Green, and Melissa Darnley for distracting me when I should have been working on this thesis instead. And lastly, I'd like to thank my family for way too many things to list here. None of this would have been possible without any of the people mentioned here!

Contents

List of Figures

List of Tables

1 Introduction

A ray-tracer is a computer graphics algorithm that simulates real-world lighting conditions by tracing the paths of individual rays of light as they travel through a computer-generated three-dimensional scene. A well-designed ray-tracer produces outstanding, almost lifelike images. Unfortunately obtaining this high level of quality requires such a large number of computations that the average computer would take quite a long time to generate a simple image. This is an inherent flaw in the ray-tracing algorithm that leaves the ability to achieve such fantastic realism within any reasonable timespan to those who can afford to purchase and maintain high-end machines (a difficult concept to define due to the rapid pace at which computer technology changes, but for our purposes a high-end machine is one equipped with dedicated graphics hardware responsible for at least 50% of the machine's retail price or US$800). Of course, any decent computer scientist can tell you that your hardware will only carry you so far, meaning that in order to achieve real-time scene generation we need well-designed software.

The need for such well-designed software is the primary motivation behind this research. In recent years, physical limitations in electronic circuit design have resulted in a decrease in computation power available to a single processor. In response to this, chip manufacturers have started producing multicore processors - several processors packaged together on one chip. Yet very few applications actually make use of this added processing power, as application developers generally feel more comfortable programming in serial, which is to say, developing applications designed to run on a single processor. Unfortunately, this results in a great deal of wasted computing potential, depending on the number of "cores" available to a given machine. For example, if a machine has a total of four cores, and only one is used at any given time, this means that upwards of 75% of the machine's potential is wasted. Since the ray-tracing algorithm requires a lot of processor time, the obvious solution is to design a variation of the ray-tracing algorithm that runs in parallel, thereby utilizing much of that wasted computing potential.

Of course, computing ray-tracing in parallel is hardly a new idea, as we'll discuss later in Chapter 3. However, there are generally several ways to modify an algorithm or program to run in parallel, and not all of them are always immediately obvious. A great deal of research and effort has already gone into the subject of parallel raytracing. In chapter 2 we will look at a history of raytracing, starting from its earliest use in the mid 80's, providing descriptions of how the algorithm works and how it has evolved over the years. Chapter 3 will then look at previous attempts to parallelize this algorithm, with special mention of their significance, how they succeeded, and potential areas of improvement. Chapter 4 documents our earliest attempts at writing a simple raytracer, followed by the main focus of our research: speed through parallelization. We present our findings in Chapter 5. Chapter 6 presents a follow up discussion of our work and presents the conclusions from our research. Following this presentation, chapter 7 presents a discussion of what the future holds, and where there is room for improvement in our research.

2 What is Ray-tracing?

Ray-tracing, not to be confused with ray-casting, illuminates a computer-generated scene by following simulated rays of light as they travel through the scene. The algorithm was first designed to provide accurate shading to computer-generated scenes. In its simplest form, these rays are generated at the center of projection (the "camera") and pass through the view plane (a screen or window onto which the three-dimensional scene is projected), illuminating objects in the scene they bump into [1].

Most modern ray-tracers are designed to have these rays continually reflect ("bounce") off objects and, in the case of glass and other transparent objects, refract ("bend") through them. This allows for the generation of natural-looking scenes with smooth shadows, believable interaction between light and objects, and proper distortion of objects obscured by glass or water. The pseudo-code for a simple ray-tracer follows, along with a detailed explanation containing the definitions of some of the terms used:

```
select center of projection and window on viewplane;
for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray from center of projection through pixel;
        for (each object in scene) {
            if (object is intersected and is closest considered thus far) {
                record intersection and object name;
            }
        set pixel's color to that at closest object intersection;
    }
}
```

[3]

Note that this is not a step-by-step set of specific instructions for the com-

puter to follow, but rather an abstraction of the algorithm for the programmer to use as a guide. That said, looking at the pseudocode in detail provides a clear understanding of exactly how the algorithm works and provides insight into how to implement the algorithm. With that in mind, a line-by-line analysis of this pseudocode follows (see Figure 1 for a visual aid):

- Setup the window to draw the scene in, and the camera to view the scene through.
- Iterate through each scanline S in the window. A scanline is a horizontal row of pixels on a screen.
- Go through each pixel P in the current scanline S. A pixel is a tiny block of color on the screen.
- Generate a new ray R. Calculate the path of R such that R originates from the camera and travels through P.
- For each object O in the scene, check if R's path intersects O at any point.
- If R intersects O, calculate where along R's path the ray intersects object O.
- For each object intersected by R, choose the object closest to R's origin point.
- Calculate which color P should be assigned based on the object(s) intersected by R.

This analysis still abstracts most of the processing done by the computer. For example, a great deal of math goes into determining whether a ray intersects an object. First, the computer needs to know every point along the path of the ray; it computes this with the equation

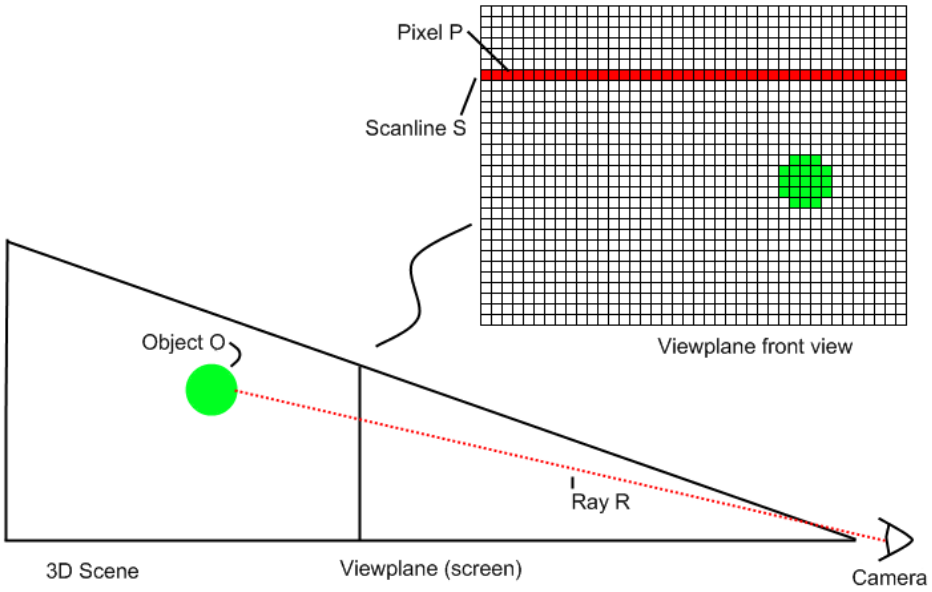

Figure 1: *Visual guide of the ray-tracing algorithm. Each object mentioned in the algorithm is pictured and labeled.*

$$x = x_0 + t\Delta x, y = y_0 + t\Delta y, z = z_0 + t\Delta z$$

where

(x, y, z) represent each point along the ray depending on some value t,

(x_0, y_0, z_0) represent the ray's origin point in three-dimensional space,

$(\Delta x, \Delta y, \Delta z)$ are defined as $(x_1 - x_0, y_1 - y_0, z_1 - z_0)$ for simplicity.

From here, the ray-tracer needs to check if this ray intersects any object. The equation varies depending on the type of surface intersected. The equation for testing ray-sphere intersection is

$$(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + 2t[\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] + (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0$$

where

r is the radius of the sphere,

(a, b, c) represents the center of the sphere in 3D space.

This formula can be solved using the quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

and the number of real (non-imaginary) roots describes the behavior of the ray. If the solution involves no real roots, then the ray does not intersect the sphere; one real root means the ray grazes (is nearly parallel to) the sphere; and two real roots means the ray intersects the sphere at those positions, where the solution that yields the smallest positive value for t represents the point of intersection closer to the camera. For a graphical representation of the behavior of this algorithm, see Figure 2. The equation for testing intersection with a polygon, also known as a plane, defined by three points in three-dimensional space, is

$$t = -\frac{(Ax_0 + By_0 + Cz_0 + D)}{(A\Delta x + B\Delta y + C\Delta z)}$$

where A, B, C, and D are obtained from the plane equation, $Ax+By+Cz+D = 0$ [3].

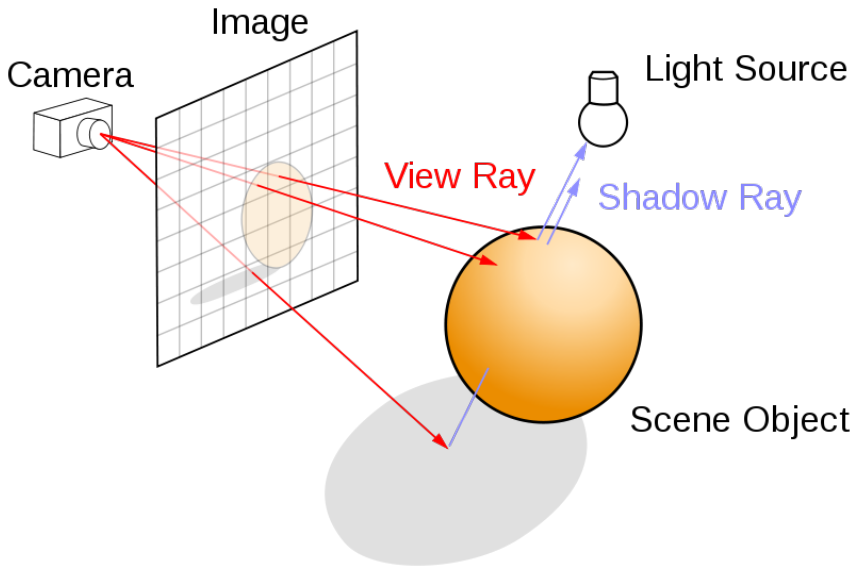

Figure 2: *Diagram showcasing behavior of the ray-tracing algorithm. View rays originate from the viewpoint (camera), pass through the view plane, hit the sphere, and produce secondary rays that actively seek out the light source. [8]*

The basic algorithm described above does not produce all of the visual effects desired. In fact, this algorithm would only serve as a hidden surface removal al-

gorithm, meaning that it would only draw objects that should be visible from the current camera view (objects with other objects between them and the camera will not be drawn). There are easier and computationally cheaper algorithms that can perform this same function, so clearly this algorithm is insufficient for our needs. In order to generate reflection and refraction effects, modifications must be made to the algorithm:

```
select center of projection and window on viewplane;
for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray from center of projection through pixel;
        pixel = RT.trace (ray, 1);
    }
}

/* Intersect ray with objects and compute shade at closest intersection. */
/* Depth is current depth in ray tree. */

RT_color RT_trace (RT_ray ray, int depth)
{
    determine closest intersection of ray with an object;

    if (object hit) {
        compute normal at intersection;
        return RT_shade (closest object hit, ray, intersection,
                                    normal, depth);
    } else
        return BACKGROUND_VALUE;
}   /* RT_trace */
```

```
/* Compute shade at point on object, tracing rays for shadows,
     reflection, and refraction. */
RT_color RT_shade (
    RT_object object,            /* Object intersected */
    RT_ray ray,                  /* Incident ray */
    RT_point point,              /* Point of intersection to shade  */
    RT_normal normal,            /* Normal at point  */
    int depth)                   /* Depth in ray tree */
{
    RT_color color;              /* Color of ray */
    RT_ray rRay, tRay, sRay;     /* Reflected, refracted, and shadow rays */
    RT_color rColor, tColor;     /* Reflected and refracted ray colors. */

    color = ambient term;
    for (each light) {
        sRay = ray to light from point;
        if (dot product of normal and direction to light is positive) {
            compute how much light is blocked by opaque and transparent
            surfaces, and use to scale diffuse and specular terms
            before adding them to color;
        }
     }

    if (depth < maxDepth) {      /* Return if \emph{depth} is too deep. */
        if (object is reflective) {
            rRay = ray in reflection direction from point;
            rColor = RT_trace (rRay, depth + 1);
            scale rColor by specular coefficient and add to color;
```

```
        }
    }

    if (object is transparent) {
        tRay = ray in refraction direction from point;
        if (total internal reflection does not occur) {
            tColor = RT_trace (tRay, depth + 1);
            scale tColor by transmission coefficient and add to color;
        }
    }
    return color;                /* Return color of ray */
} /* RT_shade */
```

[3]

The modified algorithm is a bit complex for an analysis like we did with the simpler version. The most important addition is the ability for rays to reflect off and continue to trace through objects, allowing for reflective surfaces, transparent refractive objects, and the casting of shadows by objects. The adjusted algorithm also produces shading effects on each surface, reducing the intensity of the object's color based on how much light it is getting. These effects combine to produce the realism so greatly valued in computer generated graphics.

3 Previous Work

Ray-tracing first began as a method to provide shading to machine-generated line drawings, rather than images rendered to a monitor, where a plus sign indicated a darker region. The only goal of this very early implementation was determining a surface's brightness relative to all the other surfaces in the drawing; as such color and light intensity were not factored into any calculations. Although a great step forward in shading technology, this early design had several limitations, chief among them that it took "several thousand times as much calculation time" compared to standard drawing methods [1]. While not specifically related to graphical applications, it should be noted that in the late 1960's to early 1970's, a technique similar to ray-tracing was used to simulate the splitting of atoms [3]. In 1980, Turner Whitted designed a ray-tracer capable of generating shadows, specular reflection, and refraction. While capable of generating some particularly stunning effects (for the time), Whitted designed his ray-tracer for human simplicity, not machine efficiency, and as such it featured very few mathematical optimizations. By his own timing information, calculating intersections took up about 75% of CPU time, and a simple scene of three spheres took 44 minutes to render [13]! In order to deal with this problem, computer scientists needed an optimized ray-sphere intersection equation. To this end, research led to a modified version of the equation which required fewer calculations and variables [7].

What's a computer scientist to do with an exciting new algorithm? Why, speed it up, of course! Rewriting an algorithm to run on multiple cores is not often an easy task with a definite solution. Consequently, numerous methods of parallelization have been proposed over the years. Among the simplest of these methods is a ray-tracer that uses multiple cores to render multiple frames at a time, with each available core processing a different frame from the same animation. While this method works well in circumstances where the computer knows ahead of time what the scene looks like at a given time interval (for example, a CGI movie), it is completely unacceptable for an interactive environment, such as a

game, where user input makes rendering frames ahead of time impossible [4].

Priol and Bouatouch describe two parallelized methods to reduce ray tracing execution time. In one of these methods, the ray-tracer divides work among the processors based on the intersections of rays; the other method splits work by subdivisions in 3D space. The authors also propose a load-balancing system based on partitioning the work based on the results of sub-sampling (computing only the first rays of) the scene [10].

While most parallel approaches to ray tracing involve assigning different processors to work on different pieces of the screen, alternate approaches involve distributing computational work. Busy processors can have some of their tasks transferred over to idle processors, resulting in a more balanced distribution of work [6] [2], however such methods are limited by the large memory requirements and the amount of overhead generated by inter-processor communication [11].

4 Methodology

This chapter describes our ray tracer in detail and the steps we took in its design process. It begins with a description of the earliest stages of design and the transition from sequentialism to parallelism. We then go into detail about our first attempt at a parallel ray tracer, in which the program distributes work amongst the computer's processors based on screen sub-divisions. We conclude the chapter by describing some of our attempted methods to enhance this basic parallelism and more evenly distribute work amongst all the processors of a machine, an important task known as load-balancing.

Before delving into that, however, we feel an overview of some of the challenges involved with parallelism in general is a good idea. There are three terms that get thrown around a lot during discussions about parallel processing: load-balancing, overhead and bottlenecks. In order for an application to successfully utilize all of the processing power available on any given machine, the program must divide and distribute the work to be performed among each available processor. This process is referred to as load-balancing, and a sub-optimal load-balancing technique leads to sub-optimal performance. Overhead refers to any extra processing required to achieve parallelism that would otherwise not be a factor during sequential processing on a single core. Overhead factors in the time taken to perform any load-balancing methods as well as any inter-processor communication. Problems arise when overhead causes a parallel implementation - which should reduce execution time - to take longer than the original sequential application. Bottlenecks occur when multiple threads attempt to access or modify the same data in memory at the same time. As only one processor can access a given location in memory at a time, the other threads are forced to idle until that memory location becomes available again. This problem is made even worse in the case of a "race condition", when one thread overwrites a value in use by another thread, producing incorrect results. This problem can be thought of as going to pay for something, only to find out somebody has replaced the money in your wallet with play money. The

only way to deal with a race condition is to force one thread to wait for the other to finish, which will decrease performance.

4.1 First Steps

Rather than start from the ground-up, it was decided that a better use of time would be to search the internet for an existing, open-source ray-tracer. Several implementations were found with advanced visual effects and optimized code for faster run times. In the end, we eventually settled on an implementation available as part of an online ray-tracing tutorial [12]. We began by familiarizing ourselves with the existing code base and, after we had a basic understanding of the ray-tracing code, started modifying it to suit our needs. In its current state, the code was only capable of drawing spheres; polygonal structures were beyond its capabilities. We attempted to add the ability to draw objects defined by a set of polygons. We were partially successful: although planes were not being drawn in the scene, their reflections were visible on nearby spheres (as a metaphor, imagine an object in a room that you could only see by looking at its reflection in a mirror). Although we found this bizarre behavior somewhat interesting, since this particular feature wasn't of any real importance to our research, we decided to abandon this pursuit, instead focusing on adding interactivity to Frosty. The unmodified code we obtained did not draw graphics to the screen in real-time; instead, it merely dumped the results of the ray-tracing to an image file. This was unacceptable, since our goals required the ability to interact with the environment while the program was running. We decided to use OpenGL, an open-source graphics library, to achieve the interactivity we needed, mostly due to our familiarity with it. Our original plan was to run the ray-tracing code, store the results to an internal texture (an image file applied to a flat polygon) in memory, and have OpenGL draw this texture. This did not work: results were generated slowly and they were often incorrect. Rather than invest a lot of time debugging the problem, we attempted a different approach: instead of the roundabout route of drawing to a

texture and drawing that texture to the screen, we opted to place the results of the ray tracing directly into the OpenGL framebuffer pixel-by-pixel, after which the framebuffer would be drawn directly to the screen. As a metaphor, compare the texture-based approach to hanging a painting on a wall, and the second method to painting directly on the wall. While this worked, it added significant slowdown to the code and, in fact, took longer to complete than the actual ray-tracing portion. The ultimate solution to this problem was to combine both methods: for each pixel on the screen, the program saves the results of the raytracing calculation to memory and then transfers the completed image to the OpenGL framebuffer at once, allowing for basic interactivity (adjusting the camera and lighting) with the scene. This process does not slow down the overall performance of the program by any noticeable amount of time.

With the basics out of the way, work could finally begin on the true goal of our research: improving the performance of the raytracing algorithm through parallelization. While our raytracer, Frosty, was interactive in theory, it took several seconds to draw even a simple scene, making true interactivity impossible in its current state. In order to speed things up, it would be necessary to know which portions of the code required the greatest amount of CPU time. To this end, we timed several pieces of the code to see which had the longest execution times and discovered, just as Whitted had in 1980, that ray-sphere intersection computations took responsibility for nearly 75% of the program's execution time. We began researching other attempts at speeding up the raytracing algorithm, specifically looking for parallelization attempts. In order to achieve our goals, we would need to know what had been done in the past so as to know what had and had not worked, thus being able to improve upon successful techniques. These first efforts to implement parallelism within Frosty are explained in detail in the Section 4.2 below.

4.2 Screen-Space Parallelism

With the basic ray-tracing features taken care of, work began on modifying Frosty to run in parallel. For the first test, we decided to implement a screen-space approach to parallelism; that is to say the work is divided amongst each processor based on the location of objects on the screen. The work is split into two threads: one processor handles the top half of the screen, and another processor handles the bottom half of the screen. Applying this method to a test scene of 100 randomly-placed spheres (see Figure 3), execution time decreased by roughly half. Such a 50% reduction in runtime is to be expected, since half the work is being done at the same rate. In order to test Frosty's new capabilities, we needed to attempt to "break" this implementation, so that we might be able to determine a better parallel approach and improve performance. We applied several test scenes which we believed might result in poor runtime performance. Since this approach to parallelism works by splitting the screen into two halves, top and bottom, several test scenes were run through Frosty (see Figures 4, 5, 6, and 7) that exploited Frosty's inability to adjust the division of work among threads in the event that one half of the screen in completely empty. We then compared the results of these parallel tests to our earlier results taken from the sequential version of Frosty. As a control, we also tested three balanced scenes in which both halves of the screen were a mirror-image of the other to ensure the timing for both threads were identical (see Figures 8, 9, and 10) and a scene with nothing to draw, which would serve as a benchmark for how much time a thread should take to run when it has no work to do. Our initial results were promising, showing a dramatic speed-up in the parallel execution when compared to the original sequential implementation. More detailed information, including precise timing information, can be found in Section 5.

Looking over the results of some of the tests involving the "unbalanced" scenes, however, we found some confusing and unexpected timing data. Having worked with parallelism before, we expected the parallel implementation to take

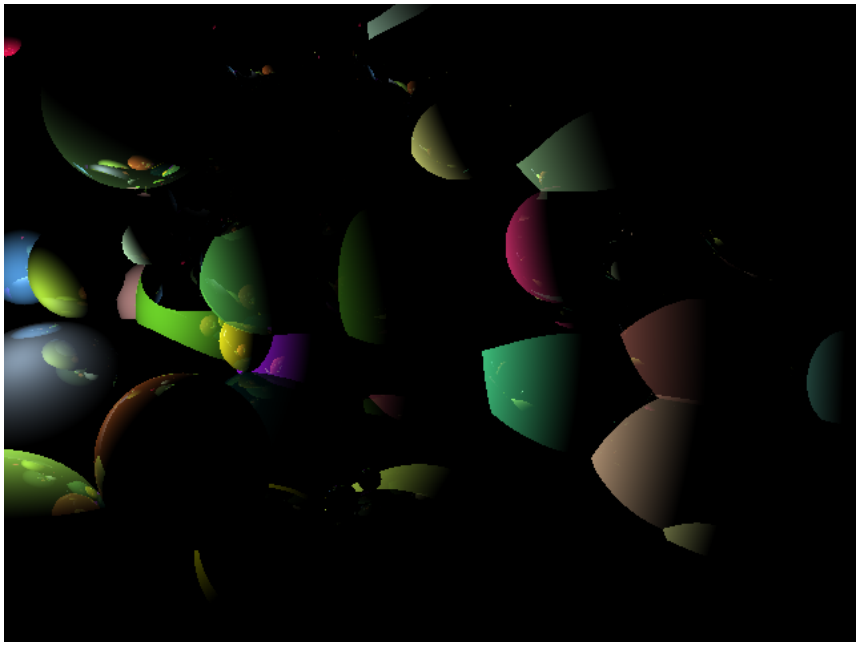

Figure 3: *First test scene, 100 spheres. The positioning and characteristics of each sphere was generated at random.*

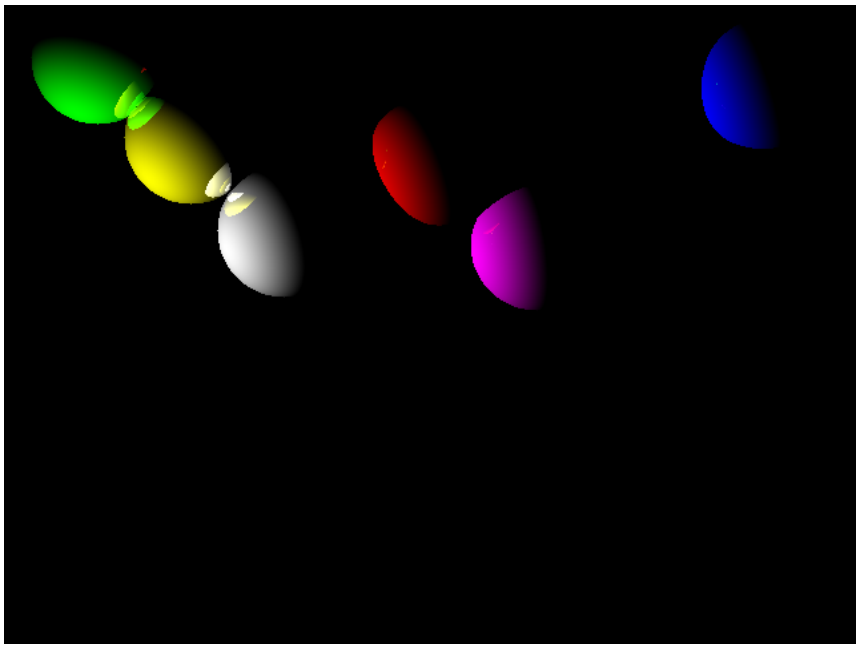

Figure 4: *Spheres at the top of the screen. The purpose of this scene was to time the multithreaded raytracer on an unbalanced scene.*

longer than the sequential ray-tracer on these tests, since one processor was doing all the work (as in the sequential version) in addition to the overhead, or additional communication and work required for multicore threading. However, the results

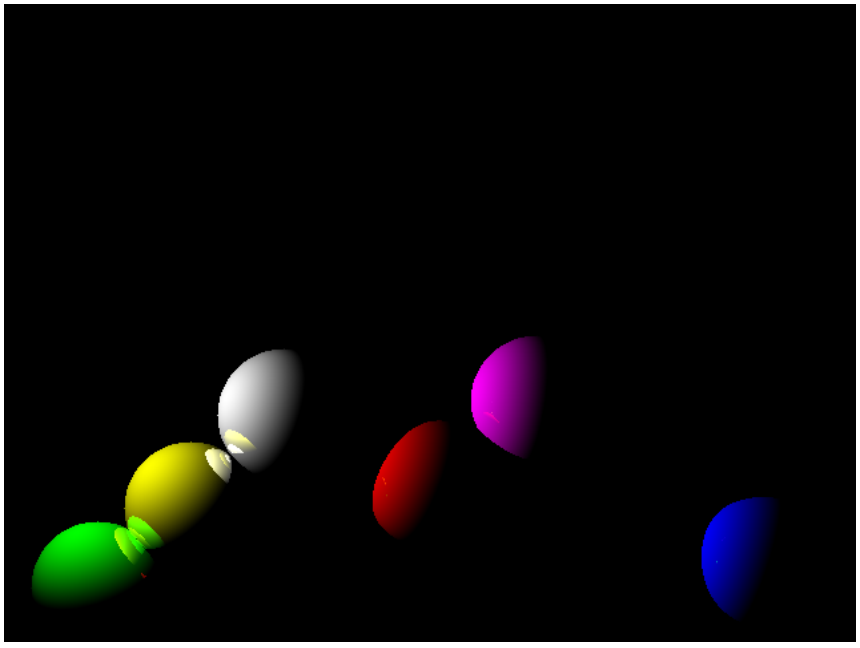

Figure 5: *Spheres at the bottom of the screen. This is a mirror-image of the previous image, tested for the same reasons.*

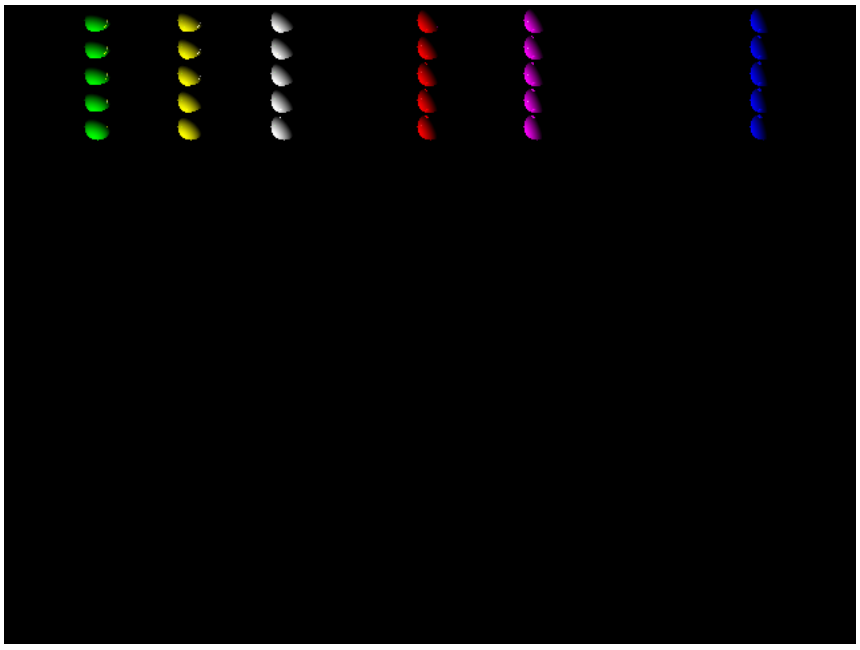

Figure 6: *Even more spheres at the top of the screen. We wanted to slow down the ray-tracer by giving it more than six spheres to draw, so this scene has one thread draw 30 spheres while the other sits idle.*

indicated otherwise, as we found the thread that should have been idle was taking nearly as much time to process as the busy thread. Thinking back to previous

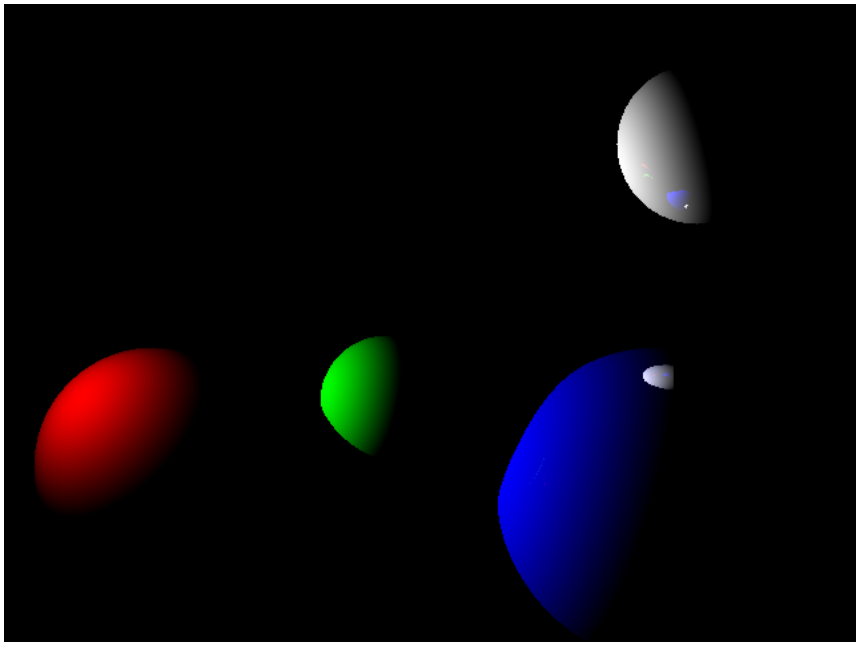

Figure 7: *Unbalanced scene. Both threads have drawing work to do, but clearly one thread has much more work to do than the other.*

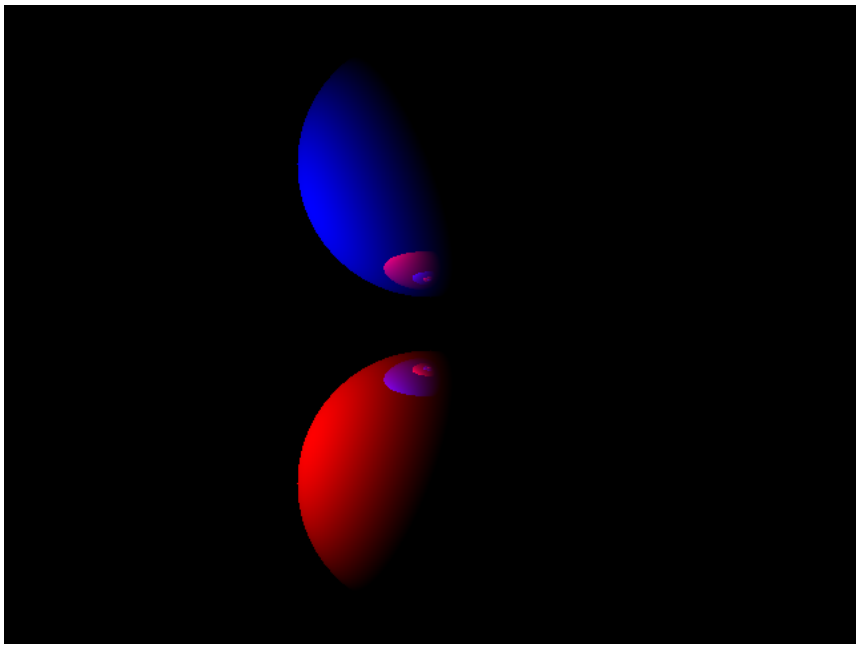

Figure 8: *Balanced Scene. Both threads must do the exact same amount of work, as both have one sphere with the same relative positioning and reflective properties.*

research, we re-added the intersection timing code and verified that the ray-sphere intersections still accounted for 75% of the CPU time. We attempted a quick fix by having the ray-tracer only look through spheres within a thread's own drawing

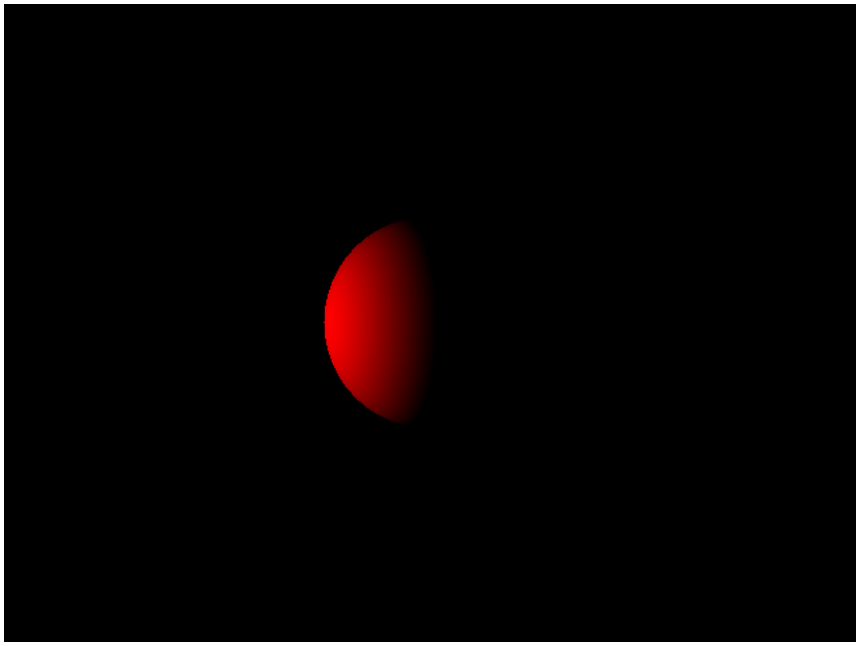

Figure 9: *Lone Sphere. Although there's only one sphere in this image, each thread is responsible for exactly one half of the sphere, so this is a balanced scene.*

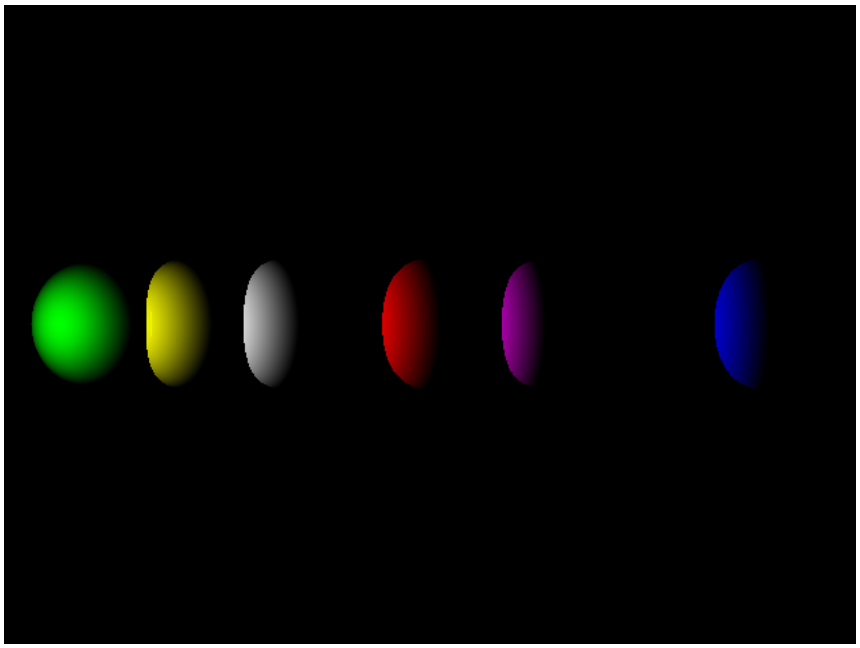

Figure 10: *Row down the middle. Similar in principle to the previous scene. Notice that the spheres are so close to each other that, depending on their point along the z-axis, the shaded regions of some spheres appear to distort the shape of the spheres behind them.*

range. Not only did this produce incorrect results, it actually ended up increasing intersection calculation time to about 95% of execution time! Regardless, this event marked a significant point in our research, as we now had a definitive element to focus on in an attempt to speed up execution time. If we could cut down the wasted intersection tests each thread performed, we could, in the best possible case, cut down execution time by another 50%.

4.3 Reducing Ray-Sphere Intersection Calculations

Parallelization is one of the two main approaches to speeding up ray-tracing, with the other being reducing ray intersection calculations [9]. Since the primary interest of our research is mostly in parallel processing, we intend to only discuss reduction of intersections as it applies to a multithreaded approach to ray tracing. To this end, it was first necessary to devise a reliable method of determining which spheres reside inside a thread's particular screen sub-division. We operated under the assumption that the first approach, described above, was sound in theory, but required more thought, rather than rushing into the implementation phase. We decided upon a set of important guidelines we considered necessary for such an implementation to be successful:

1. Threads only need to know about spheres within their own sub-division for the first "batch" of rays, coming directly from the camera.

2. Since threads are responsible for rays generated within their sub-division, they must be aware of all spheres in the scene for each subsequent "bounce" after the first. For example, if a ray originates from the camera and intersects with a sphere within thread 1's sub-division, and that ray then reflects and intersects a sphere in thread 2's sub-division, thread 1 is still responsible for tracing that ray.

3. Spheres are volumes, not points; as such they can cross multiple sub-divisions. In other words sphere size needs to be a factor when cutting spheres off.

4. The screen only has X and Y dimensions; however, spheres and rays are in 3D space and have a Z component as well. The Z dimension needs to be considered as well.

As it turns out, the "quick fix" code violated all of these guidelines. But in order to meet these requirements, the computer now had several extra calculations it would need to perform, and there was some uncertainty whether it would

actually be worth it. Ray intersection tests represent such a significant portion of execution time not because of computational complexity, but because of the sheer number of rays that need to be traced. If the sphere-rejection algorithm takes more time to compute than the ray intersection equation, it not only wastes CPU time, but also valuable research time otherwise spent doing more important tasks. Although we found this lead to be a potentially valuable area of research, after further thought and analysis of existing works, we determined our time would better be spent on our original parallelization focus. This observation may prove to be an interest subject for future research.

4.4 Load Balancing

In order to achieve fully optimal speeds, the new ray-tracer required a way to evenly distribute computational work amongst a machine's processors reliably. Proper load-balancing is important because a program is only as fast as its slowest thread. If one thread has 5 minutes worth of computations and another has only 1 minute, then that second core is going to waste for four minutes and the entire program takes five minutes to execute; if the work was equally distributed each core would be working for exactly three minutes, and the program would only take three minutes to execute. Walking is an apt metaphor: both legs should be synchronously performing the same amount of work, and if one leg suddenly stopped in the middle of a walk, it would be a good indication that something was wrong. The method described above, in which the screen is divided based on the number of cores and each core is assigned a separate piece, fails to adequately divide work among each processor, resulting in wasted processing potential and time. Its one advantage is that it's fast and requires practically no overhead, but the the potential to have one or more cores completely idle while another handles the entire workload is a serious problem. We would need a load-balancing solution to ensure each core was doing an equal amount of work. Since load-balancing requires CPU time just like everything else, it would be necessary to ensure the load-balancing code didn't take longer to execute than the unbalanced code. Thus, the load-balancing code needed to be fast, efficient, and accurate. Since our research focused on a screen-space sub-division approach to parallelization, we decided to design a load-balancing approach similar to the one proposed by Priol and Bouatouch, in which the scene is sub-sampled and then partitioned based on the amount of rays generated by the sub-sampling [10]. Since we were concerned that this method may require a lot of overhead, we also developed a load-balancing system based on the size and positions of spheres to use as a benchmark to measure the performance of the sub-sampling method and determine if the extra overhead would be worth it.

We used a technique known as binary space partitioning to generate sub-divisions on the screen. Binary space partitioning involves continuously splitting the screen into pieces by dividing an existing sub-division into two separate pieces based on some specific criteria. For our ray tracer, the goal of our binary space partitioning algorithm is to generate a number of sub-divisions equal to the number of cores available to the machine, with each sub-division having to perform roughly the same amount of computations. Figure 11 demonstrates how this process might work for a simple scene on a quad-core machine.

Our binary space partitioning algorithm achieves optimal sub-divisions by following a few simple steps. For this discussion we are assuming that the program has already calculated the approximate work required for the entire scene. The algorithm begins by sub-dividing in half based on the area of the current sub-division. The dividing line is then shifted towards the portion of the sub-division that requires more computations (either more/larger spheres or a greater number of ray-sphere intersections in that region) until each region has (roughly) the same amount of required computations. This process is then repeated for each new sub-division until the number of sub-divisions we have matches the number of cores available on the current machine. Note that this algorithm assumes that the number of cores is a power of 2 (1, 2, 4, 8, ...) and so our algorithm may not produce a completely load-balanced solution on machines with a number of cores that does not match the program's expectations.

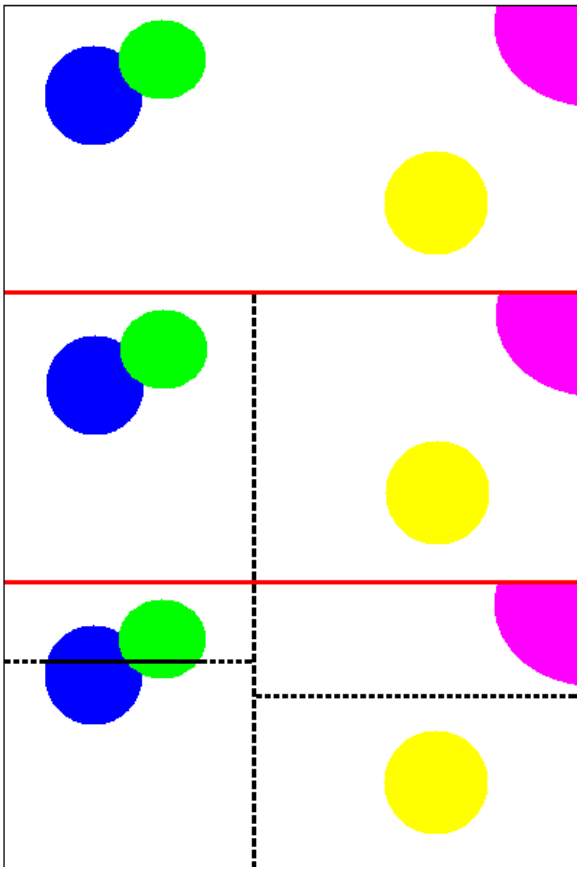

Figure 11: *Sample Binary Space Partitioning; Quad-Core Machine. This image demonstrates the sub-divisions generated by a binary search partitioning algorithm. The solid lines indicate sub-divisions while the dotted lines differentiate stages in the algorithm (sub-divisions that intersect spheres are drawn as solid lines). We start off with no sub-divisions and then divide the scene into two sub-divisions. From this point, we sub-divide both sub-divisions into two more sub-divisions each, resulting in four sub-divisions total, one for each core. Notice that each sub-division is of unique height and width.*

5 Results

The results from our first parallelization efforts were very promising. Table 1 compares the average timing results of the sequential ray-tracer to those of the parallel ray-tracer, and Table 2 compares execution time between the two threads. The results of Table 1 are consistent and encouraging, showing a decrease in execution time across the board. The times recorded in Table 2 raised more

questions than answers, however. Comparing the timing of the two threads, the ray-tracer performed as expected for the balanced tests: both threads took the same amount of time to execute using similar data.

	Sequential Execution Time	Parallel Execution Time
Scene 1	7.237	4.245
Scene 2	0.340	0.185
Scene 3	0.273	0.185
Scene 4	0.989	0.573
Scene 5	0.257	0.176
Scene 6	0.134	0.078
Scene 7	0.071	0.043
Scene 8	0.268	0.150
Empty	0.026	0.020

Table 1: *Average timing comparison between sequential and dual-threaded ray-tracer, in seconds. Timing information was obtained by counting the number of CPU clock cycles from beginning to end of execution and then dividing by the CPU's clock rate. Averages were computed by drawing each scene 100 times and rounding off to the third decimal place.*

As the tables and charts show, we were able to attain significant speedup in all of our test scenes, with speedup reaching as high as 170% in some cases. We hoped to achieve even greater performance gains with improved load-balancing algorithms, but initial results were discouraging. Not only did our load-balancing algorithm fail to evenly distribute the workload among the various processors, the additional overhead resulted in a net increase in execution time. The execution time for each thread on a dual-core machine, as well as the additional overhead required, can be seen in Table 3.

We achieved somewhat better results using the subsample-based approach to

	Thread 1 Execution Time	Thread 2 Execution Time
Scene 1	4.484	4.703
Scene 2	0.109	0.172
Scene 3	0.188	0.109
Scene 4	0.516	0.563
Scene 5	0.093	0.156
Scene 6	0.062	0.062
Scene 7	0.094	0.094
Scene 8	0.141	0.141
Empty	0.031	0.031

Table 2: *Timing comparison between threads in the dual-threaded ray-tracer, in seconds. Timing information was taken from a single trial and is rounded off to the third decimal place. Thread 1 indicates the bottom half of the screen, while thread 2 indicates the top half of the screen.*

	Sequential Time	Thread 1	Thread 2	Overhead
Scene 1	25.641	6.734	19.078	59.672
Scene 2	1.281	0.907	1.297	4.89
Scene 3	1.281	0.297	1.547	10.688
Scene 4	4.406	4.312	0.047	11.938
Scene 5	1.063	0.141	1.219	3.25

Table 3: *Timing information for two threads using load-balancing, in seconds. Overhead timing information is also provided, as is sequential processing time as a reference. Sequential execution times listed here are incongruous with those provided above due to a hardware failure on the original test machine.*

load-balancing. As seen in Table 4 experiments recorded timing results similar to unbalanced tests, albeit taking a little longer. After verifying that the load-

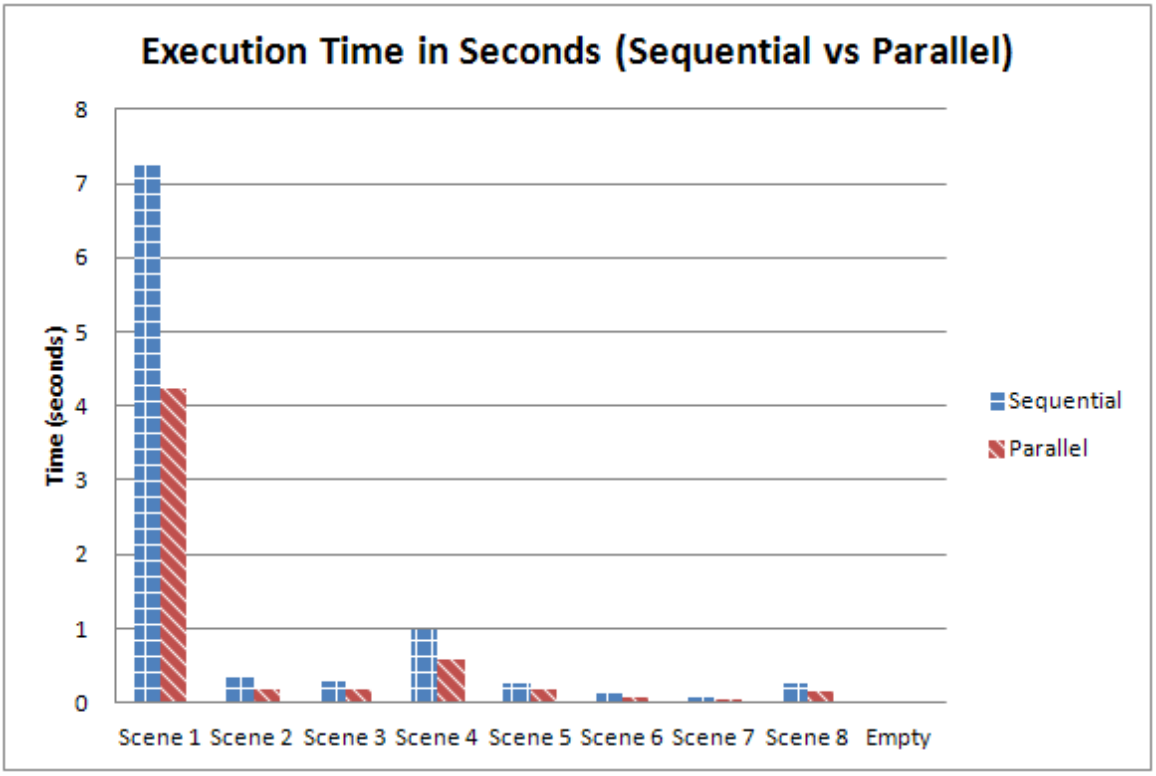

Figure 12: *Execution Time, Sequential vs. Parallel. Graphical representation of the results shown in Table 1.*

balancing algorithms were working correctly, we concluded that the added overhead was the cause of the comparative slowdown, though we don't have actual timing data as load-balancing occurs during ray-tracing in this algorithm, making it difficult to distinguish between the rendering stage and the load-balancing stage.

After running these tests on a dual-core machine, we repeated the test using the scene of 100 randomly-generated spheres on an eight-core machine, using a different number of cores for each test. Timing information is available in Table 5 and Figure 14, with speedup data shown in Figure 15. The results demonstrate some interesting behavior of our parallel application:

1. Our speedup maximizes at 4 cores. Using the unbalanced screen-space subdivision technique, we achieve a speedup of about 300

2. The sphere-based load-balancing algorithm is never faster than running Frosty on one core.

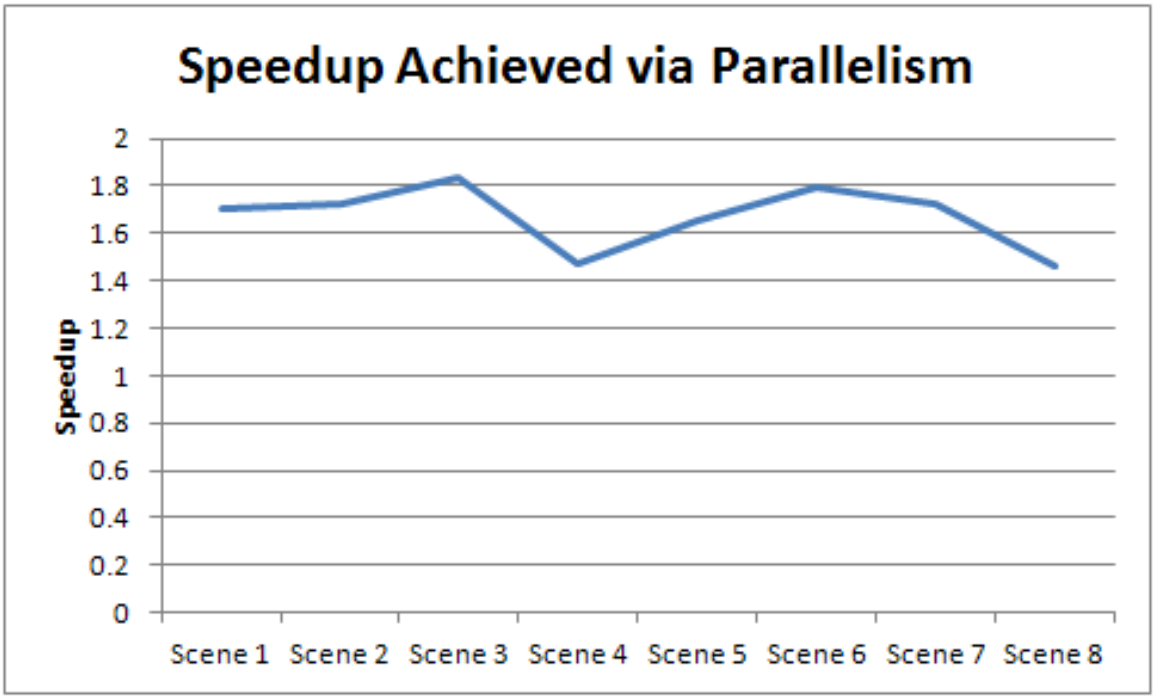

Figure 13: *Speedup, Sequential vs. Parallel. Graphical representation of the speedup achieved through the earliest screen-space sub-division parallelization method. Speedup is a percentage found by dividing the execution time of the slower implementation by that of the faster implementation (in this case, we divide sequential execution time by parallel execution time). The fact that the speedup achieved for each scene doesn't fluctuate wildly is a very good sign. When the speedup does decrease drastically, that indicates a scene in need of load-balancing.*

3. The subsample-based load-balancing algorithm is roughly on par with the screen-space method, until the speedup factor unexpectedly decreases with five or more cores.

	Thread 1	Thread 2
Scene 1	3.594	3.063
Scene 2	0.531	0.546
Scene 3	0.468	0.468
Scene 4	0.797	0.875
Scene 5	0.469	0.484
Scene 6	0.390	0.406
Scene 7	0.985	0.985
Scene 8	0.313	0.328

Table 4: *Timing information for two threads using subsample-based load-balancing, in seconds. Each thread takes roughly the same amount of time to render using this technique. The biggest time discrepancy is in Scene 1, the randomly-generated 100 sphere test scene. This is to be expected for larger scenes, where load-balancing takes more time and is less accurate.*

	1 core	2 cores	3 cores	4 cores	5 cores	6 cores	7 cores	8 cores
Horizontal	3.12	1.622	1.326	1.014	0.967	0.873	0.827	0.889
Vertical	3.12	1.794	1.201	1.03	0.905	0.874	0.827	0.827
Sphere	3.12	9.875	9.766	9.5	9.516	60+	60+	60+
Subsample	3.12	1.763	1.373	1.029	2.574	2.761	2.995	2.87

Table 5: *Timing results on an eight-core machine, in seconds. By running the same experiment with a limited number of cores on the same machine, we can tell exactly how effective our parallel application is by comparing, for example, execution time on four cores to execution time on eight cores.*

6 Conclusions

Our research accomplished several goals. The first goal was to gain experience in parallel processing with a focus on applications to computer graphics. We achieved

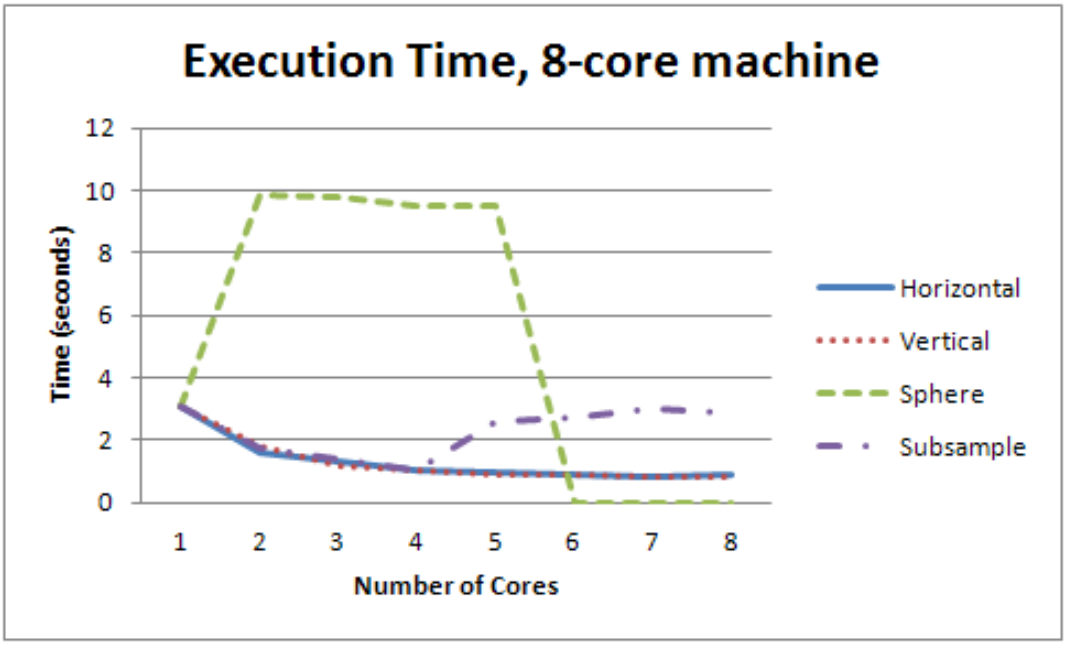

Figure 14: *Timing results on an eight-core machine. By running the same experiment with a limited number of cores on the same machine, we can tell exactly how effective our parallel application is by comparing, for example, execution time on four cores to execution time on eight cores. Results listed as 60+ in the table above are not included here.*

this goal by taking an existing ray tracer and modifying it to run in parallel, with several different techniques tested for achieving load-balancing.

We accomplished our second and third goals with our ray tracing solution. These goals were, respectively, to work with a ray tracer and increase performance to the point where a decent ray tracer could run smoothly on ordinary consumer hardware. Having experience with rasterizers such as OpenGL, we wanted to expand our computer graphics knowledge and experience. Having theoretical knowledge of but no working experience with ray tracing, we opted to design our own. Although we ultimately decided to improve upon an existing ray tracer, by studying this implementation we were able to understand the components of a ray tracer much better than simply reading about one in a textbook. Since we feel that, should the need arise, we would be able to develop our own ray tracer from scratch, this second goal has been achieved. The third and final goal, increasing

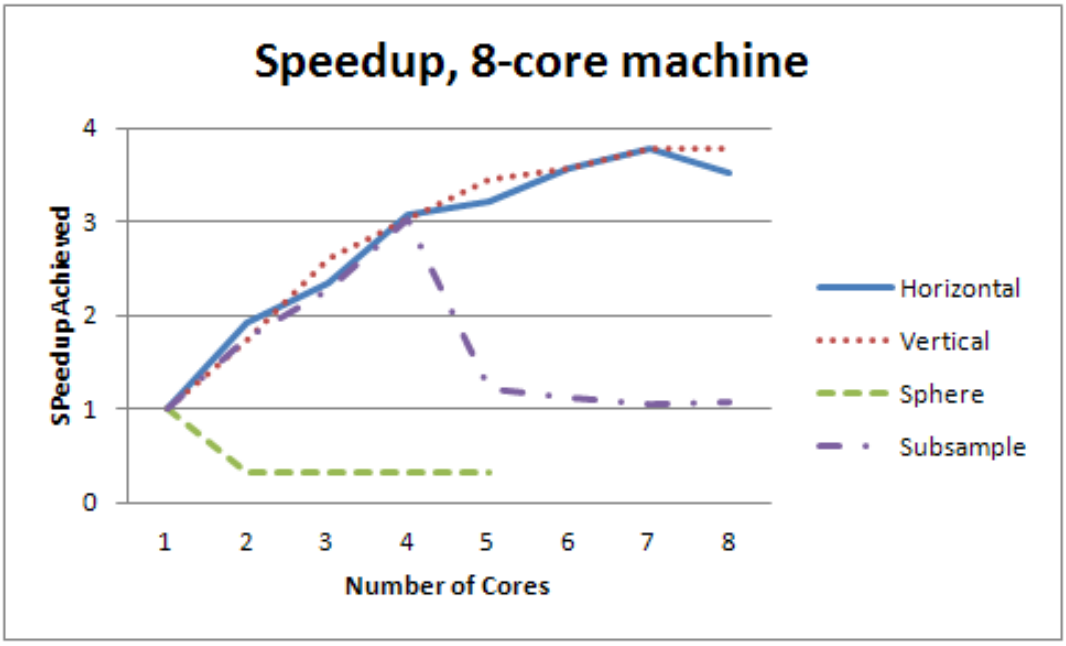

Figure 15: *Speedup on an eight-core machine. A side-by-side speedup comparison across all eight cores of the same machine allows us to gauge the impact of multiple cores on performance.*

performance to the level of real-time rendering on low-level consumer hardware, was achieved with mixed results. While this was admittedly a bit of a stretch goal, we were still able to reduce rendering time through the use of multi-core parallel processing.

However, as our results show, load-balancing was, in most cases, largely ineffective. The speedup achieved with the load-balancing techniques we used was often outclassed by the speedup achieved without any load-balancing at all. We want to stress that this does not mean that load-balancing has no impact on parallel ray-tracing applications. The only thing we can actually determine from our results is that these particular load-balancing algorithms are insufficient for our purposes. We also learned that as we add more cores, load-balancing may have a greater impact on speedup: if we can load-balance in a way that alleviates the problem of diminishing returns, we should be able to see increased speedup given any number of cores. Ideally, we'd like to achieve as close to a 1-to-1 ratio of

speedup to available processors as possible. Frosty has achieved roughly a .75-to-1 ratio with up to four available processors; after this the ratio skews towards .45-to-1. Improving this ratio would be an excellent starting point for future research.

7 Future Work

While we are pleased with the results of our research, there is certainly room for improvement. On the simplest level, we could add more functionality to Frosty. We left out a great deal of functionality from our ray-tracer: transparent objects, textured surfaces, and non-spherical shapes are some missing elements that immediately come to mind. We left these out intentionally, as the comparatively few extra computations needed to produce these effects were irrelevant to our research compared to the massive amount of computations needed to bounce thousands of rays around a scene. We feel it is worth mentioning that if we were interested in reducing time spent calculating ray-object intersections, then this might have been more important to us.

As part of our research, we limited ourselves to only speeding up the ray-tracing algorithm through parallelism. However, a great deal of speedup can be achieved by alternate methods, ranging from simplifying calculations to having the computer make educated guesses for some pixels. As mentioned above, reducing the number of ray-surface calculations required for a given scene provides a significant speedup. It makes sense, then, that finding new ways to reduce the number of or complexity of these calculations is a frequent subject of research.

The biggest potential area of improvement would be improved load-balancing techniques. As we mentioned above, the load-balancing techniques we tested with Frosty were insufficient for our purposes. To improve our load-balancing capabilities, we have two options. We could either improve our existing techniques, or perform further research into other load-balancing algorithms. One flaw in our subsample-based load-balancing algorithm is that it only takes into account the very first few rays emitting from the camera - subsequent rays generated by intersecting objects are not factored into the balancing. This means that should a handful of rays strike a sphere and nothing else, the load-balancing algorithm will adjust the thread boundaries around that one sphere. If those rays then proceed to strike several more spheres, each of which produces new rays which intersect

even more rays, then our scene is not balanced around what will actually account for much of the processing time.

The binary space partitioning algorithm we developed to generate load-balanced scene sub-divisions has some annoying limitations that may impact its performance. A weakness we've already mentioned is its inability to provide proper load-balancing on machines with a number of cores not equal to a power of two. While the impact of this may not seem immediately obvious, let us consider a machine with 3 cores. By our algorithm, the scene is first divided into two partitions, with each partition assigned to a separate core. One of these is then sub-divided into two partitions, so the work is spread out amongst the machine's three cores. But in this scenario, one core is doing half the work, while two others are doing one-fourth of the work each. Since the program has to wait around for all three threads to finish, and one core will take roughly twice as long as the other two, we aren't receiving any gains from the having a third core, and thus the extra processing power we could be utilizing instead goes to waste. We must also consider the fact that the partitioning algorithm only performs horizontal splits. This approach has some advantages and disadvantages. One of the benefits of this design is that in order to compare a vertical split as more or less efficient than a horizontal split, we'd need to calculate and balance the work required for partitions based on a vertical split in addition to doing this for the horizontal split. By ignoring the possibility of a vertical split, we save ourselves some vital CPU time. The downside to this of course is that the vertical split may be faster and easier to balance and could also produce a more equally-balanced load for each processor. Although we did not implement such an algorithm, our ray tracer is equipped to handle a partitioning algorithm that supports vertical sub-divisions.

Radiosity, a lighting algorithm somewhat similar to ray-tracing, could also benefit from the results of my research. In a sense, radiosity can be thought of as "backwards" ray-tracing: while a ray-tracer emits rays from the camera which then bounce around the scene until hitting a light source, a radiosity-based

approach emits rays from the light source(s) which then bounce around the scene until hitting the camera, just like how real-world light sources work. Radiosity is designed around the principles of thermodynamics. The algorithm simulates real-world lighting conditions by paying special attention to the light energy in the environment, keeping track of how much energy is reflected and absorbed by each surface in the room [3]. The goal of the original algorithm was to simulate the physical behavior of light, with special attention to modeling the diffuse reflection of light; the way light reflected by one surface affects the illumination of another surface [5]. Like raytracing, the bulk of the computations required by the radiosity algorithm come not from any specifically complicated calculations, but the sheer number of calculations and iterations required to properly light a scene. Based on my (admittedly limited) knowledge of radiosity, an attempt at parallelism based on my own screen-space sub-divisions could provide significant increases to the performance of this algorithm on low-end machines.

References

[1] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[2] Brian Corrie and Paul Mackerras. Parallel volume rendering and data coherence. In *Proceedings of the 1993 symposium on Parallel rendering*, PRS '93, pages 23–26, New York, NY, USA, 1993. ACM.

[3] Andries Feiner Steven K. Hughes John F. Foley, James D. van Dam. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Company, 1996.

[4] Sara Jackson and Glazer, Steve. Parallel ray tracing.

[5] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 213–222, New York, NY, USA, 1984. ACM.

[6] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Comput. Graph. Appl.*, 9(6):12–26, November 1989.

[7] Eric Haines. An introduction to ray tracing. chapter Essential ray tracing algorithms, pages 33–77. Academic Press Ltd., London, UK, UK, 1989.

[8] Henrik. Ray trace diagram. Wikipedia, April 2008.

[9] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[10] Thierry Priol and Kadi Bouatouch. Static load balancing for a parallel ray tracing on a mimd hypercube. *The Visual Computer*, 5(1&2):109–119, 1989.

[11] Erik Reinhard and Frederik W. Jansen. Rendering large scenes using parallel ray tracing. *Parallel Comput.*, 23(7):873–885, July 1997.

[12] The Codermind Team. A raytracer in c++.

[13] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.