# Real Time Ray Tracing in a Space Limited Environment

Daniel Q. Oliphant[+], G. Elisabeta Marai[+]

University of Pittsburgh, [+]Department of Computer Science, {dqo1, marai}@cs.pitt.edu

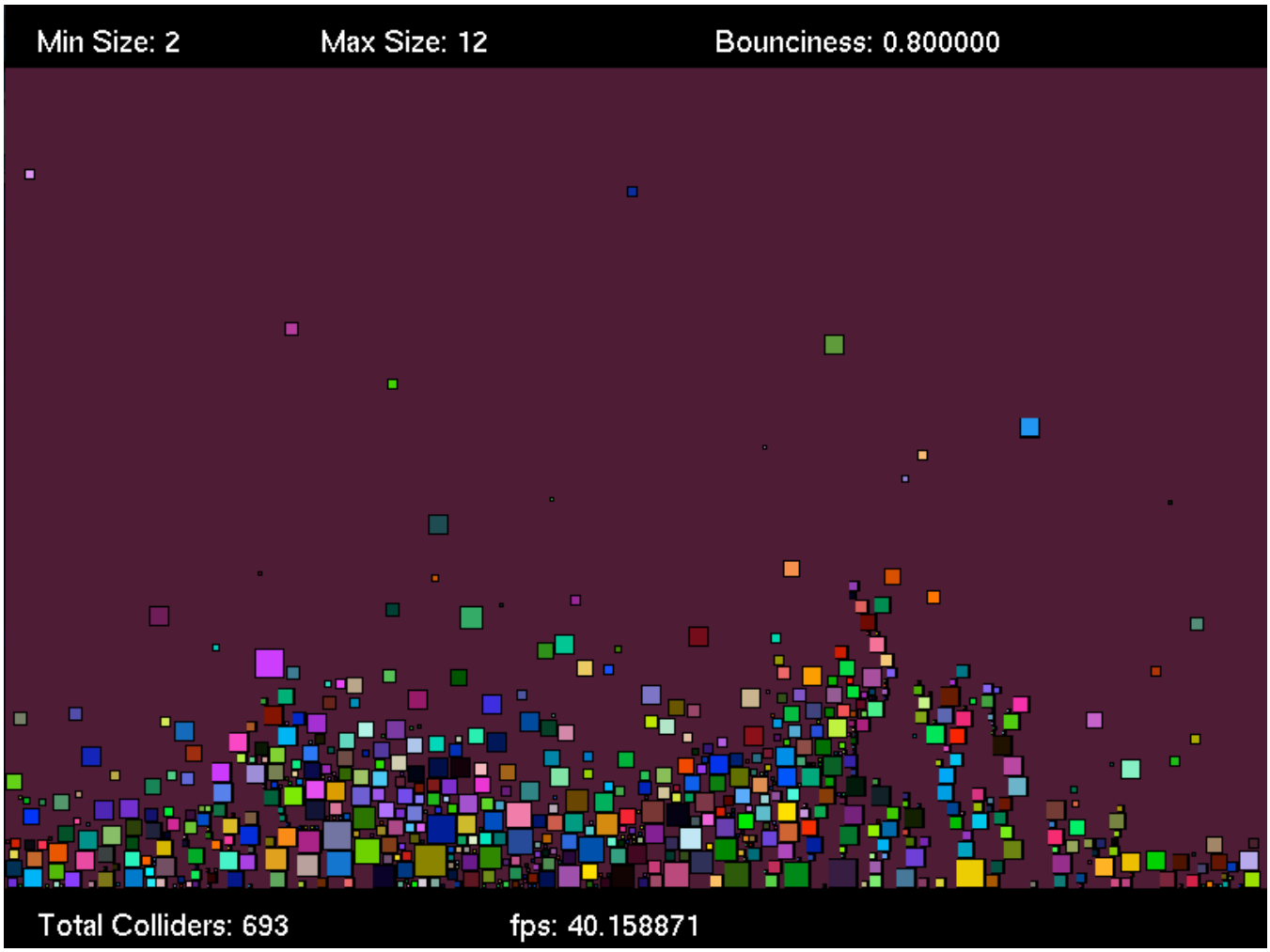


Figure 1. Proof of concept of a real-time ray-tracing collision model. A 2-dimensional space of 800 by 600 pixels is used to represent each pixel as being occupied or free. The space is initialized with a particle system; the user manipulates the particle system through the mouse; forces such as gravity can also be controlled. (For a videoclip please see: http://www.youtube.com/watch?v=HhN1WfbWCNU&feature=player_embedded).

## 1 Introduction

Ray tracing as a rendering technique has many useful applications in both simulation and entertainment. A real-time ray-tracing engine is even more valuable as it allows for user interactivity, greatly increasing the number of potential uses for such a renderer. Most ray tracing techniques are employed either to enhance realism or for purely aesthetic purposes. These engines are generally far more costly for rendering a scene than traditional rasterization techniques. In this work we explore the potential strengths of ray tracing in terms of efficiency greater than that of rasterization.

A great limitation in ray tracing is the number of objects or the geometric detail of the objects present in a scene. While a simple sphere or cube may be rendered with stunning detail and great accuracy at high frame rates, add more objects to the scene or make those objects more complex and rendering speed slows to a crawl. This is very noticeable in even the most advanced video games where ray-tracing techniques are used. While one ray traced rendition of a tree may look great and render quickly, add more trees to the scene or even an entire forest and the experience quickly degrades. This is mainly due to the common methods of detecting ray-plane intersections. Often, rays must be mathematically compared to each and every object in the scene to determine where the nearest intersection occurs (or if an intersection occurs at all). Thus, doubling the number of objects in a scene, or doubling the number of polygons in a single object, effectively doubles the number of calculations that must be

performed on each ray. To put this problem into perspective, consider a screen of 800 by 600 resolution. This relatively small screen contains 480,000 pixels, and so 480,000 rays must be traced into a scene in order to generate a single frame. For each and every geometric object in the scene (e.g. for each polygon), 480,000 ray-plane intersection tests must be performed. Add a few models of a few thousand polygons each and it quickly becomes apparent why the branches on the trees in a top-selling video game must necessarily be so flat. It is apparent that the greatest weakness in ray tracing is its substantial cost, and thus a more efficient method of ray tracing would be of great worth. In this paper we propose a space-aware approach to increase the speed at which rays are traced, as well as implicit solutions to common ray-tracing issues.

## 2 METHODS

This method of performing ray-plane intersection tests for everything in the scene can seem very counter-intuitive. In the real world, if a ball is moving through a room full of obstacles, it is not necessary to consider all the obstacles in the room to know whether the ball will continue moving or collide with something in the next millisecond. Rather, only one point in space matters in determining whether there is a collision with that ball: the area that the ball is trying to occupy. If there is already an object there, the ball will collide. If the space is empty, then the ball moves to fill it. In such an environment an increased number of objects would have no effect on the complexity of the collision algorithm for the ball. Whether there be hundreds or hundreds of millions of objects in the room with the ball, only that one location must be checked to determine whether the ball collides or keeps moving. The same real-world idea may be applied to the collision detection necessary for ray tracing. When three-dimensional space was first being represented on a computer, memory was scarce and so great levels of abstractions were necessary to hold 3d objects in memory. At the time, the idea of representing every single unit of space in a 3d scene of even modest size was completely infeasible. Today, the situation has changed. Memory has become a much more abundant resource and with this increase in availability new algorithms that utilize vast amounts of memory have become feasible. Instead of representing a box or a tree as a collection of vertices it is possible to allocate a three-dimensional array of memory and occupy each location in that array that the tree or box would occupy in real life. Then, when it comes time to test for collision upon moving into a specific space in this array, only the one index need to considered. Applying this sort of collision or intersection test along a ray is slightly more complicated than with the ball example primarily because an entire set of points along the ray must be considered and the collision closest to the origin detected.

The first step to create such an engine from scratch was to build a simple proof of concept of the collision model (Figure 1). In this two- dimensional environment every single pixel is represented in memory as being occupied or free. This allows for highly efficient, simple, and reasonably accurate collision-detection.

Upon creating this proof of concept it became apparent how brash the original idea of representing every unit of 3d space in memory might be. While memory has become far more widely available than it once was, the memory requirements of this algorithm are orders of magnitude greater than the traditional approach. A 2 dimensional space like the one used in this example (800 by 600 pixels) requires 480,000 units of memory. Add just one pixel-sized unit of depth to the scene and already the memory requirements double. A scene that is as deep as this one is wide (800 by 600 by 800 pixels) would require 384,000,000 units of memory! It seems that the 4 – 16 Gigabytes of memory available on modern machines may not yet be able to cut it.

This proof of concept also brought to mind the importance of determining what information needs to be stored at any given location in space. For the proof of concept, only the state of being occupied or not was stored. A single bit for each pixel on screen represented this. Storing any more information than that would be much more costly, but many attributes such as color, surface normals, and opacity may need to become pertinent in a ray tracing environment. At this time methods of reusing similar data with pointers and aggregating large amounts of uniform data were considered for practicality in the implementation of the actual ray-tracing engine.

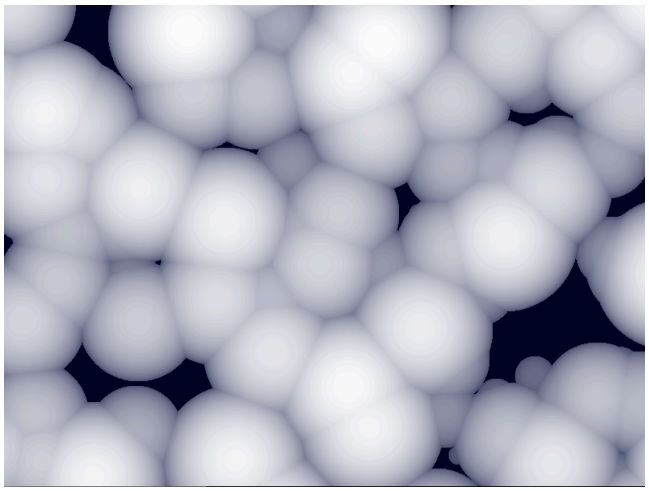

Figure 2. Orthographic Ray-Trace.

Having seen promising results with the proof of concept, the next step became to implement a very rudimentary version of the originally designed ray tracer. This first ray tracer was a single pixel deep, just like the proof of concept. At each frame, a ray was sent from each pixel on the screen straight into the scene and traveled the distance of one pixel to check if that space was occupied. If the space was occupied, the pixel from which the ray originated was colored white. If it was empty, the corresponding pixel was colored black. While this may not seem like the ray-tracing engine seen in today's hottest video games, in the most fundamental sense it was doing ray tracing.

The next step was obviously to add more than one pixel of depth to the scene. This is where the limits of a 32-bit operating system with 4gb of ram came into play. With just a few levels of depth, the memory was depleted, and so the scene was hardly even 3d. At this time it became necessary to change the original plan of representing each unit of space in memory whether or not it was occupied. By implementing an oct-tree structure, only the points that were occupied were actually represented in memory while the vast expanses of empty space were consolidated where high-level tree nodes had no children.

| Number of Points | Seconds per Frame |
|---|---|
| 1 | 0.105575 |
| 10 | 0.310432 |
| 100 | 0.828678 |
| 1,000 | 1.980592 |
| 10,000 | 4.469226 |
| 100,000 | 7.086494 |
| 1,000,000 | 3.390397 |
| 10,000,000 | 1.172221 |
| 20,000,000 | 0.942457 |

Table I. Perspective rendering.

While the oct-tree did solve the problem of memory constraints, it added a level of complexity to checking a single point in the scene. Instead of simply indexing the point in a 3-dimensional array like had originally been planned, it was necessary to traverse the oct-tree. As a trade off, the oct-tree could eliminate costly checks into vast areas of emptiness if traversed in the right order. For simplicity, an orthographic, front to back traversal of the oct-tree was executed for each ray. So while this version of the ray-tracing engine included depth, it was rendered orthographically.

In Figure 2 the results of this orthographic ray-trace can be seen. In this scene, spherical objects were added to the world, world unit by world unit. The distance away from the camera determines the brightness of the coloring, so depth can be seen even without lighting or perspective. The next and final step for the purposes of this project was to implement perspective in the ray-tracing scheme. While this was technically a frustrating algorithm to get right, it eventually boiled down to intelligently traversing the tree at the appropriate angle to the camera instead of straight back like before.

## 3 RESULTS

One of the strengths inherent to this method of ray tracing is the automatic pruning that occurs in a scene when a ray intersects with an object without have to consider the objects behind it. This strength may lend itself greatly to situations where very large numbers of objects need to be painted to the screen such as in the case of Cosmological Simulation. When millions or even billions of stars need to be rendered to the screen, it is wasteful to try to paint all those stars to a viewing area of only 480,000 pixels. Ray-tracing locks the amount of work to be done to the number of rays to be traced, thus adding more objects does not substantially decrease performance as it does in rasterization.

Performance test results based on the engine running on a single processor are shown in Table I and Table II. While the performance of rasterization techniques begins to quickly degrade around one million points, this is where the ray-tracing engine gets strongest. This is just one possible use for this different approach to rendering.

## 4 DISCUSSION AND CONCLUSION

While the results show that this engine needs further optimizations and improvements before it can be useful for interactive rendering, it can be said that these early results are promising and indicate that such an approach to rendering could solve certain classes of problems. The great amount of memory on modern systems may be utilized to take much of the load off the processor. Since memory has become relatively cheap, this approach seems logical for most efficient use of system resources.

In order to make this algorithm useful, it must be optimized to decrease the render time of any given frame. The most obvious approach to optimization is to parallelize the algorithm. Since each pixel on the screen and therefore each ray can be evaluated separately, it is possible to make this algorithm massively parallel. Such a solution would make efficient use of as many processors as there are pixels on a screen.

## REFERENCES


[1] *Baboud, Lionel, and Xavier D'ecoret. "Realistic Water Volumes in Real---Time." Eurographics Workshop on Natural Phenomena (2006).*

[2] *Fraedrich, Roland, Jens Schneider, and Rudiger Westermann. "Exploring the Millennium Run - Scalable Rendering of Large-Scale Cosmological Datasets." IEEE TVCG Oct 2009.*


| Number of Points | Seconds per Frame |
|---|---|
| 1 | 0.007379 |
| 10 | 0.015365 |
| 100 | 0.057024 |
| 1,000 | 0.153735 |
| 10,000 | 0.373861 |
| 100,000 | 0.866302 |
| 1,000,000 | 1.49762 |
| 10,000,000 | 0.883987 |
| 20,000,000 | 0.669997 |

Table II. Orthographic Rendering.