Teapot Ray Tracing with BSP Tree

Qishen Cai
CSCI 580
qishenca@usc.edu

Ann Yan
CSCI 580
annnanya@usc.edu

Wanyu Zhang
CSCI 580
wanyuzha@usc.edu

Nour Hamade
CSCI 580
nhamade@usc.edu

Abstract

Raytracing is a straightforward but expensive way to calculate more complex lighting and shading on an object. Throughout this paper, we discuss how we modified HW5 to send rays into the scene and then optimized it by building a BSP Tree and traversing the space along the ray direction.

Index Terms: Ray tracing – Reflection – Refraction – Shadows – Triangles – BSP Tree

1 Introduction

Raytracing is a straightforward way to introduce more complex lighting and material calculations to rendering. By casting a ray from each pixel into the scene [1], we are able to check for any intersections between the ray and the triangles passed into the renderer.

For our final project, we chose to modify our homework 5 code to allow for shadows, reflection, and refraction. To do this, we implemented ray tracing with triangles. By broadcasting rays into the scene, we can check for intersections and material types [6] to create increasingly complex images. However, brute-force raytracing greatly increases the processing time required to produce these images. To combat this issue, we decided to utilize BSP trees.

2 Brute-Force Ray Tracing

2.1 Overview

Our code, branched off of HW 5, overhauls the rasterizer to introduce ray tracing. We changed the Application5.cpp code to call the new ray tracing function, which is the basis of our ray tracer. Here we are able to branch off and calculate the primary ray for each pixel before passing it into another function to calculate the intersection color.

2.2 Set Up

In addition, we had to create a triangle buffer (an array) and triangle object, modify the pixel object, and create a GzRay structure to help implement ray tracing within the existing code.

2.2.1 GzTri and Triangle Buffer

With the addition of ray tracing, we needed to be able to repeatedly iterate through the triangles and store specific information about each one. Therefore, we decided to introduce an additional triangle buffer and GzTri struct to store all this information.

2.2.2 Modified Pixel

Ray tracing also forced us to store more information on each pixel to calculate everything correctly. We had to edit the GzPixel struct to store a triangle (GzTri), T-value (float), and hit point (GzCoord). The use of the T-value was for distance checking, replacing any use of Z-buffer. The hit point is needed for ray calculations used to determine the color of a pixel.

2.2.3 GzRay

Finally, we needed a way to store the data of the ray. GzRay is a simple struct we implemented to store the origin and direction of the ray. Both of these variables are of type GzCoord, as found in the homework.

2.3 Methods

When starting off with the ray tracer, we looked into "Backward Ray Tracing" by Arvo [1], "Reflections and Refractions in Ray Tracing" by Greve [3], "Ray-triangle Intersection" by Brian Curless [6] and the example source code provided in "Introduction to Ray Tracing: a Simple Method for Creating 3D Images" by Scratchapixel [2]. This allowed us to break down the exact steps we needed to take to create a brute-force ray tracer that branched off HW 5.

2.3.1 Backwards Tracing and Primary Ray Calculation

The first step we took was to create a primary ray, which is shot from the camera into each pixel. Here we read through "Backward Ray Tracing" by James Arvo [1] to better understand what we were trying to achieve. We also took a look at the Scratchapixel chapter called "Ray Tracing Algorithm in a Nutshell" [2]. The equation of a ray as used in our implementation is shown below.

$$Ray = Origin + t * direction$$

Equation 1: Ray Equation [2].

For the primary ray, we set the origin as $\{0, 0, 0\}$. Then we transformed each screen space pixel into image space and treated it as an endpoint when computing the (normalized) direction of the ray. The result was a primary ray in image space that could be used to calculate triangle intersections.

2.3.2 Triangle Ray Intersections

With our primary ray calculated and in the intended space, we now had to check it against every triangle for an intersection. The brute-force ray tracing implementation was based on "Ray-triangle Intersection" by Brian Curless [6] and Scratchapixel's "Implementing the Ray Tracing Algorithm" [2]. The method used for the BSP Tree approach will be described in a future section.

Our brute-force approach takes the primary ray and, for each triangle in the triangle buffer, calculates the plane the triangle spans. If the ray intersects this plane, we then calculate whether the intersection point is found in the triangle. If it is inside the triangle, and it is the front-most intersection based on the T-value distance check, we store the hit point. Once all intersections have been checked and the frontest-one stored, we calculate the color of the point.

2.3.3 T-Value Distance Check

Instead of using the Z-buffer check found in HW5, we replaced it with a T-value check based on the ray intersection method described prior. T-values are used to calculate where along the ray a point is located, so it serves a similar purpose to Z-values. The smaller the T-value, the closer to the origin the intersection point is located. Therefore, if the T-value found for the intersection is smaller than the one currently stored, it is in front of the previous point, and therefore is saved as the front-most intersection value.

2.3.4 Shadows

Once we were able to properly render the teapot, we implemented shadows. Reading over the algorithm provided in the source code found in the “Casting Curved Shadows on Curved Surfaces” by Lance Williams [12] and the Scratchapixel chapter “Introduction to Shading” [2], we determined that we needed to create a shadow ray to determine the visibility of the pixel based on each light source.

The pre-existing shading equation implementation in HW5 was modified to allow for shadow computations. While iterating over the lights, we performed a shadow check with a shadow ray. The hit point created by the primary ray and the front-most triangle was utilized as the shadow ray’s origin. The normalized direction of the light was set as the shadow ray’s direction, since homework 5 uses directional lights. We then checked to see if any triangle intersected with the shadow ray computed, disregarding self-intersections. If there was an intersection, the light was omitted from the shading equation summation for that point, resulting in a shadow effect**.**

2.3.5 Reflection

With shadows complete, we then implemented reflection and refraction. To start, we skimmed Bram de Greve’s work [3] on reflections and refractions to get a general understanding of the algorithm.

First, we edited the inputted file to create a mirror-behaving triangle. This allowed us to test reflection without messing with the teapot. However, it forced us to check for the triangle’s material type when calculating shading to determine whether to even calculate reflection. This required editing the triangle struct (GzTri) and previous shading calculations to allow for material-type storage and evaluation.

Next, we introduced the Fresnel Equation [3] to calculate the reflectance of the material and used Schlick’s Approximation to calculate the angle [3].

$$R_0 = (\eta - 1)^2 / (\eta + 1)^2, \text{ for } \theta = 0$$

Equation 2: Fresnel Equation [5].

$$R_\theta = R_0 + (1 - \cos \theta)^5 (1 - R_0), \text{ for } 0 < \theta < 90$$

Equation 3: Schlick’s Approximation [5].

Using the calculated values, we calculated the color of the hit point by recursively calling the ray tracing function up until a certain depth (3) was hit (to prevent infinite recursion) to simulate real-life reflection.

2.3.6 Refraction

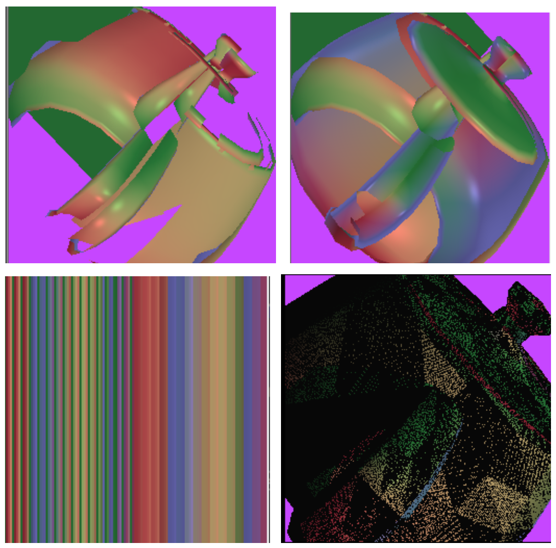
After reflection, we moved on to complete refraction as it is handled very similarly in the code. First, the triangle’s material type was checked to ensure it is transparent. Then we computed the refraction ray using Snell’s Law [3].

$$sin^2\theta_t = \left(\frac{\eta_1}{\eta_2}\right)^2 sin^2\theta_i = \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - cos^2\theta_i)$$

Equation 4: Snell’s Law [3].

With this completed, we recursively called the ray tracing function using the hit point, the current triangle, and the refraction ray until the maximum recursion depth (3) was met. We then added the resulting color to the refraction color.

2.4 Unexpected Brute-Force Outputs



One of our major problems was converting components into the same space. Hours were spent determining what space each element was in and what space all the calculations needed to be performed within. Having components in inconsistent spaces quickly led to unexpected outputs, as shown above.

In addition, we had problems with shadow outputs. The first time we rendered any kind of shadow, we found unexpected stippling across the entire teapot. After reviewing our papers and Scratchapixel [2] once more, we realized that the shadow ray was detecting the hit point’s triangle as an intersection, causing lights to be incorrectly omitted. This issue was resolved by checking for and disregarding self-intersections when determining the fronstest triangle.

2.5 Final Results and Example Outputs

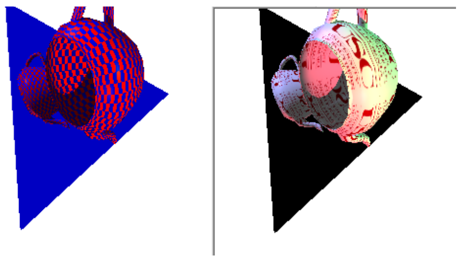


Figure 1: Reflection demonstration with textures.

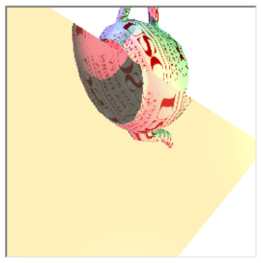


Figure 2: Refraction demonstration with textures.

With these changes, we were able to render teapots containing shadows, reflection, and refraction. This creates a more realistic teapot, and our shadows are able to be cast onto the textured version as well. However, the results take a considerable amount of time to achieve with simple brute-force raytracing, and so we sought to hasten the process by changing the way ray casts are processed and stored. This is why we implemented BSP trees.

3 Construct BSP Tree

3.1 Overview

A Binary Space Partitioning (BSP) Tree provides an easy way to divide and manage multi-dimensional spaces using a tree structure. In BSP Tree, a tree node represents a specific space, and its front child node and back child node represent two subspaces respectively dividing by a hyperplane passing through their parent spaces [1]. Each node includes one or more polygons and could have duplicate polygons with other nodes.

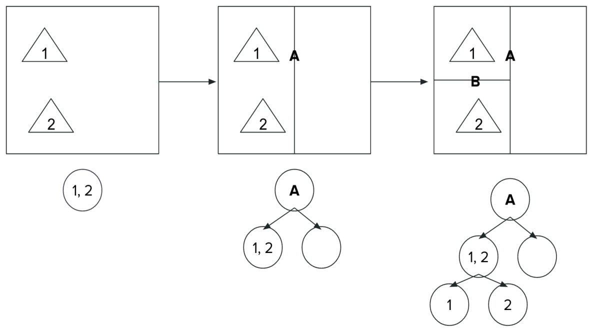


Figure 3: BSP Tree in two dimensions.

3.2 Set Up

3.2.1 BSP_Tree

To represent a tree node, we created a single struct containing two pointers to its front and back child nodes and a list of indices of all triangles inside the node space. As discussed below, the space is divided by the axis-aligned plane in order, so we added one integer variable inside the struct indicating which plane it uses. Also, the data structure consists of three pairs of float values describing the size of the bounding box, each pair of them representing the minimal and maximum values of the space on X-axis, Y-axis or Z-axis. The root node has the largest bounding box which contains all the triangles.

The construction method will be called recursively and has to be stopped at a certain time. Therefore, we use two static metrics to control the recursion of creating a BSP tree: MAX_DEPTH and MIN_SIZE. If the tree depth is larger than MAX_DEPTH or the triangles node contains is less than MIN_SIZE, we will stop dividing the space and make it a leaf node.

3.3 Methods

3.3.1 Axis Aligned Partition Planes

To start creating a BSP Tree, we had to determine the partition planes. It was important to pick planes that lead to a balanced tree where each leaf contains about the same number of polygons [8]. In addition, the hyperplane could align with the space axis or be any arbitrary plane. It is common to choose a polygon plane as a non-axis aligned plane, however, it might easily cause the unbalanced distribution of a binary tree. Additionally, a lot of fast algorithms in geometry are restricted to axis-aligned objects.

In our implementation, each node will store its own bounding box (also called spaces) in their struct described by three pairs of variables: min-x, max-x, min-y, max-y, min-z, max-z, and provide convenience when splitting the space of the node into two subspaces of front node and back node. The bounding box of the root node contains exactly every triangle in the scene and makes itself the biggest bounding box in the tree.

Starting from the root node, we choose three axis aligned planes (xy-plane, yz-plane, xz-plane) as the hyperplanes. Each of these planes divided the bounding box of the current space down the middle into two small bounding boxes. For instance, if we choose xy-plane as a partition plane, we are going to insert the plane into the middle of the current bounding box where the z-value of the plane equals the average value of min-z and max-z of the bounding box. This creates two smaller bounding boxes that are assigned to its front and back child node. We decided to split the bounding box into two equal halves because of the shape of the teapot. Cutting the bounding boxes in half is the simplest way to get a fairly balanced BSP tree. After that, we iterated each triangle in the list and used AABB Collision Detection to determine whether the triangle should be added to the front or back child of the current node.

3.3.2 AABB Collision Detection

At first, we developed the algorithm to check if the vertices of the triangle are inside the bounding box. However, the output showed some holes in the teapot. These were caused by a failure to detect larger triangles, as the vertices would fall outside of the node's bounding box. To solve this problem, we introduced Axis Aligned Bounding Box (AABB) Collision Detection to our code [11].

AABB Collision Detection creates a bounding box containing a single triangle and compares that to the bounding box stored in the tree's node. Then we compare whether the triangle's bounding box intersects with the current node's bounding box. To check for intersections, we compare the edges to see if they overlap [13]. In 2D space, a collision occurs when both the horizontal and vertical edges overlap, and in 3D space, a collision occurs when the edges of all three axes (x-axis, y-axis and z-axis) overlap.

4 Ray Tracing With BSP Tree

4.1 Overview

The BSP Tree method has been used in many applications such as hidden surface removal. The key principle is to traverse the BSP Tree in front to back order or vice versa. BSP Trees are easy to apply to ray tracing because rays only intersect with exact points on a polygon, which means we do not have to split the polygon intersecting with the hyperplane when checking for overlapping polygons.

4.2 Methods

4.2.1 Voxel Walking Algorithm

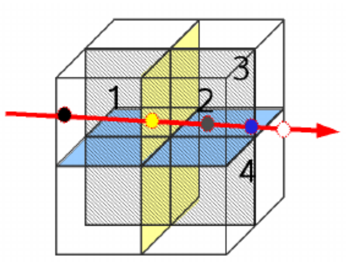


Figure 4: Ray traversing through octree, 2*2*2 voxels [3].

To traverse the BSP Tree, we utilized the Voxel Walking Algorithm. The Voxel Walking Algorithm is used to find all voxels on the ray path [9]. A voxel is a leaf node of our BSP Tree, and two adjacent voxels must be connected by a face, an edge, or a corner. The final goal of the Voxel Walking Algorithm is to find all the voxels on the ray path and compute the nearest intersection point. We used this algorithm to replace our brute force method of iterating through an array to find triangle intersections.

Following this algorithm, our code first computes two important values: min and max, which indicate the closer distance and the further distance from the bounding box of the root node to the ray origin along the ray [9]. These values are used to determine which half of the space we use and modify the min and max values. We then recursively call the algorithm until a leaf node is reached.

4.2.2 Traverse BSP Tree

When searching the BSP Tree, we start from the root node and traverse recursively with a Voxel Walking function, which takes a node, ray, min value, and max value as inputs.

If the node is a leaf, it means we have found a voxel on the path, and we will try to iterate all the triangles the voxel holds and return back the closest intersection or null.

If the node is not a leaf node, we compute the distance from the ray origin to the hyperplane that node holds, then compare this distance with min and max to decide which half space (near/far) the ray passes through. According to the algorithm, there are three options: 1. Pass through the near half only; 2. Pass through the far half only; 3. Pass through the near half first and then the far half. We will call recursive functions respectively and make sure the min and max have been modified to the correct range.

As discussed before, our code uses the axis-aligned plane as the partition plane, and we take the subspace below the partition plane as near, and the space above as far. The definition of front and back half of space is irrelevant with ray origin. At each step we have to recompute the near and far space according to the origin of the ray.

4.3 Final Results and Example Outputs

In comparison with our brute-force ray tracer, the implementation of BSP trees significantly improves the performance of our ray tracer. We included the varying times, depending on the max depth of the tree and the min size of the node. In comparison, the brute force method takes ~2450 ms. Even in the worst case of a max depth of 5 and min node size of 10, this is ~5.4 times faster. With a max depth of 15 and min node size of 5, the increase is ~55 times faster. This allows us to introduce increasingly complex objects into our scene and have them rendered within a reasonable time frame.

MIN_SIZE = 10

MAX_DEPTH	TIME
5	452 ms
8	214 ms
10	130 ms
12	72 ms
15	57 ms
20	62 ms

MAX_DEPTH = 15

MIN_SIZE	TIME
3	45 ms
5	45 ms
10	57 ms
15	64 ms
20	98 ms
30	120 ms

5 Conclusion

Throughout this project, we learned a lot about ray tracing and BSP trees. While ray tracing is expensive to calculate, the improvement in the images it produces is well worth it. Luckily, the BSP tree sped up the process a significant amount, allowing us to render high-quality, realistic images within a reasonable amount of time.

If we were to do this project again, there are a number of things we would change. Firstly, we would plan the entire project out before we started digging into the code. We ran into quite a few problems because we would code and then realize we misunderstood how something worked, forcing us to undo all the work we previously accomplished.

Overall, we learned a lot about the way ray tracing is done and how it can be optimized. If we were to continue work on this project, we would continue to build onto our ray tracer to introduce more complex lighting calculations and find some dynamic metrics to create an even faster BSP tree.

Acknowledgements

The authors wish to thank Ulrich Neumann for an enjoyable class that taught us a variety of different graphics related topics. Also, the authors would like to thank Yiyang Li and Yiqi Zhong for helping us understand what we were doing wrong.

References

[1] J. Arvo, “Backward Ray Tracing,” [Online Document] August 1986. Available: https://courses.cs.washington.edu/courses/cse457/15au/projects/trace/extra/Backward.pdf [Accessed November 20, 2022].

[2] “Introduction to Ray Tracing: a Simple Method for Creating 3D Images,” *Scratchapixel*, https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing. [Accessed: November 25, 2022].

[3] B. De Greve, “Reflections and Refractions in Ray Tracing,” [Online Document] November 2006. Available: https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf [Accessed: November 27, 2022].

[4] “Introduction to Shading”, *Scratchapixel*, https://www.scratchapixel.com/lessons/3d-basicrendering/introduction-to-shading/reflection-refraction-fresnel [Accessed November 25, 2022].

[5] H.W. Shen, “Reflection and Refraction,” https://web.cse.ohio-state.edu/~shen.94/681/Site/Slides_files/reflection_refraction.pdf [Accessed November 25, 2022].

[6] B. Curless, “Ray-triangle Intersection,” [Online Document] April 2004. Available: https://courses.cs.washington.edu/courses/cse557/09au/lectures/extras/triangle_intersection.pdf [Accessed: November 20, 2022].

[7] T. Ize, I. Wald and S. G. Parker, "Ray tracing with the BSP tree," *2008 IEEE Symposium on Interactive Ray Tracing*, 2008, pp. 159-166, doi: 10.1109/RT.2008.4634637.

[8] B. Wade, "BSP Tree Frequently Asked Questions," 2001. Available: https://people.eecs.berkeley.edu/~jrs/274s19/bsptreefaq.html#8.txt [Accessed: November 20, 2022].

[9] J. Arvo, "Linear-time voxel walking for octrees." *Ray Tracing News 1.2*, 1988. Available: https://graphics.stanford.edu/pub/Graphics/RTNews/html/rtnews2d.html#art5 [Accessed: November 23, 2022].

[10] A. Knoll, et al, "Interactive isosurface ray tracing of large octree volumes." 2006 IEEE Symposium on Interactive Ray Tracing. IEEE, 2006.

[11] PY. Cai and S. Lin Goel, “Simulations, Serious Games and Their Applications,” [Online Document] 2014. Available: https://vdoc.pub/documents/simulations-serious-games-and-their-applications-1paikss9dtf0 [Accessed: November 26, 2022].

[12] L. Williams, “Casting Curved Shadows on Curved Surfaces,” [Online Document] 1978. Available: https://dl.acm.org/doi/10.1145/800248.807402 [Accessed November 18, 2022].

[13] “Collision detection,” *LearnOpenGL*. [Online]. Available: https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection. [Accessed: 25-Nov-2022].