

ACCELERATING RAY CASTING USING CULLING TECHNIQUES
TO OPTIMIZE K-D TREES

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in Electrical Engineering

by
Anh Viet Nguyen
August 2012

© 2012

Anh Viet Nguyen

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Accelerating Ray Casting using Culling Techniques
to Optimize K-D Trees

AUTHOR: Anh Viet Nguyen

DATE SUBMITTED: August 2012

COMMITTEE CHAIR: John Oliver, Ph.D.

COMMITTEE MEMBER: Dennis Derickson, Ph.D.

COMMITTEE MEMBER: Chris Lupo, Ph.D.

ABSTRACT

Accelerating Ray Casting using Culling Techniques to Optimize K-D Trees

Anh Viet Nguyen

Ray tracing is a graphical technique that provides realistic simulation of light sources and complex lighting effects within three-dimensional scenes, but it is a time-consuming process that requires a tremendous amount of compute power. In order to reduce the number of calculations required to render an image, many different algorithms and techniques have been developed. One such development is the use of tree-like data structures to partition space for quick traversal when finding intersection points between rays and primitives. Even with this technique, ray-primitive intersection for large datasets is still the bottleneck for ray tracing.

This thesis proposes the use of a specific spatial data structure, the K-D tree, for faster ray casting of primary rays and enables a ray-triangle culling technique that compliments view frustum and backface culling. The proposed method traverses the entire tree structure to mark nodes to be inactive if it is outside of the view frustum and skipped if the triangle is a backface. In addition, a ray frustum is calculated to test the spatial coherency of the primary ray. The combination of these optimizations reduces the average number of intersection tests per ray from 98% to 99%, depending on the data size.

Keywords: ray tracing, k-d trees, view frustum, backface, ray-triangle, culling

TABLE OF CONTENTS

TABLE OF TABLES	vi
TABLE OF FIGURES	vii
TABLE OF CODES	viii
1. Introduction.....	1
2. Basics of Ray Tracing.....	4
2.1. View Rays	5
2.2. Ray-Triangle Intersection.....	8
2.3. Light Rays	10
2.4. Lighting	11
2.4.1. Ambient Light.....	11
2.4.2. Diffuse Light.....	12
2.4.3. Specular Light.....	13
2.4.4. Pixel Color	14
3. K-D Tree	15
3.1. Building the K-D Tree.....	15
3.2. Traversing the K-D Tree	17
3.3. Optimizing the K-D Tree	19
3.3.1. View Frustum Culling.....	19
3.3.2. Backface Culling.....	22
3.3.3. Ray-Triangle Culling	24
3.4. Updating the Traversal Algorithm	28
4. Results and Analysis	29
4.1. Bunny	30
4.2. Teapot.....	32
5. Conclusion	35
6. Appendices.....	36
7. References.....	72

TABLE OF TABLES

Table 1: Bunny Test Results: View Frustum Culling (VFC), Backface Culling (BFC), Ray-Triangle Culling (RTC).....	30
Table 2: Teapot Test Results: View Frustum Culling (VFC), Backface Culling (BFC), Ray-Triangle Culling (RTC).....	33
Table 3: Firefighter Data.....	38
Table 4: Teapot Data.....	39

TABLE OF FIGURES

Figure 1: Ray overview showing the view ray interaction with objects in 3D space.....	4
Figure 2: Screen UVW Coordinates where the width and height are between +/- aspect_ratio/2 and +/- 0.5, respectively. Index pair (i=0, j=0) is the top-left corner of the screen	6
Figure 3: Camera UVW Coordinates in relation to the screen coordinates.....	7
Figure 4: (a) Spatial partitions with split planes orthogonal to the axes where axis x, y, z are colored blue, green, and red, respectively. (b) K-D Tree tree structure resulting from the partition	16
Figure 5: 2-D View Frustum Culling identifying objects that are visible (solid) and invisible (dash).....	20
Figure 6: Backface Culling identifying invisible (dash) faces/triangles. If the normal vector of the face is pointing in the same direction (+/-90°) as the camera vector, then the face is a backface	23
Figure 7: Ray-Triangle Culling example showing a view ray within the radius of the encompassing circle	26
Figure 8: The distance between the center and view ray should be less than the radius ...	27
Figure 9: Rendered Images: (a) bunny, (b) teapot	29
Figure 10: Stanford bunny – The average and maximum number of intersection tests per ray for different combinations of optimizations in comparison to ray tracing with no optimizations. VFC and BFC can reduce the intersection tests by more than 45% individually. RTC results in approximately 99% reduction by itself and when combined with the other optimizations.....	31
Figure 11: Stanford bunny – The rendering times of the different optimizations in seconds. The reduction in intersection tests does not directly translate to reduction in rendering times. VFC reduces the rendering times the most due to its early termination during traversal. RTC yields similar results but mainly due to the tremendous reduction in intersection tests. The combined optimizations show how the different culling techniques compliment each other.....	32
Figure 12: Teapot – The average and maximum number of intersection tests per ray for different combinations of optimizations in comparison to ray tracing with no optimizations. Even with a different set of data, the results are similar to the Stanford bunny.....	33
Figure 13: Teapot – The rendering times of the different optimizations in seconds. Again, the results are similar to the Stanford bunny.....	34
Figure 14: Bunny Rendered Image – 69451 triangles	36
Figure 15: Firefighter Rendered Image – 21648 triangles.....	37
Figure 16: Teapot Rendered Image – 1024 triangles.....	38

TABLE OF CODES

Code 1: General Ray Tracing Algorithm.....	5
Code 2: Screen Coordinate Calculations	6
Code 3: Camera UVW Coordinate Calculations	7
Code 4: Ray Direction and Position Calculations.....	7
Code 5: Light Ray Direction and Position Calculations.....	10
Code 6: Ambient Light Calculation.....	12
Code 7: Lighting – Diffuse Lighting	12
Code 8: Lighting – View Ray	13
Code 9: Lighting – Half-angle Approximation.....	13
Code 10: Lighting – Specular Lighting	13
Code 11: Pixel Color Calculations.....	14
Code 12: Scaling Pixel Color.....	14
Code 13: K-D Tree Build Algorithm	16
Code 14: Basic K-D Tree Traversal Algorithm.....	17
Code 15: K-D Tree Children Node Traversal Test.....	17
Code 16: Optimization Algorithm	19
Code 17: View Frustum Corners Calculations	20
Code 18: View Frustum Plane Calculations	21
Code 19: Test View Frustum with Bounding Boxes	22
Code 20: Backface Culling Algorithm	23
Code 21: Projected Triangle Calculations	24
Code 22: Encompassing Circle Center and Radius Calculations	25
Code 23: Center Ray and Delta Calculation	26
Code 24: Updated K-D Tree Traversal Algorithm	28
Code 25: Camera.h.....	40
Code 26: Camera.cpp.....	42
Code 27: KDTree.h.....	43
Code 28: KDTree.cpp	49
Code 29: Light.h	50
Code 30: Main.cpp.....	50
Code 31: Mesh.h	51
Code 32: Mesh.cpp	54
Code 33: OBJ.h.....	54
Code 34: OBJ.cpp	56
Code 35: Ray.h.....	57
Code 36: Ray.cpp.....	57
Code 37: RayTracer.h	58
Code 38: RayTracer.cpp	64

Code 39: TGA.h.....	65
Code 40: TGA.cpp	66
Code 41: Triangle.h	67
Code 42: Triangle.cpp.....	68
Code 43: Vector3D.h	69
Code 44: Vector3D.cpp	71

1. Introduction

With the increasing demands for accurate realism for modern graphics applications like animated films, ray tracing is the preferred global illumination technique, but at the same time, the amount of geometric complexity becomes the counterweight to its popularity due to slow rendering times. Many different algorithms and techniques have been developed to increase performance of ray tracers such as spatial data structures like the K-D tree [1][2], spatially coherent ray packets [3][4][5][6][7][8], culling methods [9][10][4][5][11], or optimized calculations [9][4][6]. Whichever method is used, the ultimate goal is to quickly find the intersection point between the rays and geometric data, which is usually accomplished by reducing the number of intersection tests.

Assarsson, Moller [10] investigated optimizing view frustum culling for bounding boxes and created an algorithm that exploits spatial and temporal coherency of the data in relation to the view frustum. The purpose of the view frustum is to isolate data bounded by minima boxes that is only within view. Their work reduces the number of computations needed to find the intersection between the frustum and bounding boxes. Also, some of the calculations are reused from frame to frame to avoid redundancy. This algorithm is not designed specifically for ray tracing, but the concept remains valid.

The paper written by Komatsu, Kaeriyama, Suzuki, Takizawa, Kobayashi [6] describes a frustum-triangle intersection algorithm that combines frustum culling and Moeller's ray-triangle intersection equation [12] so that it can reuse calculations for each frustum. In this case, the frustum is a group of rays that can share calculations through pre-computed values and allows for early termination.

One other approach to quickly finding the closest intersection point is discrete ray tracing [13][14][15]. This two-stage method first discretizes the data into small unit voxels (three-dimensional unit: cube) and then quickly traverses adjacent voxels one by one from the starting ray position. If the ray intersects an occupied voxel, then traversal can stop immediately as the closest intersection has been found. The concern about this method is the discretization of data. If the voxels are large, the resulting image will have aliasing issues. However, if the voxels are smaller, then more memory is needed to store those voxels.

Teller and Alex [8] have created a frustum casting method that utilizes spatial data structures, beam tracing concepts, and ray walking technique. Their approach is to divide the screen into spatially coherent frusta based on the visibility of data. This is accomplished by recursively subdividing the current frustum based on the extreme (corner) rays. The rays traverse the spatial data structure using ray walking to quickly find the closest intersecting object. Although this method takes advantage of spatial coherency, there are minimal benefits if objects have small surface areas. The worst case scenario using this method is when every pixel is tested for intersection.

Similarly, Reshetov, Soupikov, Hurley [7] designs a multi-level ray tracing algorithm (MLRTA) that traverses a K-D tree with a frustum created by beams. The beams are created by the subdivisions of the screen-space like the frustum casting method. However, MRLTA takes it further by finding the optimal entry points for rays to begin traversal in the middle of the tree structure.

This thesis proposes a method that improves ray casting for K-D trees using a ray-triangle culling technique in combination with view frustum and backface culling. The proposed method traverses the tree once to mark nodes to be inactive or skipped. Nodes are decided to be inactive if its bounding box is outside of the view frustum. Otherwise, nodes can be skipped based on the spatial coherency of the view rays with the ray-triangle frustum or the orientation of the nodes' triangle with respect to the view rays. This method only affects the traversal structure of the K-D tree for ray casting, and thus, various packet-based ray tracing methods can be used to further improve performance while having minimal overhead for non-view rays.

2. Basics of Ray Tracing

Ray tracing is a precise simulation of various light rays interacting with geometric objects (triangles, spheres, etc.) in a three-dimensional scene. With the light sources, geometric data, and position of the camera, a ray tracing application will generate an image based on the interaction between objects and light sources at the view rays' intersection point as seen in Figure 1.

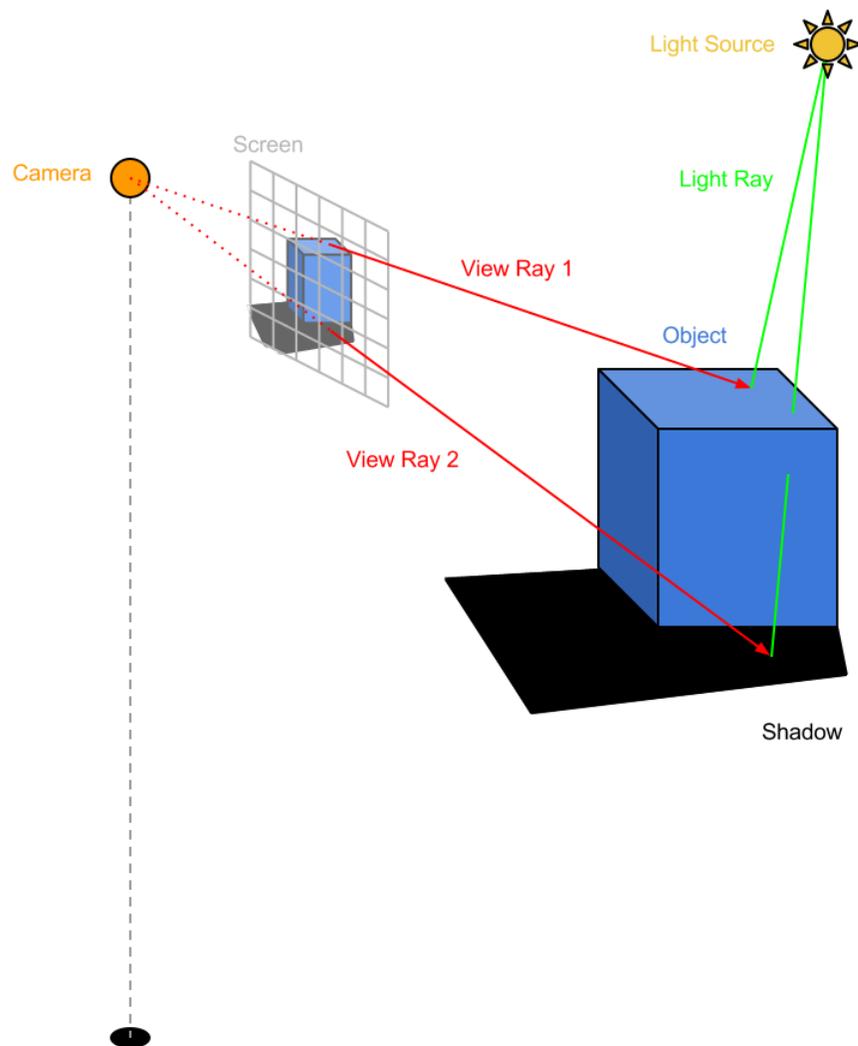


Figure 1: Ray overview showing the view ray interaction with objects in 3D space

Code 1 shows the algorithm for a basic ray tracer described in [16].

```
foreach pixel
  calculate view ray
  if ray intersects with object
    calculate lighting for intersection point
    set pixel color from lighting equation
  else
    set pixel color to background
```

Code 1: General Ray Tracing Algorithm

For each pixel, a ray is casted into the 3D scene to get the closest intersecting object. At the intersection point, the intensity of the object's color is calculated to achieve shading. This is accomplished by projecting another ray (light ray) from the intersection point to each light source. If the secondary ray intersects with any objects (view ray 2 in Figure 1), the intersection point is in shadow and the light source contributes no light. Using the total calculated light contributions with the object's material properties, the color of the pixel can be determined using a lighting model. There are many different types of lighting models, but the one used for this thesis is the Phong model [17], which will be discussed later. The following sections will explain the ray tracing process in detail.

2.1. View Rays

Before getting into the specifics of calculating the view ray, the rudimentary ray needs to be defined. A ray is composed of the starting position, direction, and length as described in the following equation (0).

$$ray = position + length * direction \quad (0)$$

A view ray (also known as primary or cast ray) is a specific type of ray that is projected from the camera's position for every pixel of the image (or screen). To calculate view

rays, the first calculation establishes the coordinates of a point on the screen for a specific pixel index pair, i and j , where the top-left corner is $(0, 0)$.

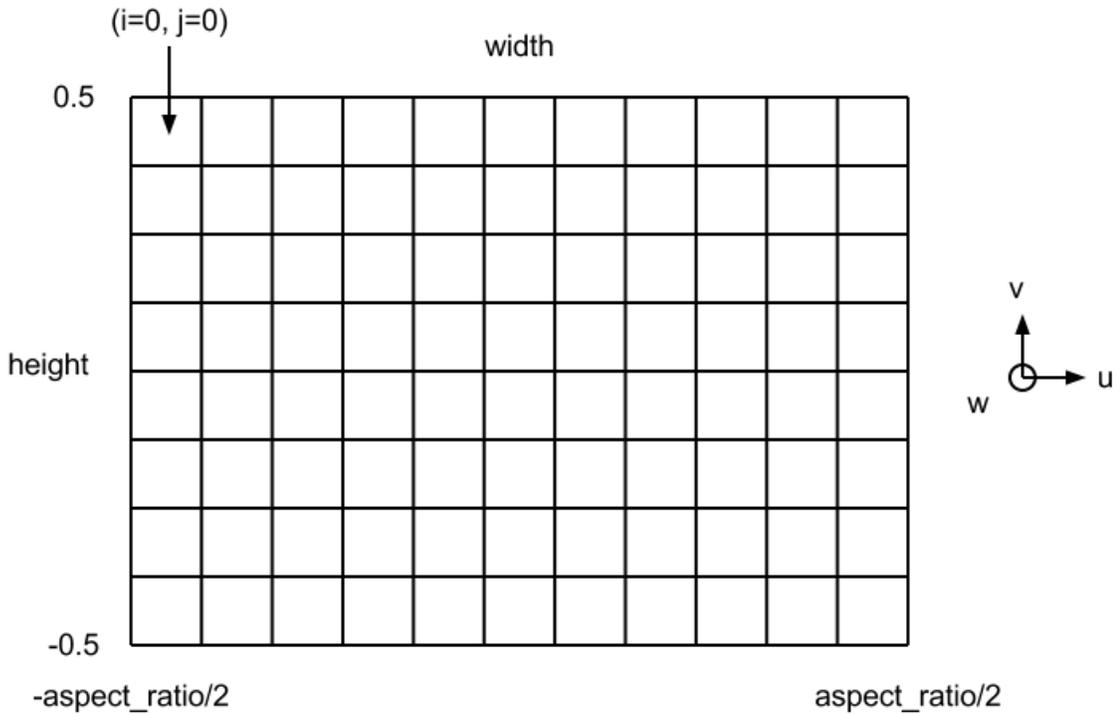


Figure 2: Screen UVW Coordinates where the width and height are between \pm aspect_ratio/2 and \pm 0.5, respectively. Index pair $(i=0, j=0)$ is the top-left corner of the screen

```

aspect_ratio = width/height
l = -aspect_ratio/2
r = aspect_ratio/2
b = -0.5
t = 0.5
s.u = (l - r)*(i + 0.5)*aspect_ratio/(width - 1) - l
s.v = (b - t)*(j + 0.5)/(height - 1) - b
s.w = 1
Normalize(s)

```

Code 2: Screen Coordinate Calculations

The next step is to calculate the uvw-coordinate of the camera.

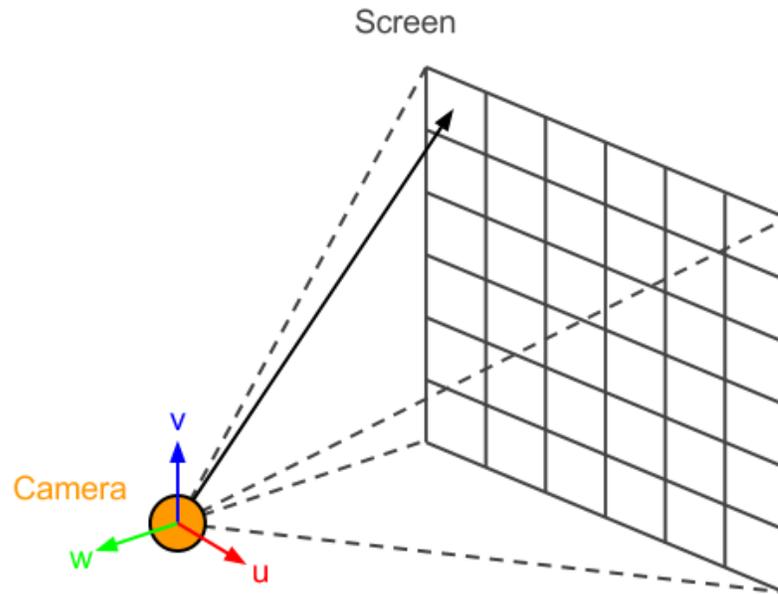


Figure 3: Camera UVW Coordinates in relation to the screen coordinates

```
w = camera.lookat - camera.position;
Normalize(w)
u = up.Cross(w)
v = w.Cross(u)
```

Code 3: Camera UVW Coordinate Calculations

Using the uvw-coordinates, the screen coordinates are transformed into the camera's coordinate system to find the ray's direction. The ray's position on the screen is the ray's direction added to the camera's position.

```
ray.direction = u*s.u + v*s.v + w*s.w
ray.position = camera.position + ray.direction
```

Code 4: Ray Direction and Position Calculations

This entire process is repeated for every pixel indices of the image.

2.2.Ray-Triangle Intersection

After the view rays are calculated, they are tested with the geometric data to find the closest intersection point. In the basic form of ray tracing, all triangles in the scene are tested for intersection. This is an inefficient method that will be addressed in the next chapter.

The following algorithm is based on the barycentric coordinates of the triangle's parametric plane [18].

For a triangle with vertices a , b , and c , the ray intersection with a parametric surface equation becomes

$$e + td = a + \beta (b - a) + \gamma (c - a) \quad (1)$$

In vector form, the intersection equations are as follows.

$$e_x + td_x = a_x + \beta (b_x - a_x) + \gamma (c_x - a_x) \quad (2)$$

$$e_y + td_y = a_y + \beta (b_y - a_y) + \gamma (c_y - a_y) \quad (3)$$

$$e_z + td_z = a_z + \beta (b_z - a_z) + \gamma (c_z - a_z) \quad (4)$$

The system of equations can then be written in the matrix form below.

$$\begin{bmatrix} a_x - b_x & a_x - c_x & d_x \\ a_y - b_y & a_y - c_y & d_y \\ a_z - b_z & a_z - c_z & d_z \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - e_x \\ a_y - e_y \\ a_z - e_z \end{bmatrix} \quad (5)$$

Then solving for the variables yields the following equations

$$\beta = \frac{\begin{vmatrix} a_x - e_x & a_x - c_x & d_x \\ a_y - e_y & a_y - c_y & d_y \\ a_z - e_z & a_z - c_z & d_z \end{vmatrix}}{|A|} \quad (6)$$

$$\gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & d_x \\ a_y - b_y & a_y - c_y & d_y \\ a_z - b_z & a_z - c_z & d_z \end{vmatrix}}{|A|} \quad (7)$$

$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - e_x \\ a_y - b_y & a_y - c_y & a_y - e_y \\ a_z - b_z & a_z - c_z & a_z - e_z \end{vmatrix}}{|A|} \quad (8)$$

$$A = \begin{bmatrix} a_x - b_x & a_x - c_x & d_x \\ a_y - b_y & a_y - c_y & d_y \\ a_z - b_z & a_z - c_z & d_z \end{bmatrix} \quad (9)$$

With the dummy variables below,

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix} \quad (10)$$

the resulting equations using Cramer's rule are

$$M = a(ei - hf) + b(gf - di) + c(dh - eg) \quad (11)$$

$$\beta = \frac{j(ei - hf) + k(gf - di) + l(dh - eg)}{M} \quad (12)$$

$$\gamma = \frac{i(ak - jb) + h(jc - al) + g(bl - kc)}{M} \quad (13)$$

$$t = -\frac{f(ak - jb) + e(jc - al) + d(bl - kc)}{M} \quad (14)$$

Finally, if any of the following conditions are met, then there is no intersection between the ray and triangle.

$$t < EPSILON \tag{15}$$

$$\gamma < 0.0 \text{ or } \gamma > 1.0 \tag{16}$$

$$\beta < 0.0 \text{ or } \beta > (1.0 - \gamma) \tag{17}$$

As a result of these tests, the distance of intersection is also computed, t . If the ray intersects the triangle, then the distance and pointer to the triangle is stored and used for comparison with other intersecting triangles. The triangle with the shortest intersecting distance is the closest, visible triangle that will then be used to calculate lighting.

2.3.Light Rays

Once an intersection point has been found for the primary ray, the color of the pixel is calculated. The first step of this calculation is to determine the contribution of all light sources at the intersection point. To test whether a light source directly contributes light, a ray is created from the intersection point to the light source as seen in Code 5.

```
light_ray.position = ray.position + ray.distance * ray.direction  
light_ray.direction = light.position - light_ray.position
```

Code 5: Light Ray Direction and Position Calculations

The light ray is then subjected to the same ray-triangle intersection test as before where all triangles are tested. If the light ray intersects with any objects/triangles (i.e., there are objects between the intersection point and the light source), the intersection point is in a shadow and the light contribution is not added in the lighting equation described in the next section.

2.4. Lighting

As mentioned before, the lighting equation is based on the Phong model. The calculation is composed of the ambient, diffuse, and specular components as seen in the equation below.

$$R_m = 2 * (L_m \cdot N) * N - L_m \quad (18)$$

$$I_p = k_a * i_a + \sum_{m \in \text{lights}} (k_d * (L_m \cdot N) * i_d + k_s * (R_m \cdot V)^\alpha * i_s) \quad (19)$$

where

I_p = color intensity at point p

i_a = ambient lighting

i_d = light source's diffuse component

i_s = light source's specular component

k_a = object's ambient reflection constant

k_d = object's diffuse reflection constant

k_s = object's specular reflection constant

α = object's shininess constant

L_m = vector from point p to light source

N = normal at point p

2.4.1. Ambient Light

The first part of the lighting equation is the calculation for ambient light, which in this case is a uniform value within in the scene. This is a crude and simple method but is

sufficient for this implementation. The ambient value is the object's color multiplied by the ambient coefficient resulting in an RGB value.

```
ambient = obj.color * obj.ambient
```

Code 6: Ambient Light Calculation

2.4.2. Diffuse Light

The second part adds the diffuse and specular contributions of each light. Since the diffuse component is uniform at all viewing angles, the calculation only consists of the normal of the surface and the light direction. This uses the property of the dot product that states when two unit vectors are dotted with each other the result is the cosine of the angle between the two vectors. If two vectors are pointing in the same direction, meaning the angle between the vectors is between -90 and +90 degrees, the dot product equates to a positive value between 0 and 1. If the result of the dot product between the surface normal and light ray is 0 or negative, the light ray is parallel or behind the surface, respectively, and no diffuse color is added. Otherwise, the resulting dot product gives the diffuse scalar value that varies based on how angled the surface is compared to light. The diffuse contribution for the pixel (RGB value) becomes the object's diffuse constant multiplied by the object's color and diffuse lighting calculation.

```
dn1 = normal.Dot(light_ray.direction)
if dn1 < 0.0
    dn1 = 0.0
diffuse = obj.diffuse*obj.color*dn1
```

Code 7: Lighting – Diffuse Lighting

2.4.3. Specular Light

For the specular lighting, the calculations are a bit more involved because specular lighting is dependent on the viewing angle. The code below calculates the view ray.

```
v = light_ray.position - camera.position  
Normalize(v)
```

Code 8: Lighting – View Ray

Instead of calculating the lights reflected ray as shown in the lighting equation (19), the half-angle vector between the view and light ray is calculated with the equation below.

$$h = \frac{V+L}{|V+L|} \quad (20)$$

```
h = v + light_ray.direction  
Normalize(h)
```

Code 9: Lighting – Half-angle Approximation

The half-angle vector dotted with the normal vector only equates to approximately half of $R_m \cdot V$. However, this is not an issue as the resulting value is raised by the power of the shininess constant. In other words, the shininess constant is adjusted to achieve the same value as $R_m \cdot V$. The specular contribution for the pixel (scalar value) becomes the object's specular constant multiplied by the specular reflection calculation.

```
svr = h.Dot(normal)  
if svr < 0.0  
    svr = 0.0  
svr = pow(svr, 1.0/roughness)  
specular = obj.specular*svr
```

Code 10: Lighting – Specular Lighting

2.4.4. Pixel Color

Lastly, the ambient, diffuse, and specular contributions are added together to get the color of the pixel as seen in Code 11. The diffuse and specular contributions are divided by the number of lights, averaging out the contributions of each light.

```
color = ambient
foreach light
    calculate diffuse and specular contributions
    color.r += (diffuse.r + specular)/num_lights
    color.g += (diffuse.g + specular)/num_lights
    color.b += (diffuse.b + specular)/num_lights
```

Code 11: Pixel Color Calculations

Code 12: then scales the color between 0 and 255 before it is written to the image file.

```
color *= 255

if color.r < 0.0
    color.r = 0
else if color.r > 255.0
    color.r = 255

if color.g < 0.0
    color.g = 0
else if color.g > 255.0
    color.g = 255

if color.b < 0.0
    color.b = 0
else if color.b > 255.0
    color.b = 255
```

Code 12: Scaling Pixel Color

3. K-D Tree

The K-D tree is an axis-aligned k-dimensional binary space partitioning tree that recursively halves space with split planes orthogonal to an axis. One of the key features of the K-D tree is its balanced branches that allows for a flatter tree structure. This means that time spent traversing the tree to find the appropriate node is minimized for different paths.

The use of the K-D tree for ray tracing significantly benefits the performance by reducing the amount of intersection tests between rays and triangles. In the previous chapter, it was said that all triangles in the scene are tested for each ray. However, if the triangle data is systematically partitioned, then the divided sections of the data can potentially be eliminated based on the position and direction of the rays. The following section will explain the process of building the tree, the conditions for traversing the tree, and the optimizations that can be made to reduce the number of intersection tests even further.

3.1. Building the K-D Tree

The key to building a balanced tree revolves around how the data is spatially divided. The commonly used method is the median plane. It is a simple method that allows for an accurately balanced tree structure and quick traversal. Additionally, the k-dimensional split simplifies the process even more by only considering one axis per tree level. Code 13 shows the recursive algorithm for the k-dimensional split and how to obtain the split plane.

```
KDTree_Build(data, axis):
```

```

if criterion has not been met
    sort data according to current axis
    get median value (split plane)
    store triangle associated with the median value
    split data into 2 subsets using on the median value
    recursively repeat with both subsets while incrementing the axis
else
    store triangles

```

Code 13: K-D Tree Build Algorithm

The criterion for stopping the recursive build is the size of the current dataset. If the size is below the threshold, the remaining data is stored in the leaf node. Otherwise, the algorithm will continue to build the tree.

If the criterion has not been met, the data is first sorted according to the current axis (i.e., if the current axis is 'x', the current data is only sorted by the x-axis values). This allows the median to easily be found and used to split the data into subsets. Each subset is associated with a child node that will recursively divide to build the rest of the tree. An example partitioned space and resulting tree structure is seen in Figure 4.

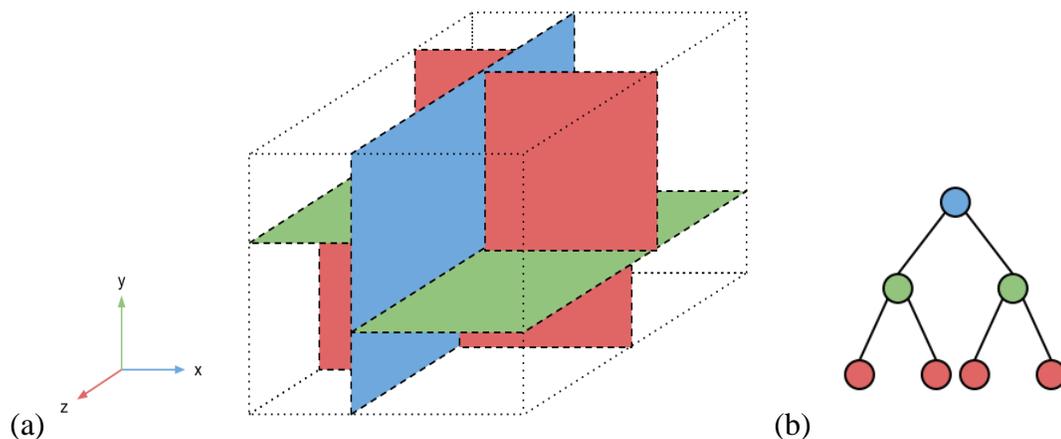


Figure 4: (a) Spatial partitions with split planes orthogonal to the axes where axis x, y, z are colored blue, green, and red, respectively. (b) K-D Tree tree structure resulting from the partition

3.2.Traversing the K-D Tree

The basic traversal process is simply a ray-triangle intersection test followed by tests to decide if both children nodes need to be traversed. Code 14 shows the general algorithm for traversing a tree.

```
Traversal(node):
    test for ray-triangle intersection
    traverse children nodes
```

Code 14: Basic K-D Tree Traversal Algorithm

The details for testing the traversal of the children nodes are shown in Code 15.

```
if !hit
    if ray.position[axis] <= split_plane
        recursively traverse left node
        if !hit && ray.direction[axis] >= -0.01
            recursively traverse right node
    else
        recursively traverse right node
        if !hit && ray.direction[axis] <= 0.01
            recursively traverse left node
```

Code 15: K-D Tree Children Node Traversal Test

The first condition tests if the current node's triangle has been hit. This means that the traversal is stopped immediately once an intersection point has been found. If there is no intersection, then a few conditions are tested to decide if both children nodes need to be traversed.

To decide whether both nodes need to be traversed, the ray position is tested against the split plane. If the ray's position is less than the split plane, then the ray begins on the left side of the split plane. Regardless of the direction, the left node is traversed to test for intersections. If no intersections were found by traversing the left node and if the ray direction is positive (pointing right), then the right node is traversed. When testing for the

direction of the ray, a small delta value is used for border cases when the ray direction is almost parallel to the split plane (ray direction approximately 0).

For cases when the position is greater than the split plane, the opposite is tested. Since the ray originates from the right side of the split plane, the right node is tested. If there are no intersections while traversing the right node and if the right direction is negative (pointing left), then the left node is traversed.

3.3.Optimizing the K-D Tree

Now with the K-D trees established, a few improvements to performance are investigated. The K-D tree is a significant contribution in terms of reducing the number of intersection tests, but the tree can be optimized specifically for ray tracing. At each level of traversal, a computationally expensive ray-triangle intersection test is performed. The optimizations discussed in this section pre-computes values as well as define tags to quickly eliminate unnecessary ray-triangle intersection tests during view ray traversal.

Code 16 shows the one-time traversal algorithm that runs right after the build process of the K-D tree by performing view frustum and backface culling to tag the nodes while calculating the ray-triangle frustum. The following sections will describe each optimization methods in detail.

```
Optimize:
  perform view frustum culling
  if node is active
    perform backface culling
    if node is not skipped
      calculate ray-triangle frustum
```

Code 16: Optimization Algorithm

3.3.1. View Frustum Culling

View frustum culling is a method that removes any data that is not within the six planes that define the viewing boundaries. By applying the view frustum culling method to the K-D tree, nodes are identified that are not within the view frustum and marked for early termination of the current traversal path. Figure 5 shows a top-down view of a view frustum culling scenario with the dotted-lined objects being culled and solid-lined objects being considered for intersection tests.

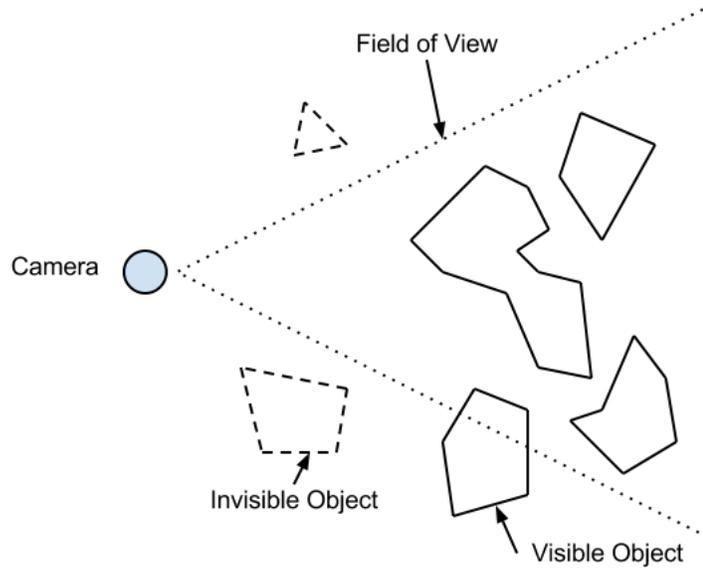


Figure 5: 2-D View Frustum Culling identifying objects that are visible (solid) and invisible (dash)

The first step for view frustum culling is to define the view frustum. When the rays are being computed (Chapter 2.1: View Ray), the corner rays are used to define the eight corners of the frustum as seen in Code 17.

```

if top left corner ray
    corners[0] = ray.position;
    corners[4] = ray.position + ray.direction * FAR_PLANE_DISTANCE;

if top right corner ray
    corners[1] = ray.position;
    corners[5] = ray.position + ray.direction * FAR_PLANE_DISTANCE;

if bottom left corner ray
    corners[2] = ray.position;
    corners[6] = ray.position + ray.direction * FAR_PLANE_DISTANCE;

if bottom right corner ray
    corners[3] = ray.position;
    corners[7] = ray.position + ray.direction * FAR_PLANE_DISTANCE;

```

Code 17: View Frustum Corners Calculations

Using the eight corners of the view frustum, the six planes can be calculated as seen in Code 18. Each plane is calculated so that the normals are pointing towards the center of

the frustum. The normals are calculated by computing the cross product between vectors created from the corners. Then the d offset is calculated by using the plane equation (20) where (a,b,c) is the normal vector and (x,y,z) is a coordinate position.

$$distance = x * a + y * b + z * c + d \quad (21)$$

If evaluating the plane equation with (x,y,z) coordinate equates to 0, then the coordinate is on the plane. If the resulting distance is a positive value, the coordinate is on the side of the plane where the normal is pointing. The opposite is true if the distance is negative. For the d offset, the plane equation has to equate to 0, meaning a point on the plane is used as the (x,y,z) coordinate. Since the corners define the plane, one of the corners is used as the coordinate.

```
top_plane.normal = (corners[4] - corners[0]).Cross(corners[1] - corners[0]);
Normalize(top_plane);
top_plane.d = -top_plane.normal.Dot(corners[0]);

right_plane.normal = (corners[1] - corners[3]).Cross(corners[7] - corners[3]);
Normalize(right_plane);
right_plane.d = -right_plane.normal.Dot(corners[7]);

bottom_plane.normal = (corners[3] - corners[2]).Cross(corners[6] -
corners[2]);
Normalize(bottom_plane);
bottom_plane.d = -bottom_plane.normal.Dot(corners[7]);

left_plane.normal = (corners[6] - corners[2]).Cross(corners[0] - corners[2]);
Normalize(left_plane);
left_plane.d = -left_plane.normal.Dot(corners[0]);

near_plane.normal = (corners[0] - corners[2]).Cross(corners[3] - corners[2]);
Normalize(near_plane);
near_plane.d = -near_plane.normal.Dot(corners[0]);

far_plane.normal = (corners[4] - corners[5]).Cross(corners[7] - corners[5]);
Normalize(far_plane);
far_plane.d = -far_plane.normal.Dot(corners[7]);
```

Code 18: View Frustum Plane Calculations

Now that the view frustum planes have been established, the culling process can begin. This involves traversing the K-D tree and testing the bounding box of the current node with the view frustum as seen in Code 19.

```
foreach plane of view frustum, p
    numOutside = 0;
    foreach corner of bounding box, c
        dist = p.normal.Dot(c) + p.d;
        if dist < 0
            numOutside++
    if numOutside == 8
        set node as inactive
        break
```

Code 19: Test View Frustum with Bounding Boxes

For each plane of the view frustum, the corners of the bounding box are tested to see which side of the plane they reside. This is done by calculating the distance of the point in relation to the plane using the plane equation (20). Since the frustum planes are calculated so that all of the normals point inside the frustum, any coordinates outside the frustum will result in a negative distance. If all corners are outside of one frustum plane, the bounding box is known to be completely outside the view frustum so there is no need to test further. The node is set to be inactive and the testing terminates for this path.

3.3.2. Backface Culling

Backface culling is a simple method that removes any faces/triangles/polygons that are facing away from the view point. Since the backs of objects are not visible, it is not necessary to test for ray intersection.

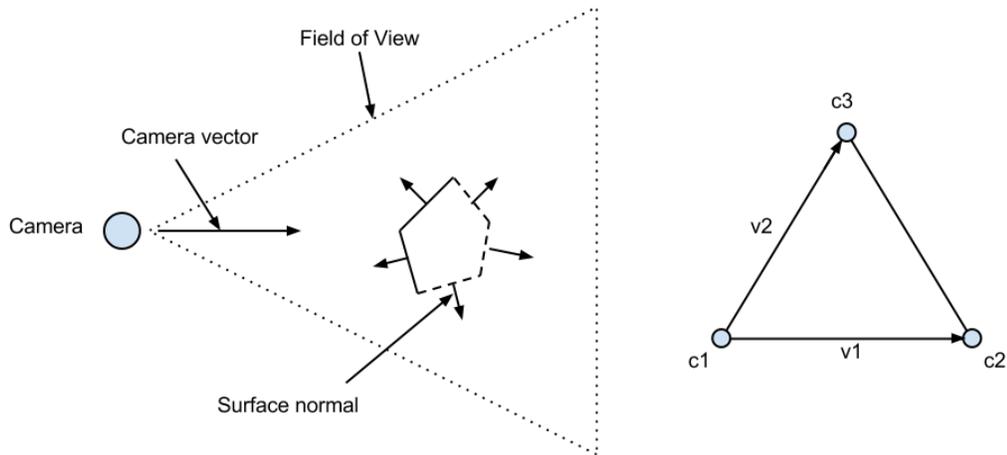


Figure 6: Backface Culling identifying invisible (dash) faces/triangles. If the normal vector of the face is pointing in the same direction (+/-90°) as the camera vector, then the face is a backface

To test if a triangle is facing the opposite direction, the normal of the triangle is dotted with the view vector. The algorithm for backface culling is shown below.

```

normal = (triangle.c2-triangle.c1).Cross(triangle.c3-triangle.c1);
Normalize(normal);
view_vector = camera.lookat - camera.position;
Normalize(view_vector);
if view_vector.Dot(normal) > EPSILON
    set node to skip

```

Code 20: Backface Culling Algorithm

The normal of the triangle is the normalized cross product of vectors v1 and v2 as seen in Figure 6. The view vector is defined as the normalized vector from the camera's position to the look-at position. The dot product between the normal and view vector gives the direction of the normal in relation to the view direction. If the result of the dot product is positive (the epsilon value accounts for edge cases), then the view vector and triangle normal are point in the same direction, meaning the triangle is a backface. The node is marked to be skipped during ray traversal.

3.3.3. Ray-Triangle Culling

Ray-triangle culling is another quick method to reduce unnecessary ray-triangle intersection tests. It projects the triangle onto the near plane, finds the optimum circle and radius encompassing the projected triangle, calculates the ray from the center of the circle to the camera, and computes the maximum delta value between the actual and center ray. During ray traversal, the center ray and delta value are used to quickly test for rays by using the dot product. If the result of the dot product is less than the delta value, then the ray-triangle intersection test is performed.

```
v = camera.position - triangle.c1;
Normalize(v);
d =
(triangle.c1.Dot(near_plane.normal)+near_plane.d)/(v.Dot(near_plane.normal));
c1 = triangle.c1 - v * d;

v = camera.position - triangle.c2;
Normalize(v);
d =
(triangle.c2.Dot(near_plane.normal)+near_plane.d)/(v.Dot(near_plane.normal));
c2 = triangle.c2 - v * d;

v = camera.position - triangle.c3;
Normalize(v);
d =
(triangle.c3.Dot(near_plane.normal)+near_plane.d)/(v.Dot(near_plane.normal));
c3 = triangle.c3 - v * d;
```

Code 21: Projected Triangle Calculations

To find the radius for the encompassing circle, the projected positions of the triangle vertices on the near plane are calculated using the following equations.

$$t = -\frac{p_0 \cdot N + d}{V \cdot N} \quad (22)$$

$$r = p_0 + t * d \quad (23)$$

(21) calculated the distance between the point p_0 (triangle vertex) along the vector V and the plane (near plane) defined by the normal N and offset d . Using this distance, the ray equation (22) calculates the position from p_0 along the direction d for the distance t .

```
l1 = Length(c1 - c2);
l2 = Length(c1 - c3);
l3 = Length(c2 - c3);
if l1 > l2
    if l1 > l3
        center = (c1+c2)/2;
        radius = Length(c1-center);
    else
        center = (c2+c3)/2;
        radius = Length(c2-center);
else
    if l2 > l3
        center = (c1+c3)/2;
        radius = Length(c1-center);
    else
        center = (c2+c3)/2;
        radius = Length(c2-center);
```

Code 22: Encompassing Circle Center and Radius Calculations

With the triangle vertices projected on the near plane, the center of the encompassing circle is calculated by finding the longest edge between the projected vertices and averaging the positions of the two projected vertices. Then the radius becomes the length between the center and one of the projected vertices of the longest edge. This results in an encompassing circle that intersects two of the projected vertices and tightly fits around the projected triangle. Figure 7 shows the calculated radius in blue and the encompassing circle in black.

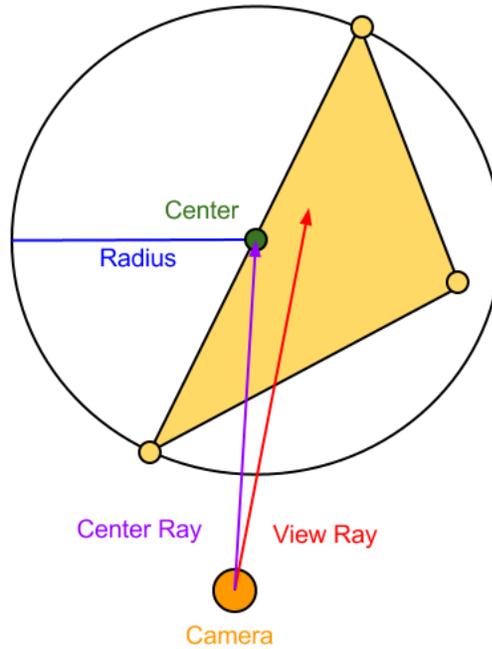


Figure 7: Ray-Triangle Culling example showing a view ray within the radius of the encompassing circle

```

node.center_ray = center - camera.position;
Normalize(node.center_ray);
if (radius <= 1.0) {
    node.delta = sqrt(1-(radius * radius))-0.001;
else
    node.delta = 0;

```

Code 23: Center Ray and Delta Calculation

Using the calculated center of the encompassing circle, the center ray is calculated by subtracting the camera's position from the circle's center. The center ray is then normalized to be used in the dot product during ray traversal.

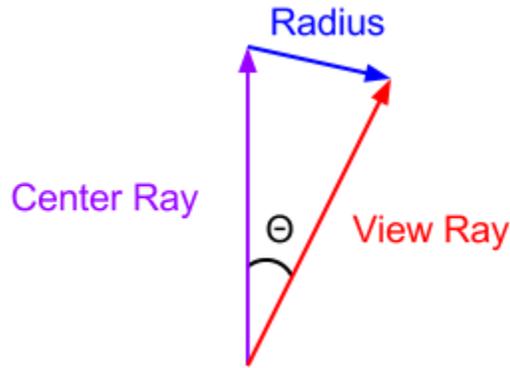


Figure 8: The distance between the center and view ray should be less than the radius

The delta value is calculated so that distance between the view and center ray is less than the radius. If the view ray is vector a , the center ray is vector b , and the distance between the two vectors is x , then

$$a \cdot b = \cos(\theta) \quad (24)$$

$$x = \text{length}(b) * \sin(\theta) \quad (25)$$

Since the center ray b is a unit vector, $\text{length}(b)$ is equal to one so the distance x simply becomes

$$x = \sin(\theta) \quad (26)$$

To solve for x in terms of a and b ,

$$\theta = \arccos(a \cdot b) \quad (27)$$

$$x = \sin(\arccos(a \cdot b)) = \sqrt{1 - (a \cdot b)^2} \quad (28)$$

The distance x has to be smaller than the radius, so the equations can be rewritten as

$$\sqrt{1 - (a \cdot b)^2} < radius \quad (29)$$

and rearranged to

$$a \cdot b > \sqrt{1 - radius^2} \quad (30)$$

Code 23 shows the pre-computed delta value using (30). Then during ray traversal, only the view and center ray need to be dotted with each other and compared with the delta value.

3.4.Updating the Traversal Algorithm

With the reduced tree from the three methods discussed earlier, the traversal process becomes the following.

```
Traversal(node):
    if node is active
        if node is not skipped and
            node.center_ray.Dot(ray.direction) > node.delta
            test for ray-triangle intersection
            traverse children nodes
```

Code 24: Updated K-D Tree Traversal Algorithm

With the traversal of children nodes being exactly the same as Code 15, the additional optimizations have minimal impact on the calculations per traversal. As the algorithm is recursively traversing the tree, the node is first tested if it is active. If not, the path is terminated and the recursion of other paths continues. The next condition tests if the ray-triangle intersection test can be skipped. If the node is skipped, the traversal continues to the children nodes.

4. Results and Analysis

To test the effectiveness of the optimizations, the K-D tree is built so that each node has a maximum of one triangle. Then a series of test cases are run to see how each optimization performs individually and how they perform together by measuring the average number of intersection tests per ray, the maximum number of intersections tests for all rays, and the rendering times. Although the rendering times are recorder, the actual time is not as important as the delta between the different optimizations. The ray tracing code was written as a proof of concept to showcase the key measurement, which is the reduced number of intersection tests.



Figure 9: Rendered Images: (a) bunny, (b) teapot

The objects used for the series of tests are the Stanford bunny model with 69451 triangles and teapot with 1024 triangles as seen in Figure 9. Each test uses a different combination of the three optimizations to show their capabilities of working individually and together.

4.1.Bunny

Since the bunny is a larger model, only a portion of the bunny is rendered to show the effects of view frustum culling. Table 1 shows the test results for each optimization: view frustum culling (VFC), backface culling (BFC), and ray-triangle culling (RTC).

Optimizations	Avg. Intersection Tests	Max Intersection Tests	Avg. Rendering Time [s]
None	1498	69451	924.7164
VFC	910	40187	570.3152
BFC	823	36164	775.062
RTC	10	1101	615.2254
BFC+RTC	8	697	616.7252
VFC+BFC	433	17331	488.5744
VFC+RTC	25	1795	367.8758
VFC+BFC+RTC	23	1334	366.9638

Table 1: Bunny Test Results: View Frustum Culling (VFC), Backface Culling (BFC), Ray-Triangle Culling (RTC)

With no optimizations, the average number of intersection tests per ray is 1498 while the maximum is 69451, which means all of the triangles were tested for intersection for at least one ray. Looking at the backface culling results, the average number of intersection tests is reduced by 45.1% and the maximum is reduced by 47.9%. Since the model is almost symmetrical, the results are expected. The ray-triangle culling method did very well in reducing the average number of intersection tests by about 99.3% and the maximum by 98.4%. With the specific point of view, the view frustum culling was able to cull more than 39.2% of the intersection tests.

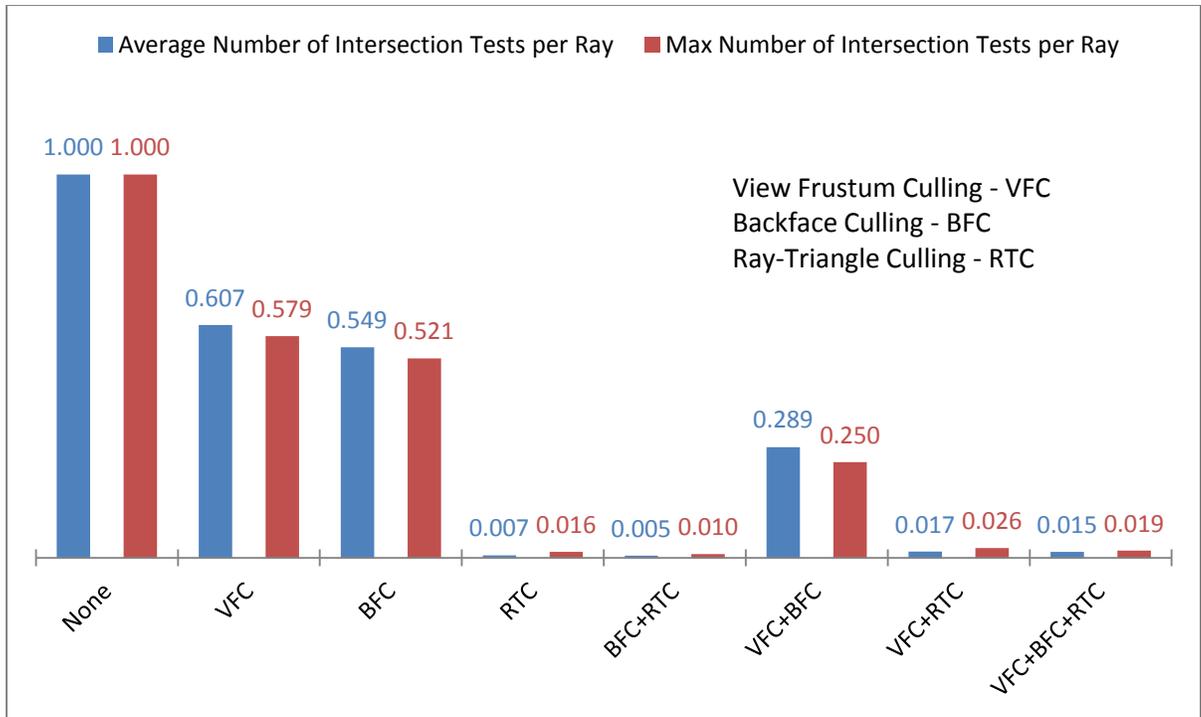


Figure 10: Stanford bunny – The average and maximum number of intersection tests per ray for different combinations of optimizations in comparison to ray tracing with no optimizations. VFC and BFC can reduce the intersection tests by more than 45% individually. RTC results in approximately 99% reduction by itself and when combined with the other optimizations.

When combining the different optimizations, backface culling and ray-triangle culling worked very well together with a total reduction of 99.5% to the average and 99.0% to the maximum number of intersection tests. However, when combining with the view frustum culling optimization, both of the average and maximum number of tests slightly increased, but the total rendering time is significantly reduced by 62.1% from the rendering time without any optimizations. The increase in tests is caused by the conservative nature of the view frustum culling at the edges of the screen. Nevertheless, the view frustum culling algorithm’s early termination quickly eliminate paths from traversal and results in a faster rendering time.

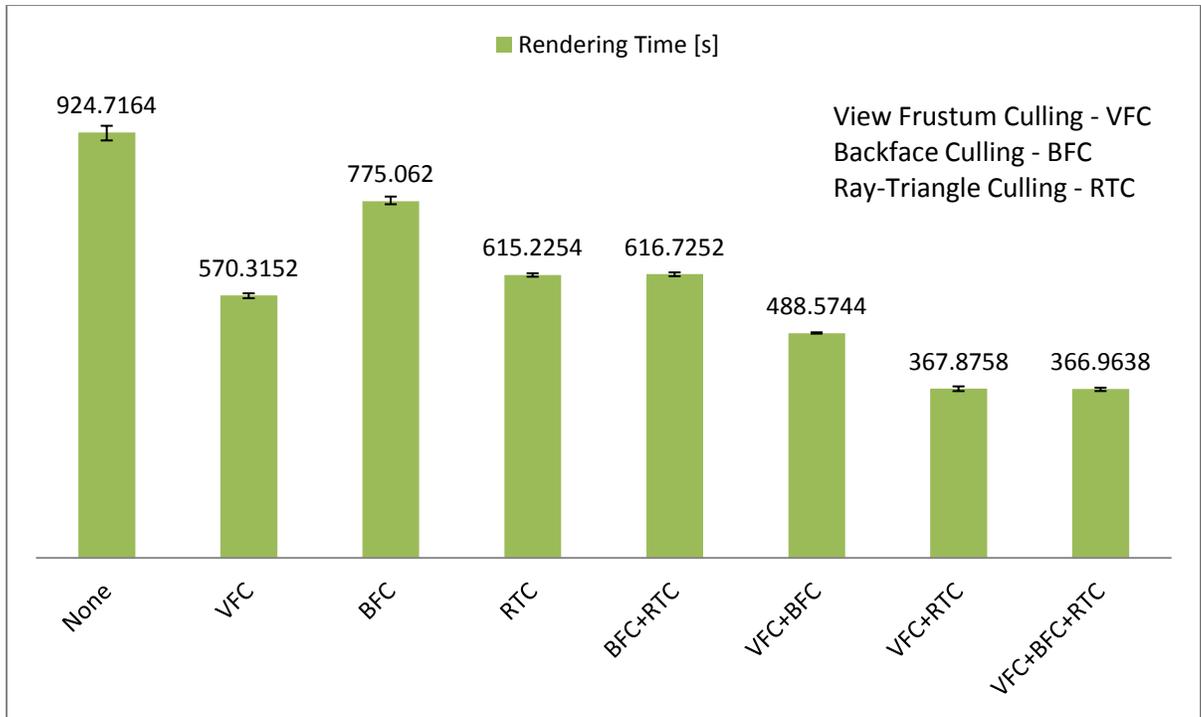


Figure 11: Stanford bunny – The rendering times of the different optimizations in seconds. The reduction in intersection tests does not directly translate to reduction in rendering times. VFC reduces the rendering times the most due to its early termination during traversal. RTC yields similar results but mainly due to the tremendous reduction in intersection tests. The combined optimizations show how the different culling techniques compliment each other.

Figure 11 shows the rendering times for the different combinations of optimizations. The backface culling plus ray-triangle culling combination yields a 36.8% improvement. With the addition of the view frustum culling, the total rendering time is reduced by 62.1%. Each optimization compliments the others well as the culling techniques can reduce the rendering times even further when combine together.

4.2. Teapot

The next set of test cases uses a smaller model that is entirely encompassed in the view frustum. Therefore, any test cases with view frustum culling will see no additional benefits, but at the same time, there are no extra costs. The results are shown in Table 2.

Optimizations	Avg. Intersection Tests	Max Intersection Tests	Avg. Rendering Time [s]
None	112	1024	64.4
VFC	112	1024	64.2
BFC	68	598	56.4
RE	0.72	127	40.6
BFC+RE	0.55	76	40.8
VFC+BFC	68	598	55.4
VFC+RE	0.72	127	41.6
VFC+BFC+RE	0.55	76	41.4

Table 2: Teapot Test Results: View Frustum Culling (VFC), Backface Culling (BFC), Ray-Triangle Culling (RTC)

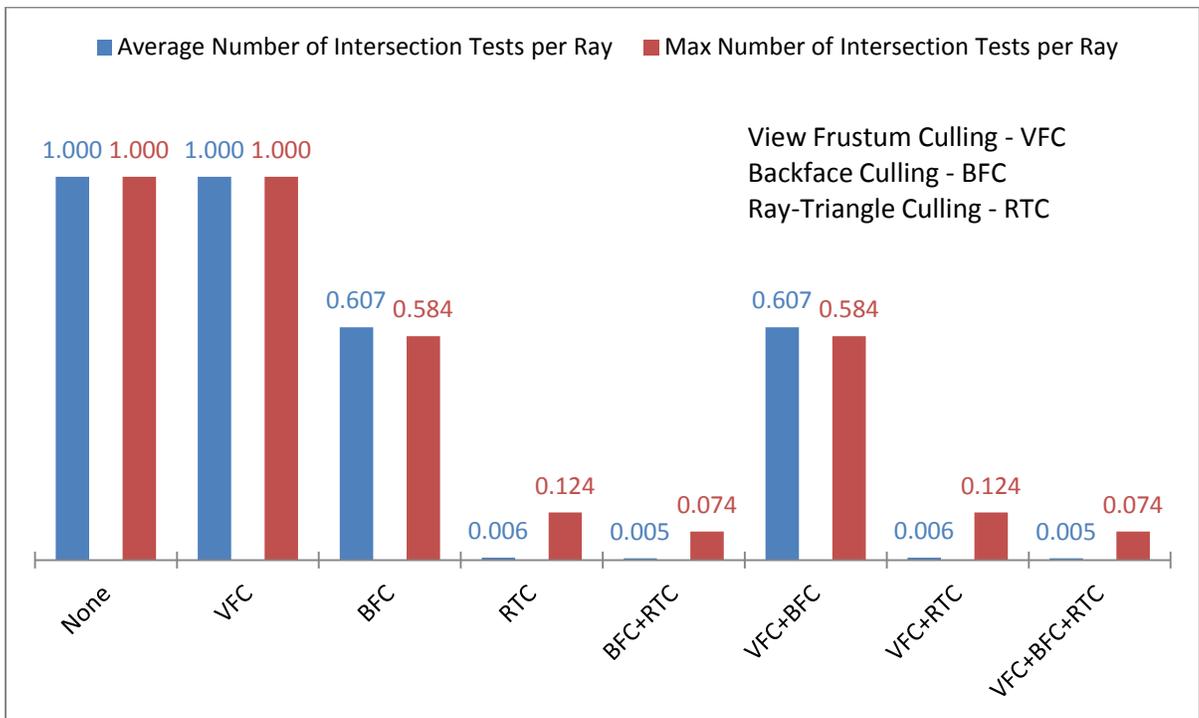


Figure 12: Teapot – The average and maximum number of intersection tests per ray for different combinations of optimizations in comparison to ray tracing with no optimizations. Even with a different set of data, the results are similar to the Stanford bunny.

Backface culling reduced the average number of intersection tests of 39.2% and the maximum by 41.6%. Like the results for the previous model, the ray-triangle culling reduced the average by 99.3% and the maximum by 87.6%. The combination of the two

resulted in a 99.5% reduction for the average number of tests and 92.6% for the maximum.

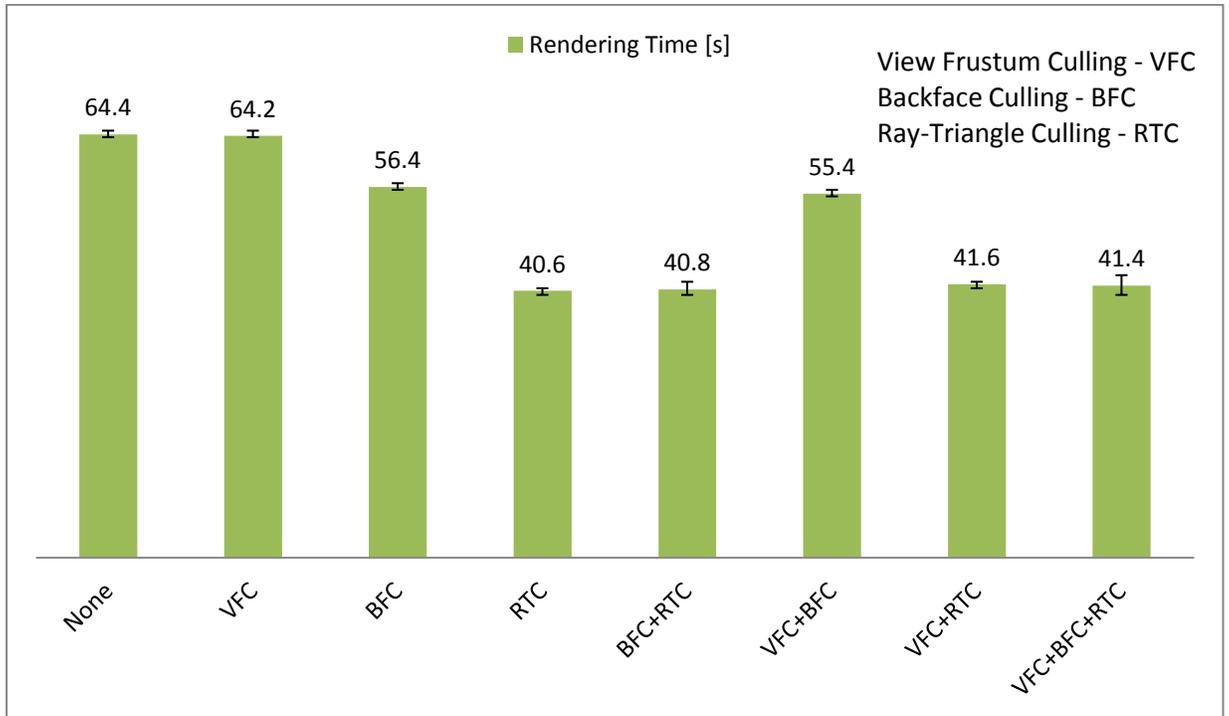


Figure 13: Teapot – The rendering times of the different optimizations in seconds. Again, the results are similar to the Stanford bunny.

Again, the ray-triangle culling optimization decreases the rendering time more than backface culling as seen in Figure 12. The speed up is 12.4% for backface culling and 37.0% for ray-triangle culling. As for the combination, the total improvement for rendering time is also 37.0%.

5. Conclusion

The proposed optimizations for the K-D tree using view frustum culling, backface culling, and ray-triangle culling have shown to significantly reduce to the number of intersection tests per ray for ray casting. For more detailed results, please refer to the Appendices.

- Average number intersection tests per ray: 98% to 99%
- Maximum intersection tests per ray: 92% to 99%

As a result, the rendering times are reduced by 37% to 62%.

This method, however, is only limited to view rays but still allows for reflected/refracted rays and global illumination. Since the nodes of the K-D tree are only marked to be inactive or skipped, these tags can be disregarded for non-view rays during traversal. The overhead of checking the type of ray is inconsequential to the resulting rendering time.

The proposed method can be improved further with an optimized tree traversal algorithm like the screen-space subdivision methods [8] along with similar packet-based ray tracing methods [3][4][5][6][7]. They have shown to be effective algorithms at reducing the number of tree traversals needed to find the correct intersection points by grouping similar rays and finding alternative entry nodes. Another method that can exploit the spatial coherency of rays is to parallelize the ray tracer. The screen-space can be divided equally for all cores and split the tree traversal into separate threads.

6. Appendices



Figure 14: Bunny Rendered Image – 69451 triangles

None	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	1498	1498	1498	1498	1498
Max intersection tests	69451	69451	69451	69451	69451
Rendering Time [s]	933.767	911.087	939.396	907.551	931.781
VFC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	910	910	910	910	910
Max intersection tests	40187	40187	40187	40187	40187
Rendering Time [s]	573.756	575.179	564.423	568.707	569.511
BFC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	823	823	823	823	823
Max intersection tests	36164	36164	36164	36164	36164
Rendering Time [s]	785.51	772.852	770.719	777.382	768.847
RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	10	10	10	10	10
Max intersection tests	1101	1101	1101	1101	1101
Rendering Time [s]	615.381	611.328	618.771	618.706	611.941
BFC+RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	8	8	8	8	8
Max intersection tests	697	697	697	697	697
Rendering Time [s]	615.142	620.686	616.132	619.332	612.334
VFC+BFC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	433	433	433	433	433
Max intersection tests	17331	17331	17331	17331	17331
Rendering Time [s]	489.071	487.837	487.548	488.153	490.263

VFC+RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	25	25	25	25	25
Max intersection tests	1795	1795	1795	1795	1795
Rendering Time [s]	372.773	362.511	363.17	372.453	368.472
VFC+BFC+RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	23	23	23	23	23
Max intersection tests	1334	1334	1334	1334	1334
Rendering Time [s]	369.377	369.962	368.752	362.616	364.112



Figure 15: Firefighter Rendered Image – 21648 triangles

None	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	1873	1873	1873	1873	1873
Max intersection tests	21648	21648	21648	21648	21648
Rendering Time [s]	1180.04	1195.19	1192.39	1175.24	1196.82
VFC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	857.77	857.77	857.77	857.77	857.77
Max intersection tests	9870	9870	9870	9870	9870
Rendering Time [s]	535.007	534.515	515.781	525.851	532.112
BFC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	967.482	967.482	967.482	967.482	967.482
Max intersection tests	10955	10955	10955	10955	10955
Rendering Time [s]	921.588	900.634	901.408	908.85	893.666
RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	2	2	2	2	2
Max intersection tests	1005	1005	1005	1005	1005
Rendering Time [s]	712.723	721.091	723.268	715.374	741.235
BFC+RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	1.6	1.6	1.6	1.6	1.6

Max intersection tests	588	588	588	588	588
Rendering Time [s]	741.623	742.093	773.094	757.348	755.919
VFC+BFC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	462	462	462	462	462
Max intersection tests	5157	5157	5157	5157	5157
Rendering Time [s]	412.935	432.369	442.73	430.887	409.839
VFC+RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	21	21	21	21	21
Max intersection tests	1222	1222	1222	1222	1222
Rendering Time [s]	340.404	333.275	326.377	341.241	346.581
VFC+BFC+RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	21	21	21	21	21
Max intersection tests	805	805	805	805	805
Rendering Time [s]	317.199	307.651	321.072	308.216	319.33

Table 3: Firefighter Data



Figure 16: Teapot Rendered Image – 1024 triangles

None	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	133	133	133	133	133
Max intersection tests	1024	1024	1024	1024	1024
Rendering Time [s]	78	77	77	77	77
VFC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	133	133	133	133	133
Max intersection tests	1024	1024	1024	1024	1024
Rendering Time [s]	78	79	78	79	79
BFC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	78	78	78	78	78
Max intersection tests	598	598	598	598	598

Rendering Time [s]	67	67	66	67	66
RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	44	44	44	44	44
Max intersection tests	908	908	908	908	908
Rendering Time [s]	60	60	59	59	60
BFC+RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	25	25	25	25	25
Max intersection tests	521	521	521	521	521
Rendering Time [s]	57	57	57	56	57
VFC+BFC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	78	78	78	78	78
Max intersection tests	598	598	598	598	598
Rendering Time [s]	66	67	67	67	67
VFC+RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	44	44	44	44	44
Max intersection tests	908	908	908	908	908
Rendering Time [s]	59	59	59	59	60
VFC+BFC+RTC	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. intersection tests	25	25	25	25	25
Max intersection tests	521	521	521	521	521
Rendering Time [s]	56	57	57	57	57

Table 4: Teapot Data

```

#ifndef CAMERA_H
#define CAMERA_H

#include <list>

#include "Ray.h"
#include "Vector3D.h"

struct Plane {
    Vector3D normal;
    double d;
};

class Camera {
public:
    Camera(int width, int height);
    ~Camera();

    void GenerateRays(std::list<Ray> *rays);
public:
    Vector3D position;
    Vector3D lookat;
};

```

```

    Plane planes[6];
private:
    int width, height;
    double aspect_ratio;

    double l, r, t, b;
    double rlw, tbh;
};

#endif // CAMERA_H

```

Code 25: Camera.h

```

#include "Camera.h"

Camera::Camera(int width, int height) {
    this->width = width;
    this->height = height;
    aspect_ratio = (double)width/height;

    position.x = 0.0;
    position.y = 1.0;
    position.z = 1.0;
    lookat.x = 0.0;
    lookat.y = 0.0;
    lookat.z = 0.0;

    l = -aspect_ratio/2.0;
    r = aspect_ratio/2.0;
    b = -0.5;
    t = 0.5;
    rlw = aspect_ratio/(width-1);
    tbh = 1.0/(height-1);
}

Camera::~Camera() {
}

void Camera::GenerateRays(std::list<Ray> *rays) {
    Ray ray;
    Vector3D u, v, w;
    Vector3D up(0.0, 1.0, 0.0);
    Vector3D s, t;
    int i, j;

    Vector3D corners[8];
    Vector3D v1, v2;

    w = lookat-position;
    w.Normalize();
    u = up.Cross(w);
    v = w.Cross(u);
}

```

```

s.z = 1.0;
for (j = 0; j < height; j++) {
    s.y = -(b+tbh*(j+0.5));
    for (i = 0; i < width; i++) {
        s.x = -(l+r1w*(i+0.5));
        t = s;
        t.Normalize();
        ray.dir = u*t.x + v*t.y + w*t.z;
        ray.pos = position + ray.dir;
        ray.dir.Normalize();
        ray.pixel = j*width + i;
        rays->push_back(ray);
        if (i == 0 && j == 0) {
            corners[0] = ray.pos;
            corners[4] = ray.pos + ray.dir*10000;
        }
        if (i == (width-1) && j == 0) {
            corners[1] = ray.pos;
            corners[5] = ray.pos + ray.dir*10000;
        }
        if (i == 0 && j == (height-1)) {
            corners[2] = ray.pos;
            corners[6] = ray.pos + ray.dir*10000;
        }
        if (i == (width-1) && j == (height-1)) {
            corners[3] = ray.pos;
            corners[7] = ray.pos + ray.dir*10000;
        }
    }
}

// Top
v1 = corners[4] - corners[0];
v2 = corners[1] - corners[0];
planes[0].normal = v1.Cross(v2);
planes[0].normal.Normalize();
// Right
v1 = corners[1] - corners[3];
v2 = corners[7] - corners[3];
planes[1].normal = v1.Cross(v2);
planes[1].normal.Normalize();
// Bottom
v1 = corners[3] - corners[2];
v2 = corners[6] - corners[2];
planes[2].normal = v1.Cross(v2);
planes[2].normal.Normalize();
// Left
v1 = corners[6] - corners[2];
v2 = corners[0] - corners[2];
planes[3].normal = v1.Cross(v2);
planes[3].normal.Normalize();
// Near
v1 = corners[0] - corners[2];
v2 = corners[3] - corners[2];

```

```

planes[4].normal = v1.Cross(v2);
planes[4].normal.Normalize();
// Far
v1 = corners[4] - corners[5];
v2 = corners[7] - corners[5];
planes[5].normal = v1.Cross(v2);
planes[5].normal.Normalize();

planes[0].d = -planes[0].normal.Dot(corners[0]);
planes[1].d = -planes[1].normal.Dot(corners[7]);
planes[2].d = -planes[2].normal.Dot(corners[7]);
planes[3].d = -planes[3].normal.Dot(corners[0]);
planes[4].d = -planes[4].normal.Dot(corners[0]);
planes[5].d = -planes[5].normal.Dot(corners[7]);
}

```

Code 26: Camera.cpp

```

#ifndef KD_TREE_H
#define KD_TREE_H

#include <vector>

#include "Camera.h"
#include "Mesh.h"
#include "Ray.h"
#include "Triangle.h"
#include "Vector3D.h"

struct KD_Node {
    KD_Node *left, *right;
    Face *face;
    int axis;
    Vector3D min, max;
    bool inactive, skip;
    Vector3D sp;
    double sp_radius;
    KD_Node() { left = 0; right = 0; face = 0; axis = 0; inactive = false; skip
= false; }
};

class KD_Tree {
public:
    KD_Tree();
    ~KD_Tree();

    void Build();
    void Cull(Camera *cam, KD_Node *node);
    void Intersect(Ray *ray, KD_Node *node, bool &hit, bool optimize);
    void Delete(KD_Node *node);
public:

```

```

    KD_Node *root;
    Mesh *object;
    Vector3D position;
};

#endif

```

Code 27: KDTree.h

```

#include <iostream>
#include "KD_Tree.h"

KD_Tree::KD_Tree() {
    root = 0;
}

KD_Tree::~KD_Tree() {
}

void KD_Tree::Build() {
    KD_Node *node, *left_node, *right_node;
    std::vector<KD_Node *> node_stack;
    std::vector<std::vector<Face *>> build_stack;
    std::vector<Face *> *current_stack;
    std::vector<Face *> right_stack, left_stack;
    Face *f;
    int i, j, k, l, m;
    double x;
    double median;
    unsigned int num_faces;
    unsigned int a;

    Vector3D um;

    if (object == 0) {
        std::cout << "Need objects to build kd-tree.\n";
    }

    // Create root node
    root = new KD_Node();
    root->axis = 0;
    root->max = object->max;
    root->min = object->min;
    node_stack.push_back(root);

    build_stack.push_back(object->faces); // Assumes only one object
    while (!build_stack.empty()) {
        node = node_stack.back();
        current_stack = &build_stack.back();

        // Get number of faces in current stack

```

```

num_faces = current_stack->size();

// Calculate half of total number of faces
if (num_faces & 1) {
    k = num_faces/2;
}
else {
    k = num_faces/2-1;
}

// Get median
l = 0;
m = num_faces-1;
while (l < m) {
    x = (*current_stack)[k]->max[node->axis];
    i = l;
    j = m;
    do {
        while ((*current_stack)[i]->max[node->axis] < x) {
            i++;
        }
        while (x < (*current_stack)[j]->max[node->axis]) {
            j--;
        }
        if (i <= j) {
            f = (*current_stack)[i];
            (*current_stack)[i] = (*current_stack)[j];
            (*current_stack)[j] = f;
            i++;
            j--;
        }
    } while (i <= j);
    if (j < k) {
        l = i;
    }
    if (k < i) {
        m = j;
    }
}
median = (*current_stack)[k]->max[node->axis];
node->face = (*current_stack)[k];

// Add to stack
left_stack.clear();
right_stack.clear();
for (a = 0; a < num_faces; a++) {
    if ((*current_stack)[a]->max[node->axis] <= median) {
        if (a != k) {
            // Left
            left_stack.push_back((*current_stack)[a]);
        }
    }
    else {
        // Right
        right_stack.push_back((*current_stack)[a]);
    }
}

```

```

    }
}

// Pop current stack from build stack
build_stack.pop_back();

// Pop current node from node stack
node_stack.pop_back();

if (right_stack.size() != 0) {
    // Add right stacks to build stack
    build_stack.push_back(right_stack);

    // Create right node
    right_node = new KD_Node();

    // Link right nodes to current node
    node->right = right_node;

    // Set axis
    node->right->axis = (node->axis+1)%3;

    // Set max and min
    node->right->max = node->max;
    node->right->min = node->min;
    if (node->axis == 0) {
        node->right->min.x = median;
    }
    if (node->axis == 1) {
        node->right->min.y = median;
    }
    if (node->axis == 2) {
        node->right->min.z = median;
    }

    // Add right nodes to node stack
    node_stack.push_back(right_node);
}
if (left_stack.size() != 0) {
    // Add left stack to build stack
    build_stack.push_back(left_stack);

    // Create left node
    left_node = new KD_Node();

    // Link left nodes to current node
    node->left = left_node;

    // Set axis
    node->left->axis = (node->axis+1)%3;

    // Set max and min
    node->left->max = node->max;
    node->left->min = node->min;
    if (node->axis == 0) {

```

```

        node->left->max.x = median;
    }
    if (node->axis == 1) {
        node->left->max.y = median;
    }
    if (node->axis == 2) {
        node->left->max.z = median;
    }

    // Add left nodes to node stack
    node_stack.push_back(left_node);
}
}
}

void KD_Tree::Cull(Camera *cam, KD_Node *node) {
    Mesh *obj;
    Vector3D corners[8], max, min;
    Vector3D normal, v, sp, c1 ,c2, c3, cd;
    Vector3D cv;
    Triangle tri;
    int p, c;
    double dist, l1, l2, l3, radius;
    int numOutside;

    if (node != 0 && node->inactive == false) {
        // VFC
        max = node->max+position;
        min = node->min+position;
        corners[0].x = min.x;
        corners[0].y = max.y;
        corners[0].z = max.z;

        corners[1].x = max.x;
        corners[1].y = max.y;
        corners[1].z = max.z;

        corners[2].x = min.x;
        corners[2].y = min.y;
        corners[2].z = max.z;

        corners[3].x = max.x;
        corners[3].y = min.y;
        corners[3].z = max.z;

        corners[4].x = min.x;
        corners[4].y = max.y;
        corners[4].z = min.z;

        corners[5].x = max.x;
        corners[5].y = max.y;
        corners[5].z = min.z;

        corners[6].x = min.x;
        corners[6].y = min.y;

```

```

corners[6].z = min.z;

corners[7].x = max.x;
corners[7].y = min.y;
corners[7].z = min.z;

for (p = 0; p < 6; p++) {
    numOutside = 0;
    for (c = 0; c < 8; c++) {
        dist = cam->planes[p].normal.Dot(corners[c]) + cam->planes[p].d;
        if (dist < 0) {
            // Outside
            numOutside++;
        }
    }
    if (numOutside == 8) {
        node->inactive = true;
        break;
    }
}

if (node->inactive == false) {
    obj = object;
    tri.c1 = *(obj->vertices[node->face->vertex[0]])+position;
    tri.c2 = *(obj->vertices[node->face->vertex[1]])+position;
    tri.c3 = *(obj->vertices[node->face->vertex[2]])+position;

    // Back face culling
    normal = (tri.c2-tri.c1).Cross(tri.c3-tri.c1);
    normal.Normalize();
    cv = (cam->lookat - cam->position);
    cv.Normalize();
    if (cv.Dot(normal) > 0.05) {
        node->skip = true;
    }
    else {
        v = cam->position-tri.c1;
        v.Normalize();
        c1 = tri.c1 - v*(tri.c1.Dot(cam->planes[4].normal) + cam->planes[4].d)/(v.Dot(cam->planes[4].normal));
        v = cam->position-tri.c2;
        v.Normalize();
        c2 = tri.c2 - v*(tri.c2.Dot(cam->planes[4].normal) + cam->planes[4].d)/(v.Dot(cam->planes[4].normal));
        v = cam->position-tri.c3;
        v.Normalize();
        c3 = tri.c3 - v*(tri.c3.Dot(cam->planes[4].normal) + cam->planes[4].d)/(v.Dot(cam->planes[4].normal));
        l1 = (c1 - c2).Length();
        l2 = (c1 - c3).Length();
        l3 = (c2 - c3).Length();
        if (l1 > l2) {
            if (l1 > l3) {
                cd = (c1+c2)/2;
                radius = (c1-cd).Length();
            }
        }
    }
}

```

```

        }
        else {
            cd = (c2+c3)/2;
            radius = (c2-cd).Length();
        }
    }
    else {
        if (l2 > l3) {
            cd = (c1+c3)/2;
            radius = (c1-cd).Length();
        }
        else {
            cd = (c2+c3)/2;
            radius = (c2-cd).Length();
        }
    }

    node->sp = cd-cam->position;
    node->sp.Normalize();

    if (radius <= 1.0) {
        node->sp_radius = sqrt(1-(radius*radius))-0.001;
    }
    else {
        node->sp_radius = 0;
    }
}

// Continue traversal
this->Cull(cam, node->left);
this->Cull(cam, node->right);
}
}
}

void KD_Tree::Intersect(Ray *ray, KD_Node *node, bool &hit, bool optimize) {
    Mesh *obj;
    Triangle tri;
    double temp;
    Vector3D v;

    if (node != 0) {
        if (!node->skip || !optimize) {
            if (node->sp.Dot(ray->dir) > node->sp_radius || !optimize) {
                obj = object;

                // Triangle intersection
                tri.c1 = *(obj->vertices[node->face->vertex[0]])+position;
                tri.c2 = *(obj->vertices[node->face->vertex[1]])+position;
                tri.c3 = *(obj->vertices[node->face->vertex[2]])+position;
                temp = tri.Intersect(ray);
                if ((temp >= 0) && (temp < ray->dist)) {
                    hit = true;
                    ray->hit_tri = node->face;
                }
            }
        }
    }
}

```

```

        ray->dist = temp;
    }
}
}
if (!node->inactive || !optimize) {
    // Ray-plane intersection
    if (!hit) {
        if (ray->pos[node->axis] <= node->face->max[node-
>axis]+position[node->axis]) { // Left
            this->Intersect(ray, node->left, hit, optimize);
            if (!hit && ray->dir[node->axis] >= -0.01) { // Right
                this->Intersect(ray, node->right, hit, optimize);
            }
        }
        else { // Right
            this->Intersect(ray, node->right, hit, optimize);
            if (!hit && ray->dir[node->axis] <= 0.01) { // Left
                this->Intersect(ray, node->left, hit, optimize);
            }
        }
    }
}
}
}

void KD_Tree::Delete(KD_Node *node) {
    if (node != 0) {
        this->Delete(node->left);
        this->Delete(node->right);
        delete node;
        node = 0;
    }
}
}

```

Code 28: KDTree.cpp

```

#ifndef LIGHT_H
#define LIGHT_H

#include "Vector3D.h"

class Light {
public:
    Light();
    ~Light();
public:
    Vector3D pos;
    Vector3D color;
};

```

```
#endif // LIGHT_H
```

Code 29: Light.h

```
#include "RayTracer.h"
```

```
int main() {  
    RayTracer r;  
    r.Initialize();  
    r.Render();  
    r.Clean();  
    return 0;  
}
```

Code 30: Main.cpp

```
#ifndef MESH_H  
#define MESH_H
```

```
#include <vector>
```

```
#include "Vector3D.h"
```

```
#define EPSILON 0.001
```

```
struct Face {  
    unsigned int vertex[3];  
    unsigned int texcoord[3];  
    unsigned int normal[3];  
    Vector3D max, min;  
};
```

```
class Mesh {  
public:  
    Mesh();  
    ~Mesh();  
  
    void CalculateMaxMin();  
    void CalculateNormals();  
public:  
    std::vector<Vector3D *> vertices;  
    std::vector<Face *> faces;  
    std::vector<Vector3D *> normals;  
    std::vector<Vector3D *> texcoords;  
    std::vector<Vector3D *> centers;  
    Vector3D position;  
    Vector3D max, min, center;  
    double radius;
```

```

    Vector3D pigment;
    double ambient, diffuse, specular;
    double reflection, refraction;
    double roughness;
    double ior;
};

#endif // MESH_H

```

Code 31: Mesh.h

```

#include <iostream>

#include "Mesh.h"

Mesh::Mesh() {
    vertices.clear();
    faces.clear();
    normals.clear();
    texcoords.clear();
    position.x = 0;
    position.y = 0;
    position.z = 0;
}

Mesh::~Mesh() {
    unsigned int num, i;
    num = vertices.size();
    for (i = 0; i < num; i++) {
        if (vertices[i]) {
            delete vertices[i];
            vertices[i] = 0;
        }
    }
    num = faces.size();
    for (i = 0; i < num; i++) {
        if (faces[i]) {
            delete faces[i];
            faces[i] = 0;
        }
    }
    num = normals.size();
    for (i = 0; i < num; i++) {
        if (normals[i]) {
            delete normals[i];
            normals[i] = 0;
        }
    }
    num = texcoords.size();
    for (i = 0; i < num; i++) {
        if (texcoords[i]) {
            delete texcoords[i];

```

```

        texcoords[i] = 0;
    }
}
num = centers.size();
for (i = 0; i < num; i++) {
    if (centers[i]) {
        delete centers[i];
        centers[i] = 0;
    }
}
}

void Mesh::CalculateMaxMin() {
    Face *f;
    Vector3D mesh_max, mesh_min;
    unsigned int num_faces;
    unsigned int i, v;

    num_faces = faces.size();
    mesh_max = *(vertices[faces[0]->vertex[0]]);
    mesh_min = *(vertices[faces[0]->vertex[0]]);
    for (i = 0; i < num_faces; i++) {
        f = faces[i];
        f->max = *(vertices[f->vertex[0]]);
        f->min = *(vertices[f->vertex[0]]);

        // Face max min
        for (v = 1; v <= 2; v++) {
            if ((vertices[f->vertex[v]]->x > f->max.x) {
                f->max.x = (vertices[f->vertex[v]]->x;
            }
            if ((vertices[f->vertex[v]]->x < f->min.x) {
                f->min.x = (vertices[f->vertex[v]]->x;
            }
            if ((vertices[f->vertex[v]]->y > f->max.y) {
                f->max.y = (vertices[f->vertex[v]]->y;
            }
            if ((vertices[f->vertex[v]]->y < f->min.y) {
                f->min.y = (vertices[f->vertex[v]]->y;
            }
            if ((vertices[f->vertex[v]]->z > f->max.z) {
                f->max.z = (vertices[f->vertex[v]]->z;
            }
            if ((vertices[f->vertex[v]]->z < f->min.z) {
                f->min.z = (vertices[f->vertex[v]]->z;
            }
        }
    }

    // Mesh max min
    if (f->max.x > mesh_max.x) {
        mesh_max.x = f->max.x;
    }
    if (f->min.x < mesh_min.x) {
        mesh_min.x = f->min.x;
    }
}

```

```

    if (f->max.y > mesh_max.y) {
        mesh_max.y = f->max.y;
    }
    if (f->min.y < mesh_min.y) {
        mesh_min.y = f->min.y;
    }
    if (f->max.z > mesh_max.z) {
        mesh_max.z = f->max.z;
    }
    if (f->min.z < mesh_min.z) {
        mesh_min.z = f->min.z;
    }
}
center = (mesh_max + mesh_min)/2;
max = mesh_max - center;
min = mesh_min - center;
radius = max.x;
if (max.y > radius) {
    radius = max.y;
}
if (max.z > radius) {
    radius = max.z;
}
}

void Mesh::CalculateNormals() {
    Face *f;
    Vector3D a, b, c, *d;
    unsigned int num;
    unsigned int i;

    // Delete existing normals
    num = normals.size();
    for (i = 0; i < num; i++) {
        if (normals[i]) {
            delete normals[i];
            normals[i] = 0;
        }
    }
    normals.clear();

    num = vertices.size();
    for (i = 0; i < num; i++) {
        d = new Vector3D();
        normals.push_back(d);
    }

    num = faces.size();
    for (i = 0; i < num; i++) {
        f = faces[i];
        a = *vertices[f->vertex[2]]-*vertices[f->vertex[0]];
        b = *vertices[f->vertex[1]]-*vertices[f->vertex[0]];
        c = b.Cross(a);

        *normals[f->vertex[0]] += c;
    }
}

```

```

        *normals[f->vertex[1]] += c;
        *normals[f->vertex[2]] += c;

        f->normal[0] = f->vertex[0];
        f->normal[1] = f->vertex[1];
        f->normal[2] = f->vertex[2];
    }

    num = normals.size();
    for (i = 0; i < num; i++) {
        normals[i]->Normalize();
    }
}

```

Code 32: Mesh.cpp

```

#ifndef OBJ_H
#define OBJ_H

#include "Mesh.h"

class OBJ : public Mesh {
public:
    OBJ();
    ~OBJ();

    bool Open(const char *filename);
};

#endif // OBJ_H

```

Code 33: OBJ.h

```

#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

#include "OBJ.h"

OBJ::OBJ() {
}

OBJ::~~OBJ() {
}

bool OBJ::Open(const char *filename) {
    std::ifstream file;
    std::stringstream buffer, temp;
    std::string token;
}

```

```

Vector3D *v;
Face *f;
bool has_texture;
char c;
int s, t, num_divs;
int i;

// Open file
file.open(filename);
if (!file.good()) {
    std::cout << "Error: Unable to open " << filename << std::endl;
}

// Read file to string stream
buffer << file.rdbuf();

// Close file
file.close();

// Parse file
while (buffer.good()) {
    buffer >> token;

    // Handle vertices
    if (token.compare("v") == 0) {
        v = new Vector3D();
        buffer >> v->x >> v->y >> v->z;
        vertices.push_back(v);
    }

    // Handle texture coordinates
    else if (token.compare("vt") == 0) {
        v = new Vector3D();
        buffer >> v->x >> v->y;
        texcoords.push_back(v);
    }

    // Handle normals
    else if (token.compare("vn") == 0) {
        v = new Vector3D();
        buffer >> v->x >> v->y >> v->z;
        normals.push_back(v);
    }

    // Handle faces
    else if (token.compare("f") == 0) {
        f = new Face();
        for (i = 0; i < 3; i++) {
            // Count number of divisions
            temp.clear();
            buffer >> token;
            temp << token;
            num_divs = -1;
            has_texture = true;
            for (s = 0, t = 0; t != std::string::npos; s = t) {

```

```

        t = token.find('/', s+1);
        if ((s != 0) && (t == s+1)) {
            has_texture = false;
        }
        num_divs++;
    }

    if (num_divs == 0) {
        temp >> f->vertex[i];
    }
    else if (num_divs == 1) {
        temp >> f->vertex[i] >> c >> f->texcoord[i];
        f->texcoord[i] -= 1;
    }
    else if (num_divs == 2) {
        if (has_texture) {
            temp >> f->vertex[i] >> c >> f->texcoord[i] >> c >> f-
>normal[i];
            f->texcoord[i] -= 1;
        }
        else {
            temp >> f->vertex[i] >> c >> c >> f->normal[i];
        }
        f->normal[i] -= 1;
    }
    f->vertex[i] -= 1;
}

        faces.push_back(f);
    }

    // Handle comments
    else if (token.compare("#") == 0) {
        // Do nothing
    }

    // Handle unknown
    else {
        // Do nothing
    }
}

return true;
}

```

Code 34: OBJ.cpp

```

#ifndef RAY_H
#define RAY_H

#include "Mesh.h"
#include "Vector3D.h"

```

```

class Ray {
public:
    Ray();
    ~Ray();
public:
    Vector3D pos, dir;
    unsigned int pixel;
    double dist;
    unsigned int hit_obj;
    Face *hit_tri;
};

#endif // RAY_H

```

Code 35: Ray.h

```

#include "Ray.h"

Ray::Ray() {
    pixel = 0;
    dist = 0.0;
    hit_obj = 0;
    hit_tri = 0;
}

Ray::~Ray() {
}

```

Code 36: Ray.cpp

```

#ifndef RAYTRACER_H
#define RAYTRACER_H

#include <iostream>
#include <list>
#include <vector>

#include "Camera.h"
#include "KD_Tree.h"
#include "Light.h"
#include "OBJ.h"
#include "Ray.h"
#include "TGA.h"
#include "Triangle.h"
#include "Vector3D.h"

#define SCREEN_WIDTH 200
#define SCREEN_HEIGHT 200

```

```

#define FAR_PLANE 10000.0

class RayTracer {
public:
    RayTracer();
    ~RayTracer();

    bool Initialize();
    bool Load(const char *filename);
    void Render();
    void Clean();
private:
    void Cast();
    bool Intersect(Ray *ray, bool optimize);
    void Lighting(Ray *ray, Vector3D &color);
    void Output();
private:
    Camera camera;
    TGA frame;

    std::list<Ray> cast_rays;
    std::list<Ray> light_rays;
    std::vector<Mesh *> objects;
    std::vector<Light> lights;

    std::vector<KD_Tree *> kdttrees;
};

#endif // RAYTRACER_H

```

Code 37: RayTracer.h

```

#include "RayTracer.h"

RayTracer::RayTracer() : camera(SCREEN_WIDTH, SCREEN_HEIGHT),
frame(SCREEN_WIDTH, SCREEN_HEIGHT) {
    Light l;

    // Set camera
    camera.position.x = -2.75;
    camera.position.y = 2.0;
    camera.position.z = 3.0;
    camera.lookat.x = -1.25;
    camera.lookat.y = 1.5;
    camera.lookat.z = -2.0;

    // Set lights
    l.color.x = 1.0;
    l.color.y = 1.0;
    l.color.z = 1.0;
    l.pos.x = -20.0;

```

```

    l.pos.y = 20.0;
    l.pos.z = 20.0;
    lights.push_back(l);
}

RayTracer::~RayTracer() {
    std::vector<Mesh *>::iterator it;
    for (it = objects.begin(); it != objects.end(); it++) {
        if (*it) {
            delete *it;
        }
    }
}

bool RayTracer::Initialize() {
    bool result;

    result = this->load("bunny.obj");
    objects[0]->position.x = 0.0;
    objects[0]->position.y = 0.0;
    objects[0]->position.z = 0.0;
    kdrees[0]->position = objects[0]->position;

    objects[0]->pigment.x = 0.8;
    objects[0]->pigment.y = 0.1;
    objects[0]->pigment.z = 0.1;

    objects[0]->ambient = 0.5;
    objects[0]->diffuse = 0.9;
    objects[0]->specular = 0.1;
    objects[0]->reflection = 0.0;
    objects[0]->refraction = 0.0;
    objects[0]->roughness = 0.05;
    objects[0]->ior = 0.0;

    result = this->load("wall.obj");
    objects[1]->position.x = 0.0;
    objects[1]->position.y = 0.0;
    objects[1]->position.z = 0.0;
    kdrees[1]->position = objects[1]->position;

    objects[1]->pigment.x = 0.1;
    objects[1]->pigment.y = 0.2;
    objects[1]->pigment.z = 0.7;

    objects[1]->ambient = 0.5;
    objects[1]->diffuse = 0.9;
    objects[1]->specular = 0.1;
    objects[1]->reflection = 0.9;
    objects[1]->refraction = 0.0;
    objects[1]->roughness = 0.05;
    objects[1]->ior = 0.0;
    return result;
}

```

```

bool RayTracer::Load(const char *filename) {
    OBJ *obj;
    KD_Tree *kdtree;
    bool result;

    obj = new OBJ();

    // Load model
    result = obj->Open(filename);
    objects.push_back(obj);

    obj->CalculateNormals();
    obj->CalculateMaxMin();

    // Build K-D tree
    kdtree = new KD_Tree();
    kdtree->object = obj;
    kdtree->Build();
    kdtrees.push_back(kdtree);

    return result;
}

void RayTracer::Render() {
    Ray ray;
    bool hit;
    Vector3D color;
    Pixel pixel;
    int i;

    // Generate initial cast rays
    this->Cast();

    for (i = 0; i < kdtrees.size(); i++) {
        kdtrees[i]->Cull(&camera, kdtrees[i]->root);
    }

    // Test intersection for each ray
    while (!cast_rays.empty()) {
        // Get ray from list
        ray = cast_rays.front();
        cast_rays.pop_front();

        // Test intersection
        hit = this->Intersect(&ray, true);
        if (hit) {
            light_rays.push_back(ray);
        }
        else {
            frame.pixels[ray.pixel].r = 0;
            frame.pixels[ray.pixel].g = 0;
            frame.pixels[ray.pixel].b = 0;
        }
    }
}

```

```

// Calculate lighting for each light ray
while (!light_rays.empty()) {
    // Get ray from list
    ray = light_rays.front();
    light_rays.pop_front();

    // Calculate lighting
    this->Lighting(&ray, color);

    // Scale light contributions
    color *= 255;

    // Bound color values
    if (color.x < 0.0) {
        color.x = 0;
    }
    else if (color.x > 255.0) {
        color.x = 255;
    }
    if (color.y < 0.0) {
        color.y = 0;
    }
    else if (color.y > 255.0) {
        color.y = 255;
    }
    if (color.z < 0.0) {
        color.z = 0;
    }
    else if (color.z > 255.0) {
        color.z = 255;
    }

    // Set pixel color
    pixel.r = (unsigned char)color.x;
    pixel.g = (unsigned char)color.y;
    pixel.b = (unsigned char)color.z;
    frame.pixels[ray.pixel] = pixel;
}

// Output frame to file
this->Output();
}

void RayTracer::Clean() {
    int i;
    for (i = 0; i < kdtrees.size(); i++) {
        kdtrees[i]->Delete(kdtrees[i]->root);
    }
}

void RayTracer::Cast() {
    camera.GenerateRays(&cast_rays);
}

bool RayTracer::Intersect(Ray *ray, bool optimize) {

```

```

bool hit, h;
int i;

hit = false;
ray->dist = FAR_PLANE;

// Test ray against list of objects
for (i = 0; i < kdtrees.size(); i++) {
    h = false;
    kdtrees[i]->Intersect(ray, kdtrees[i]->root, h, optimize);
    if (h) {
        hit = true;
        ray->hit_obj = i;
    }
}

return hit;
}

void RayTracer::Lighting(Ray *ray, Vector3D &color) {
    Mesh *obj;
    Face *face;
    Light light;
    Ray light_ray, reflect_ray;
    Vector3D normal, reflect_color;
    Vector3D v, h;
    Vector3D n1, n2, n3;
    double dn1, svr;
    double dn;
    unsigned int num_lights;
    unsigned int i;
    int count;
    bool hit;

    num_lights = lights.size();

    // Get object
    obj = objects[ray->hit_obj];

    // Get object's face
    face = ray->hit_tri;

    // Get intersection point
    light_ray.pos = ray->pos + ray->dir * ray->dist;

    // Get object's normal
    n1 = *(obj->vertices[face->vertex[0]])+obj->position-light_ray.pos;
    n2 = *(obj->vertices[face->vertex[1]])+obj->position-light_ray.pos;
    n3 = *(obj->vertices[face->vertex[2]])+obj->position-light_ray.pos;
    normal = *(obj->normals[face->normal[0]])/n1.Length() + *(obj->normals[face->normal[1]])/n2.Length() + *(obj->normals[face->normal[2]])/n3.Length();
    normal.Normalize();

    // Calculate ambient color

```

```

color = obj->pigment * obj->ambient;

for (i = 0; i < num_lights; i++) {
    // Get light source
    light = lights[i];

    // Calculate light ray
    light_ray.dir = light.pos - light_ray.pos;
    light_ray.dist = light_ray.dir.Length();
    light_ray.dir.Normalize();

    // Intersection
    hit = this->Intersect(&light_ray, false);
    if (hit) { // No direct path to light (ie, in shadow)
    }
    else {
        // Calculate diffuse component
        dnl = normal.Dot(light_ray.dir);
        if (dnl < 0.0) {
            dnl = 0.0;
        }
        dnl *= obj->diffuse;

        // Calculate specular component
        v = light_ray.pos-camera.position;
        v.Normalize();
        h = v+light_ray.dir;
        h.Normalize();
        svr = h.Dot(normal);
        if (svr < 0.0) {
            svr = 0.0;
        }
        svr = obj->specular*pow(svr, 1.0/obj->roughness);

        // Calculate diffuse and specular color
        color.x += (dnl*obj->pigment.x*light.color.x +
svr*light.color.x)/num_lights;
        color.y += (dnl*obj->pigment.y*light.color.y +
svr*light.color.y)/num_lights;
        color.z += (dnl*obj->pigment.z*light.color.z +
svr*light.color.z)/num_lights;
    }
}

if (obj->refraction == 1.0) {
    // Calculate refraction
}
else if (obj->reflection != 0.0) {
    // Create reflection ray
    reflect_ray.dir = ray->dir - normal*2.0*ray->dir.Dot(normal);
    reflect_ray.dir.Normalize();
    reflect_ray.pos = light_ray.pos;
    hit = this->Intersect(&reflect_ray, false);
    if (hit) {
        this->Lighting(&reflect_ray, reflect_color);
    }
}

```

```

        color = color + reflect_color*obj->reflection;
    }
    else {
    }
}
}

void RayTracer::Output() {
    frame.Write("output");
}

```

Code 38: RayTracer.cpp

```

#ifndef TGA_H
#define TGA_H

struct Pixel {
    unsigned char r, g, b;
};

class TGA {
public:
    TGA(short width, short height);
    ~TGA();

    bool Write(const char *name);
public:
    // Header
    char IDLength;           // 0-255
    char ColorMapType;      // 0 = none
                            // 1 = has color map
    char ImageType;         // 0 = none
                            // 1 = uncompressed color-mapped image
                            // 2 = uncompressed true-color image
                            // 3 = uncompressed black-and-white image
                            // 9 = run-length encoded color map image
                            // 10 = run-length encoded true-color image
                            // 11 = run-length encoded black-and-white

image
    short ColorMapOffset;   // Offset to color map table
    short ColorMapLength;   // Number of entries
    char ColorMapEntrySize; // Number of bits per pixel for color map
    short ImageXOrigin;     // Absolute coordinate of origin
    short ImageYOrigin;     // Absolute coordinate of origin
    short ImageWidth;       // Width in pixels
    short ImageHeight;      // Height in pixels
    char PixelDepth;        // Number of bits per pixel for image
    char ImageDescriptor;   // Bit 3-0: alpha channel depth
                            // Targa 16 = 0
                            // Targa 24 = 0
                            // Targa 32 = 8
                            // Bit 4: reserved, must be 0

```

```

// Bit 5: screen origin
// 0 = origin in lower left-hand corner
// 1 = origin in upper left-hand corner
// Bit 7-6: data storage interleaving flag
// 00 = non-interleaved
// 01 = two-way (even/odd) interleaving
// 10 = four-way interleaving
// 11 = reserved

// Data
Pixel *pixels;
};

#endif // TGA_H

```

Code 39: TGA.h

```

#include <iostream>
#include <fstream>

#include "TGA.h"

TGA::TGA(short width, short height) {
    int size, i;

    IDLength = 0;
    ColorMapType = 0;
    ImageType = 2;
    ColorMapOffset = 0;
    ColorMapLength = 0;
    ColorMapEntrySize = 0;
    ImageXOrigin = 0;
    ImageYOrigin = 0;
    ImageWidth = width;
    ImageHeight = height;
    PixelDepth = 24;
    ImageDescriptor = 1<<5;

    size = width * height;
    pixels = new Pixel[size];
    for (i = 0; i < size; i++) {
        pixels[i].r = 0;
        pixels[i].g = 0;
        pixels[i].b = 0;
    }
}

TGA::~TGA() {
    if (pixels != 0) {
        delete[] pixels;
    }
}

```

```

bool TGA::Write(const char *name) {
    std::ofstream file;
    std::string filename;
    int size, i;

    // Create TGA file
    filename = name;
    filename += ".tga";
    file.open(filename, std::ios::binary);
    if (file.good() == false) {
        std::cout << "Error: Unable to open " << filename.c_str() << "." <<
std::endl;
        return false;
    }

    // Write header
    file << (char)IDLength <<
        (char)ColorMapType <<
        (char)ImageType <<
        (char)(ColorMapOffset&0x00FF) <<
        (char)((ColorMapOffset&0xFF00)/256) <<
        (char)(ColorMapLength&0x00FF) <<
        (char)((ColorMapLength&0xFF00)/256) <<
        (char)ColorMapEntrySize <<
        (char)(ImageXOrigin&0x00FF) <<
        (char)((ImageXOrigin&0xFF00)/256) <<
        (char)(ImageYOrigin&0x00FF) <<
        (char)((ImageYOrigin&0xFF00)/256) <<
        (char)(ImageWidth&0x00FF) <<
        (char)((ImageWidth&0xFF00)/256) <<
        (char)(ImageHeight&0x00FF) <<
        (char)((ImageHeight&0xFF00)/256) <<
        (char)PixelDepth <<
        (char)ImageDescriptor;

    // Write data
    size = ImageWidth * ImageHeight;
    for (i = 0; i < size; i++) {
        file << (char)(pixels[i]).b <<
            (char)(pixels[i]).g <<
            (char)(pixels[i]).r;
    }

    file.close();

    return true;
}

```

Code 40: TGA.cpp

```

#ifndef TRIANGLE_H
#define TRIANGLE_H

```

```

#include "Ray.h"
#include "Vector3D.h"

#define EPSILON 0.001

class Triangle {
public:
    Triangle();
    ~Triangle();

    double Intersect(Ray *ray);
public:
    Vector3D c1, c2, c3;
};

#endif // TRIANGLE_H

```

Code 41: Triangle.h

```

#include "Triangle.h"

Triangle::Triangle() {
}

Triangle::~Triangle() {
}

double Triangle::Intersect(Ray *ray) {
    double a, b, c, d, e, f, g, h, i, j, k, l, M;
    double xa, xb, xc, xd, xe, ya, yb, yc, yd, ye, za, zb, zc, zd, ze;
    double eihf, gfdi, dheg, akjb, jcal, blkc;
    double beta, gamma, t;

    xa = c1.x;
    xb = c2.x;
    xc = c3.x;
    xd = (ray->dir).x;
    xe = (ray->pos).x;
    ya = c1.y;
    yb = c2.y;
    yc = c3.y;
    yd = (ray->dir).y;
    ye = (ray->pos).y;
    za = c1.z;
    zb = c2.z;
    zc = c3.z;
    zd = (ray->dir).z;
    ze = (ray->pos).z;
    a = xa-xb;

```

```

b = ya-yb;
c = za-zb;
d = xa-xc;
e = ya-yc;
f = za-zc;
g = xd;
h = yd;
i = zd;
j = xa-xe;
k = ya-ye;
l = za-ze;
eihf = e*i-h*f;
gfdi = g*f-d*i;
dheg = d*h-e*g;
akjb = a*k-j*b;
jcal = j*c-a*l;
blkc = b*l-k*c;
M = a*(eihf)+b*(gfdi)+c*(dheg);

t = -(f*(akjb)+e*(jcal)+d*(blkc))/M;
if (t < EPSILON) {
    return -1.0;
}

gamma = (i*(akjb)+h*(jcal)+g*(blkc))/M;
if ((gamma < 0.0) || (gamma > 1.0)) {
    return -1.0;
}

beta = (j*(eihf)+k*(gfdi)+l*(dheg))/M;
if ((beta < 0) || (beta > (1-gamma))) {
    return -1.0;
}
return t;
}

```

Code 42: Triangle.cpp

```

#ifndef VECTOR3D_H
#define VECTOR3D_H

class Vector3D {
public:
    Vector3D();
    Vector3D(double x, double y, double z);
    ~Vector3D();

    double operator[](int i);

    Vector3D & operator=(const Vector3D &v);
    Vector3D operator+(const Vector3D &v) const;
    Vector3D operator-(const Vector3D &v) const;
    Vector3D operator*(double s) const;

```

```

    Vector3D operator/(double s) const;
    void operator+=(const Vector3D &v);
    void operator-=(const Vector3D &v);
    void operator*=(double s);
    void operator/=(double s);

    void Clear();
    Vector3D Cross(const Vector3D &v);
    double Dot(const Vector3D &v);
    void Normalize();
    double Length();
public:
    double x, y, z;
};

#endif // VECTOR3D_H

```

Code 43: Vector3D.h

```

#include <math.h>

#include "Vector3D.h"

Vector3D::Vector3D() {
    x = 0.0;
    y = 0.0;
    z = 0.0;
}

Vector3D::Vector3D(double x, double y, double z) {
    this->x = x;
    this->y = y;
    this->z = z;
}

Vector3D::~Vector3D() {
}

double Vector3D::operator[](int i) {
    if (i == 0) {
        return x;
    }
    else if (i == 1) {
        return y;
    }
    else if (i == 2) {
        return z;
    }
    else {
        return 0;
    }
}

```

```

}

Vector3D & Vector3D::operator=(const Vector3D &v) {
    x = v.x;
    y = v.y;
    z = v.z;
    return *this;
}

Vector3D Vector3D::operator+(const Vector3D &v) const {
    Vector3D o;
    o.x = x+v.x;
    o.y = y+v.y;
    o.z = z+v.z;
    return o;
}

Vector3D Vector3D::operator-(const Vector3D &v) const {
    Vector3D o;
    o.x = x-v.x;
    o.y = y-v.y;
    o.z = z-v.z;
    return o;
}

Vector3D Vector3D::operator*(double s) const {
    Vector3D o;
    o.x = x*s;
    o.y = y*s;
    o.z = z*s;
    return o;
}

Vector3D Vector3D::operator/(double s) const {
    Vector3D o;
    o.x = x/s;
    o.y = y/s;
    o.z = z/s;
    return o;
}

void Vector3D::operator+=(const Vector3D &v) {
    x += v.x;
    y += v.y;
    z += v.z;
}

void Vector3D::operator-=(const Vector3D &v) {
    x -= v.x;
    y -= v.y;
    z -= v.z;
}

void Vector3D::operator*=(double s) {
    x *= s;
}

```

```

    y *= s;
    z *= s;
}

void Vector3D::operator/=(double s) {
    x /= s;
    y /= s;
    z /= s;
}

void Vector3D::Clear() {
    x = 0.0;
    y = 0.0;
    z = 0.0;
}

Vector3D Vector3D::Cross(const Vector3D &v) {
    Vector3D o;
    o.x = y*v.z-z*v.y;
    o.y = z*v.x-x*v.z;
    o.z = x*v.y-y*v.x;
    return o;
}

double Vector3D::Dot(const Vector3D &v) {
    double o;
    o = x*v.x + y*v.y + z*v.z;
    return o;
}

void Vector3D::Normalize() {
    double length = this->Length();
    x /= length;
    y /= length;
    z /= length;
}

double Vector3D::Length() {
    return sqrt(x*x + y*y + z*z);
}

```

Code 44: Vector3D.cpp

7. References

- [1] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, vol. 18, no. 9, 1975.
- [2] D. Fussell and K. R. Subramanian, "Faster Ray Tracing using K-D Trees," 1988.
- [3] J. Amanatides, "Ray Tracing with Cones".
- [4] S. Boulos, I. Wald and P. Shirley, "Geometric and Arithmetic Culling Methods for Entire Ray Packets".
- [5] K. Dmitriev, V. Harvan and H. Seidel, "Faster Ray Tracing with SIMD Shaft Culling," 2004.
- [6] K. Komatsu, Y. Kaeriyama, K. Suzuki, H. Takizawa and H. Kobayashi, "A Fast Ray Frustum-Triangle Intersection Algorithm with Precomputation and Early Termination," *IPSSJ*, vol. 1, pp. 1-11, 2008.
- [7] A. Reshetov, A. Soupikov and J. Hurley, "Multi-Level Ray Tracing Algorithm".
- [8] S. Teller and J. Alex, "Frustum Casting for Progressive, Interactive Rendering," 1998.
- [9] J. Amanatides and K. Choi, "Ray Tracing Triangular Meshes".
- [10] U. Assarsson and T. Moller, "Optimized View Frustum Culling Algorithms for Bounding Boxes," 2000.
- [11] M. S. Sunar, A. M. Zin and T. M. T. Sembok, "Improved View Frustum Culling Technique for Real-Time Virtual Heritage Application," *The International Journal of Virtual Reality*, vol. 7(3), pp. 43-48, 2008.
- [12] T. Moller and B. Trumbore, "Fast, Minimum Storage Ray/Triangle Intersection," *Graphics Tools*, pp. 21-28, 1997.
- [13] J. Delfosse, W. T. Hewitt and M. Meriaux, "An Investigation of Discrete Ray-Tracing," *4th Discrete Geometry in Computer Imagery Conference, Grenoble*, pp. 65-76, 1994.

- [14] N. Stolte and R. Caubet, "Discrete Ray-Tracing of Huge Voxel Spaces".
- [15] R. Yagel, D. Cohen and A. Kaufman, "Discrete Ray Tracing," 1992.
- [16] P. Shirley, M. Ashikhmin, M. Gleicher, S. Marschner, E. Reinhard, K. Sung, W. Thompson and P. Willemsen, *Fundamentals of Computer Graphics*, Second Edition, A K Peters, Ltd., 2005.
- [17] B. T. Phong, "Illumination for Computer Generated Pictures," *Communications of ACM* 18, vol. 6, pp. 311-317, 1975.
- [18] J. M. Snyder and A. H. Barr, "Ray Tracing Complex Models containing Surface Tesselations," *Computer Graphics*, vol. 21, no. 4, pp. 119-128, 1987.