

Ray-Casting

CS500 – Project1

Rahil Momin

Contents:

Libraries	2
Ray – Casting	2
Intersection Detection	2
BVH Tree	3
Image Processing	4

Libraries:

Apart from the windows libraries, Eigen was used as a math library and OpenGL and glut as real time rendering API.

Ray – Casting:

Ray's are casted for each pixel on the screen and the ray's are casted through the center of said pixel; that said each ray passes through the center of a pixel, but they originate at the camera, also called "eye". To cast ray in view space each ray is transformed through the orientation of the camera.

Each ray has two components, pos and dir. The position is the camera position, the direction is calculated as following

```
Vector3f X = raytrace->rx * raytrace->orient._transformVector(Vector3f::UnitX());
Vector3f Y = raytrace->ry * raytrace->orient._transformVector(Vector3f::UnitY());
Vector3f Z = -1.0f * raytrace->orient._transformVector(Vector3f::UnitZ());

float dx = (2.0f * (x + 0.5f) / width) - 1.0f;
float dy = (2.0f * (y + 0.5f) / height) - 1.0f;

Vector3f dir = dx * X + dy * Y + Z;
```

The X, Y, Z are the oriented axis where ry is given and rx is calculated as

$rx = ry \times \frac{width}{height}$; width and height are the width and height of the screen. Indeed, the dir needs to be normalized.

Intersection Detection:

There are four basic shapes ray intersection implementation. The shapes include sphere, box, cylinder and triangle. Once the intersection test has passed the information that is collected are the length of intersection along the ray (t), point of intersection (p), shape collided with (object) and the normal of collision.

For meshes that do not belong to basic shapes, the meshes are treated as a group of triangles and each triangle in the mesh is considered a different shape. Each shape includes a Bounding Volume (Bbox) which is a box and encapsulates the entire shape.

each bounding volume is calculated as min and max points of the box.

Spheres:

```
Bbox(C - Vector3f(r, r, r), C + Vector3f(r, r, r));
```

Where C is the center and r are the radius.

Box:

```
Bbox(base, base + diag);
```

Where base is the min point of the box and diag the diagonal vector of the box.

Cylinder:

```
Vector3f P0 = B + Vector3f(r, r, r);
Vector3f P1 = B - Vector3f(r, r, r);
Vector3f P2 = A + B + Vector3f(r, r, r);
Vector3f P3 = A + B - Vector3f(r, r, r);

Vector3f minPoint;
minPoint[0] = std::fmin(std::fmin(P0[0], P1[0]), std::fmin(P2[0], P3[0]));
minPoint[1] = std::fmin(std::fmin(P0[1], P1[1]), std::fmin(P2[1], P3[1]));
minPoint[2] = std::fmin(std::fmin(P0[2], P1[2]), std::fmin(P2[2], P3[2]));

Vector3f maxPoint;
maxPoint[0] = std::fmax(std::fmax(P0[0], P1[0]), std::fmax(P2[0], P3[0]));
maxPoint[1] = std::fmax(std::fmax(P0[1], P1[1]), std::fmax(P2[1], P3[1]));
maxPoint[2] = std::fmax(std::fmax(P0[2], P1[2]), std::fmax(P2[2], P3[2]));

Bbox(minPoint, maxPoint);
```

Where B is the base point of the cylinder, r is the radius of the cylinder and A is the axis vector.

minPoint and maxPoint are calculated as the minimum and maximum of x, y and z components from all four points.

Triangle:

```
Vector3f minPoint;
minPoint[0] = std::fmin(std::fmin(V0[0], V1[0]), V2[0]);
minPoint[1] = std::fmin(std::fmin(V0[1], V1[1]), V2[1]);
minPoint[2] = std::fmin(std::fmin(V0[2], V1[2]), V2[2]);

Vector3f maxPoint;
maxPoint[0] = std::fmax(std::fmax(V0[0], V1[0]), V2[0]);
maxPoint[1] = std::fmax(std::fmax(V0[1], V1[1]), V2[1]);
maxPoint[2] = std::fmax(std::fmax(V0[2], V1[2]), V2[2]);

Bbox(minPoint, maxPoint);
```

Where V0, V1, V2 are the points of the triangle. minPoint and maxPoint are calculated as the minimum and maximum of x, y and z components from all three points.

BVH Tree:

All shapes are added to a Bounding volume hierarchy (BVH). The BVH is imported from Eigen's "Eigen_unsupported/Eigen/BVH" header file. The BVH tree used is a KDTree provided as `KdBVH<float, 3, Shape*>`. A `Minimizer` class and a global function `Bbox` `bounding_box(const Shape* obj)` was provided in order for the Eigen library to build the KDTree. The speed up from

this implementation was timed for five passes; time was measured in milliseconds. The passes were performed under Release x86 configuration.

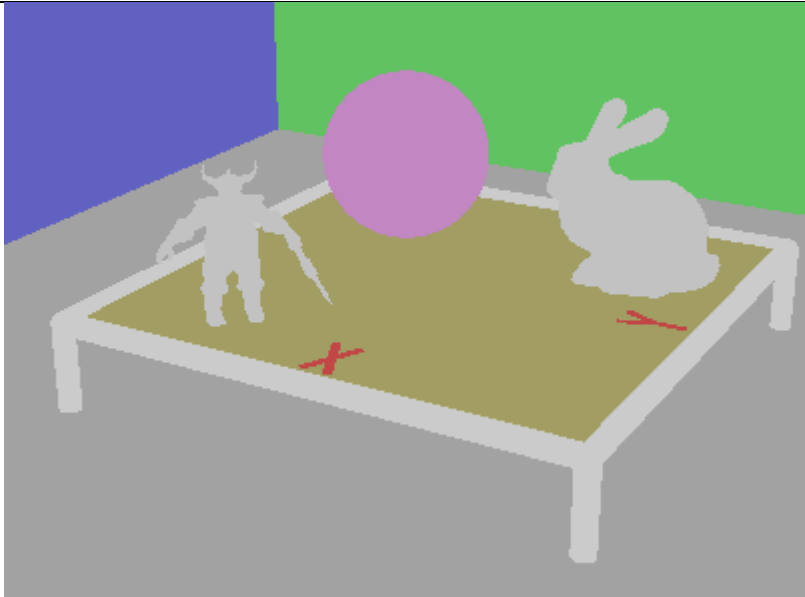
Pass	Linear Time	KDTree Time
1	20515	196
2	21252	153
3	22455	141
4	22393	126
5	22268	120
AVG	21776	147


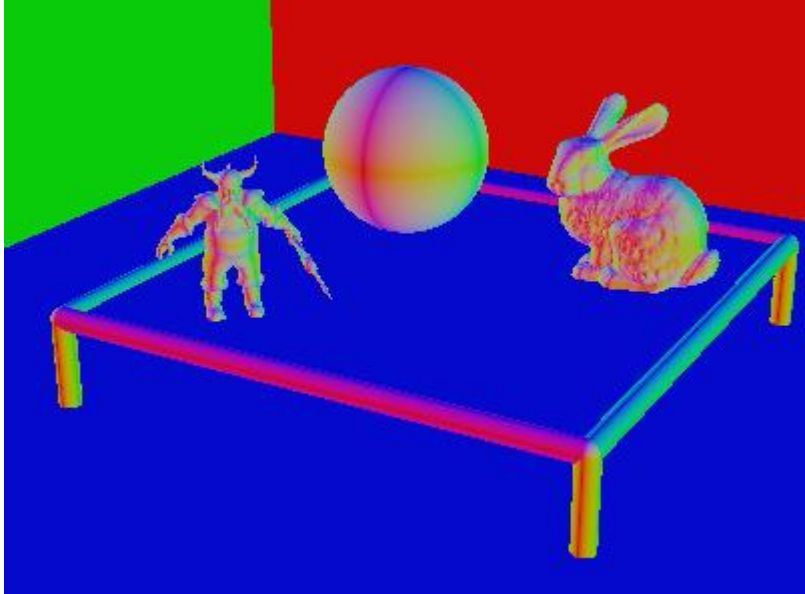
As seen from the times above by implementing a BVH Tree the ray-tracer receives a speedup by the factor of 148.

The KDTree only started to show effect after 31 objects that is when the objects “dwarf” and “bunny” were introduced in the scene. If the high poly objects were not in the scene both times, linear and Tree were averaged around 120 milliseconds.

Image Processing:

Three Images were generated from the ray-tracer representing three different calculations. All images are collected by ray-casting for each pixel, processing the ray through the BVH and collecting the data required to produce the image.

Image	Data
	This image represents the diffuse colour from the closest object that the ray intersected with.

	<p>This image represents the distance (t) of the closest intersected object from the camera. For the purposes of data collection the range is clamped to $[0,1]$ by using the grey scale value of $(t-5)/4$</p>
	<p>This image represents the normal (N) of the closest intersected object at the point of intersection from the camera.</p>