Advanced Technologies: Multi-threaded CPU based Ray-tracer

Alexander Stephen Allman
16015338

University of the West of England

April 29, 2020

The aim of this project was to create a CPU based Ray-tracer which demonstrated realistic rendering through direct lighting and various material types. This report summarises the techniques followed to create such a system whilst also utilising multi-threading and optimisation techniques like Bounding Volume Hierarchy to keep it as close to real time as possible.

1 Introduction

In recent years ray-tracing has become mainstream due to NVIDIA releasing their RTX line of GPUs focused on real time ray-tracing (Laguna, 2018). Prior to this rendering photo-realistic images via ray-tracing could take many minutes to days at a time per frame depending on the complexity and illumination of the scene (Seymour, 2014), and whether or not multiple machines and GPUs were utilised for rendering. This project will focus specifically on creating a CPU based ray-tracer. This allows for the core ray-tracer functionality to be built in an easier to debug domain and once confirmed to be functional, can later be adapted to make use of the GPU. The system will also make use of couple of performance enhancing features such as multi-threading and Bounding volume hierarchy to help close the performance gap whilst the ray-tracer is CPU based.

2 Related Work

Whilst real time ray-tracing is quite a recent achievement, ray-tracing as a concept has been around much longer. Whitted proposed the original ray-tracer (Whitted, 1980) which used the idea of firing rays from the viewport into the scene and following each rays reflections until it no longer hits objects in the scene. Calculating the colour of the respective pixel based upon the intensity calculated for every reflection. The intensity calculation was later iterated upon by Kajiya (Kajiya, 1986) and is known as the rendering equation. Pete Shirley also has a collection of books that explain the concept in an easy to digest manner and takes a developer step by step through creating their own ray-tracer (Shirley, 2016). His books were used in this project to help get a base understanding on the concepts of a ray-tracer before developing the project further.

3 Method

3.1 Vector Library

In order to do anything with rays and calculations related to them, a Vector library is required. Both Vector2 and Vector3 will be required for use in the larger project. Vector2's will be utilised when sampling from textures and Vector3 will be utilised for position and direction. Operator overrides for addition, subtraction, multiplication and division will also be very useful alongside some common vector math functions such as Dot product (EQ.1), Cross Product (EQ.2) and normalisation.

The equation for Dot product:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_n b_n \qquad (1)$$

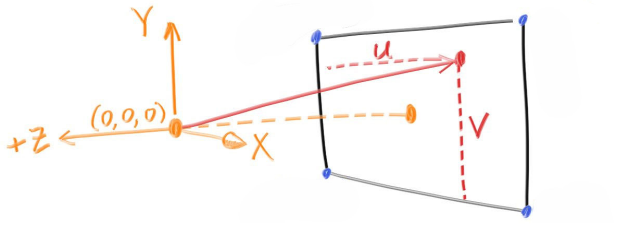


Figure 1: *Camera Visualisation (Shirley, 2016).*

The equation for Cross product:

$$a \times b = \|a\| \, \|b\| \sin(\theta) \; n \tag{2}$$

From here a ray can be constructed using a Vector3 for the origin and a Vector3 for the direction of the ray. This project also included the inverse direction of the ray as it is used in AABBs calculations.

3.2 Camera and Colouring Pixels

The camera is an important aspect of the ray-tracer as it translates screen space pixel positions into a normalised form for building the rays from. The cameras center (0,0,0) lies in the center of the window and follows the right handed coordinate convention; A visualisation of this can be seen in Figure 1.

The bounds of the camera are calculated in relation to the aspect ratio of the screen resolution. In this projects case for the resolution 1280x720 and an aspect ratio of 16:9 :

$$\begin{aligned} Range_x &= -8 \leq x \leq 8 \\ Range_y &= -4.5 \leq y \leq 4.5 \end{aligned} \tag{3}$$

With this information rays can be constructed roughly for the center of each pixel of the screen. Each rays start position starts at the camera center position in screen space and then the direction of the ray is calculated as follows.

Firstly, linear interpolation is used to calculate a ratio of how far along the current iteration is through the screens pixels as seen in EQ (4).

$$\begin{aligned} Ratio_x &= \frac{CurrentPixel_x}{Screen_{width}} \\ Ratio_y &= \frac{CurrentPixel_y}{Screen_{height}} \end{aligned} \tag{4}$$

The rays direction is then constructed by using this ratio to calculate an offset and adding it to the cameras lower bound (Shirley, 2016). This is seen in EQ (5) and EQ (6) respectively.

$$\begin{aligned} Offset_x &= Ratio_x \times AspectRatio_x \\ Offset_y &= Ratio_y \times AspectRatio_y \end{aligned} \tag{5}$$

$$\begin{aligned} RayDir_x &= LowerBound_x + Offset_x \\ RayDir_y &= LowerBound_y + Offset_y \end{aligned} \tag{6}$$

For example calculating the ray direction for pixel (480,160) for the above aspect ratio and resolution would go as followed:

$$\begin{aligned} Ratio &= (480/1280, 160/720) = (0.375, 0.\dot{2}) \\ Offset &= (0.375 \times 16, 0.\dot{2} \times 9) = (6, 2) \\ RayDir &= (-8+6, -4.5+2) = (-2, -2.5) \end{aligned} \tag{7}$$

Therefore the resultant Dir in a Vector3 would be (-2, -2.5, -1). The Z is always -1 as we want to fire the rays into the scene forwards not backwards behind the origin. The resultant ray created from this process will then be fed into the bounding volume hierarchy (BVH) to determine if the ray collides with any objects in the scene. If so the pixels colour is calculated from a combination of the objects material and the scenes lighting; otherwise a solid colour is used as a background fill.

3.3 Multi-Threading

Firing rays and calculating the colour of each pixel is a very costly process and is the one of the slowest parts of ray-tracing. In order to speed this up the project utilises multi-threading and specifically a thread pool. By using a thread pool the overhead of initialising threads every update is removed which is a significant negative; especially if many threads are used per update. In the project the colour data for each pixel is stored in a contiguous array and accessed through helper functions to keep a more co-ordinate like interaction.

```
void ColourPixelAtPosition(int& x, int& y, sf::Color col)
{
    _colours.at(y * _columns + x) = col;
}
```

Listing 1: *Changing the colour of a pixel at a given x and y position*

Once the thread pool has been established with a given thread count, a stride of the contiguous array is calculated as seen in Figure 8.

$$ArrayStride = \frac{ArraySize}{ThreadCount} \tag{8}$$

Then each thread is given a segment of the array to process based upon the stride. This allows multiple sets of pixels to be processed simultaneously across these threads.

```
for(i=0; i < threadCount; ++i)
{
    int startIndex = i * stride;
```

```
        int endIndex = (i + 1) * stride;
        endIndex = endIndex > pixelArray.Size() - 1 ? pixelArray.Size() - 1 : endIndex;

        threadPool.ProcessColours(startIndex, endIndex);
    }
```

Listing 2: *Splitting the colour calculation for each pixel amongst the thread pool*

3.4 Ray Intersections

In order to render objects and models in the scene the ray needs to be able to check if it has intersected them. If this check passes and it has intersected with an object then the pixel related to the ray can be coloured in relation to the object hit.

In all the following methods of calculating intersections the ray is described by the parametric equation found in EQ (9) where P is a point along the ray, O is the origin of the ray, t is a distance along the ray and D is the direction the ray is traveling.

$$P = O + tD \tag{9}$$

This project utilises three main primitive types of ray intersection detection:

Sphere Intersection: For sphere intersections the project utilises an analytic solution. Since spheres can be described in algebraic form as:

$$x^2 + y^2 + z^2 = R^2 \tag{10}$$

where x, y, z is a Cartesian point and R is the radius of the sphere. The Cartesian point can be then substituted for the rays point P (EQ (11)) and further developed in EQ (12) and EQ (13):

$$P^2 - R^2 = 0 \tag{11}$$

$$|O + tD|^2 - R^2 = 0 \tag{12}$$

$$O^2 + (Dt)^2 + 2ODt - R^2 = D^2t^2 + 2ODt - R^20 \tag{13}$$

EQ (13) can be described in the form of the quadratic equation:

$$f(x) = ax^2 + bx + c \tag{14}$$

where $a = D^2$, $b = 2OD$ and $c = O^2 - R^2$.

From here the discriminant can be used to calculate how many roots there are for a given intersection. t is then calculated via EQ (15) where Δ is the discriminant:

$$t = \frac{-b \pm \sqrt{\Delta}}{2a} \tag{15}$$

The lowest value for t above zero is then taken to calculate P from EQ (9). It must be above zero otherwise the sphere intersected with is behind the rays origin, not in front of it. The lowest value for t ensures that it is the intersection closest to the camera. A visualisation of this can be seen in Figure 2.

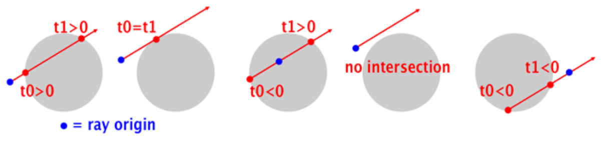


Figure 2: *Ray sphere intersection cases (Scratchapixel, 2014c).*

Box Intersection: For box intersections this project utilises the slab method combined with Axis Aligned Bounding Boxes. Since a AABB can be described as a set of lines parallel to each axis of the co-ordinate system an axis of each bound can be described in the form of the line equation:

$$y = BoxMinBound_x \tag{16}$$

Where y can be substituted for P from EQ (9):

$$O_x + tD_x = BoxMinBound_x \tag{17}$$

and then rearranged to find t:

$$tMin_x = (BoxMinBound_x - O_x)/D_x \tag{18}$$

This process is then followed for every axis comparing against the previous axis and choosing the greatest value for t:

```
 float tmin = (BoundMin.x - ray.orig.x) / r.dir.x;
 float tmax = (BoundMax.x - ray.orig.x) / r.dir.x;

 if (tmin > tmax) swap(tmin, tmax);

 float tymin = (min.y - r.orig.y) / r.dir.y;
 float tymax = (max.y - r.orig.y) / r.dir.y;

 if (tymin > tymax) swap(tymin, tymax);

 if ((tmin > tymax) || (tymin > tmax))
     return false;
```

```

    if (tymin > tmin)
        tmin = tymin;

    if (tymax < tmax)
        tmax = tymax;
```

Listing 3: *Calculating the tMin and tMax between the x and y axis. This is then repeated in the same manner for the z axis*

Triangle Intersection: For Triangle intersections the project utilises the Möller-Trumbore algorithm (Tomas Möller, 1997). This algorithm in addition to producing a point of collision, also returns Barycentric coordinates which can be used to interpolate between the three vertices of the triangle to get an exact point within the triangles bounds. These are utilised mainly in texturing to determine where to sample from a texture for a given pixel within the bounds of the vertices of the triangle.

This algorithm leverages the parameterization of P from EQ (9) in terms of Barycentric coordinates:

$$P = wA + uB + vC \tag{19}$$

where $w = 1 - u - v$ and A, B, C are the vertices of the triangle.

From here EQ (19) can be expanded by substituting P and w shown in EQ (20):

$$\begin{aligned} O + tD &= A + u(B - A) + v(C - A) \\ O - A &= -tD + u(B - A) + v(C - A) \end{aligned} \tag{20}$$

Since the three unknowns in this equation are u, v, t we treat t as the third axis of the u, v coordinate system which allows us to express the intersection P in terms of Barycentric coordinates. To find these unknowns Cramer's rule is utilised:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\left[|-D \quad E_1 \quad E_2|\right]} \begin{bmatrix} |T & E_1 & E_2| \\ |-D & T & E_2| \\ |-D & E_1 & T| \end{bmatrix} \tag{21}$$

where $T = O - A$, $E_1 = B - A$ and $E_2 = C - A$.

With knowing that $|ABC| = -(A \times C) \cdot B = -(C \times B) \cdot A$, EQ(21) can be written in terms of cross and dot products between known variables:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} \tag{22}$$

Which can be refactored into:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix} \tag{23}$$

Where $P = (D \times E_2)$ and $Q = (T \times E_1)$.

3.5 Bounding Volume Hierarchy and Surface Area Heuristic

Bounding Volume Hierarchy is a type of spacial data structure where each node in the tree is a large AABB volume containing progressively smaller volumes until it reaches the point of individual objects at the bottom of the tree. It is utilised in this project in order to speed up the detection of ray intersections. By grouping the objects in the scene into progressively larger bounding volumes; a large part of the white space in a scene is removed. In addition to this not all objects in the scene need to be tested against every single ray. The ray instead recursively goes through each layer of the hierarchy, cutting out a large amount of objects in the scene per layer due to how each layer is divided.

For each layer of the tree, Surface Area Heuristic is used to determine which axis is the best to divide on and at which point along that axis. This is described in EQ (24) where $C(A, B)$ is the cost of splitting a node into volumes A and B; $t_{traversal}$ is the time to traverse an interior node; p_A and p_B are the probabilities that the ray passes through volume A and B respectively; N_A and N_B are the number of triangles in volumes A and B respectively; a_i and b_i are the ith triangle in volumes and $t_{intersect}$ is the cost for one ray-triangle intersection (Wiche, 2018).

$$C(A,B) = t_{traversal} + p_A \sum_{i=1}^{N_A} t_{intersect}(a_i) + p_B \sum_{i=1}^{N_B} t_{intersect}(b_i) \tag{24}$$

This formula rewards the axis that packs the most triangles into the smallest area as the smaller the area, the less likely to get pierced by a ray. A visualisation of this is shown in Figure 3.

This method of constructing BVH's was also utilised on models in the project. As models are built of many triangles themselves; dividing those triangle into their own BVH in addition to the overarching scene BVH helps reduce the impact of the additional ray intersections checks. An example of how a ray would traverse through the bounding boxes in a BVH can be seen in Figure 4.

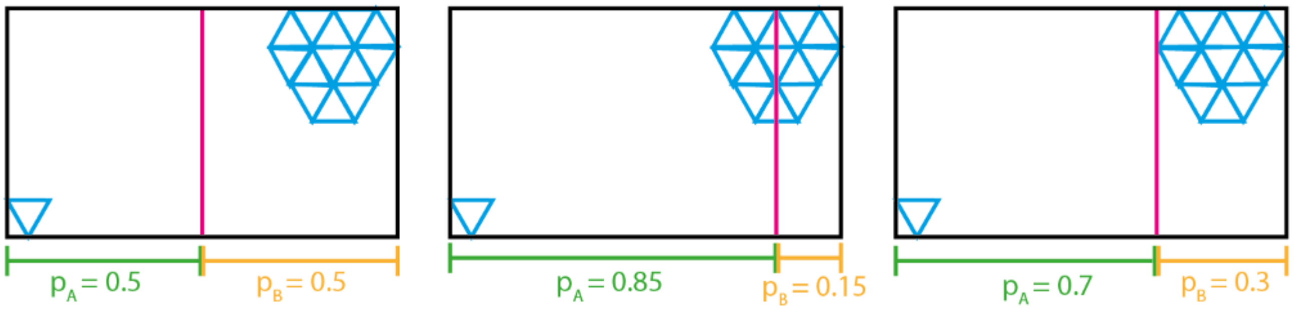
3.6 Barycentric Coordinates

As mentioned above Barycentric coordinates can be used to express any point within a triangle through using EQ (25):

$$P = uA + vB + wC \tag{25}$$

Where P is the point within the triangle; u, v and w are three scalars such that $u + v + w = 1$ and A, B, C are the three vertices of the triangle.

$$C(A,B) = t_{traversal} + p_A \sum_{i=1}^{N_A} t_{intersect}(a_i) + p_B \sum_{i=1}^{N_B} t_{intersect}(b_i)$$



$$1 + 0.5\sum_{i=1}^{1} 2 + 0.5\sum_{i=1}^{13} 2 = 15 \qquad 1 + 0.85\sum_{i=1}^{9} 2 + 0.15\sum_{i=1}^{8} 2 = 18.7 \qquad 1 + 0.7\sum_{i=1}^{1} 2 + 0.3\sum_{i=1}^{13} 2 = 10.2$$

Figure 3: *Visualisation of three different SAH costs (Wiche, 2018).*

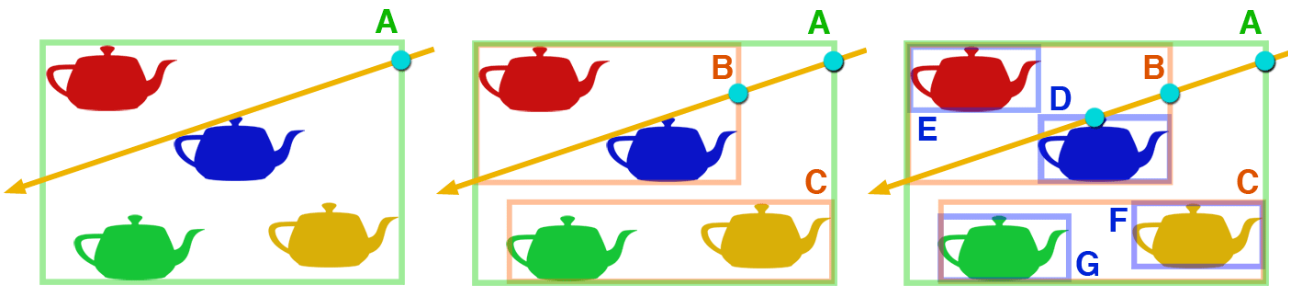


Figure 4: *Visualisation of BVH Traversal (Scratchapixel, 2014b).*

Through using this technique, position, texture coordinates and normals can be calculated for a given point in the triangle by substituting the appropriate vertex's information. For example to calculating the normal at a given point within a triangle can be seen in EQ (26):

$$P_{norm} = uVertA_{norm} + vVertB_{norm} + wVertC_{norm} \tag{26}$$

3.7 Texturing and Bilinear Interpolation

In this project texturing is done through utilising the Barycentric coordinate formula as mentioned above. This is done through calculating the normalised texture coordinates using EQ (25). When using this normalised texture coordinate for sampling from a texture, Bilinear interpolation is utilised.

Linear interpolation is defined with the function in EQ (27), where $0 \leq t \leq 1$ and a and b are two points to interpolate between:

$$P = a(1-t) + bt \tag{27}$$

Bilinear interpolation extends on this concept to return the product of two linear interpolations. As seen in Figure 5, a and b are both calculated using EQ (27) shown in EQ (28):

$$\begin{aligned} a &= c00 \times (1-t_x) + c10 \times t_x \\ b &= c01 \times (1-t_x) + c11 \times t_x \end{aligned} \tag{28}$$

Where t_x is the x component of the texture coordinate.

A linear interpolation is then used between a and b to find c:

$$c = a \times (1-t_y) + b \times t_y \tag{29}$$

Where t_y is the y component of the texture coordinate.

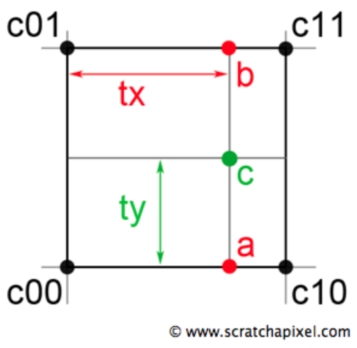


Figure 5: *Finding Point c within 2D boundaries (Scratchapixel, 2014a).*

3.8 Direct Lighting and Tone-mapping

This projects has two options for lighting the scene that's rendered. These being point lighting and area lighting. Each has its own particular features and disadvantages. All lighting types only calculate lighting for pixels in which their rays have intersected an object within the scene.

To simulate lighting within the ray-tracer a simplified version of the EQ (30) has been adopted. This is due to the integral $\int_\Omega \ldots d\omega_i$ being computationally impossible to solve; as the software would have to infinitely sample points within the hemisphere Ω centered on n. Since the materials used in this project do not emit light, the emitted spectral radiance $L_e(X, \omega_o)$ is also omitted from the lighting calculations used.

$$\begin{aligned} L_o(X,\omega_o) &= \\ L_e(X,\omega_o) &+ \\ \int_\Omega f_r(X,\omega_i,\omega_o)L_i(X,\omega_i)(\omega_i \cdot n)d\omega_i \end{aligned} \tag{30}$$

Where L_o is the resultant colour; L_e is the emitted colour of the object; f_r is the bidirectional reflectance distribution function; ω_i is the inverse direction of incoming light; L_i is the colour of the light directed at X; n is the surface normal at X and $(\omega_i \cdot n)$ is the weakening factor of outward irradiance due to the incident angle (Wolfe, 2016).

Point Lighting: Point lighting is the faster version of the two lighting methods due to only firing a single ray for the lighting calculation. This results in hard shadows as the objects in the scene are either occluded from the light, or not. The equation used for this can be seen in EQ (31):

$$\begin{aligned} L_o &= \\ \frac{n \cdot \omega_i}{|X\omega_i|} v(x \leftrightarrow x_i) L_i(X,\omega_i) f_r(X,\omega_i,\omega_o) \end{aligned} \tag{31}$$

Where $v(x \leftrightarrow x_i)$ is the visibility between the object and the light source.

Area Lighting: In order to gain photo-realistic soft shadows, the integral mentioned above would need to be solved. Since this is not possible a Monte Carlo approximation (Ghete, 2007) is utilised. As described in EQ (32) multiple rays are fired out at random points within the bounds of the area light towards X. An average of all the rays are then taken to determine the final colour. Due to processing multiple rays per light calculation this is significantly more costly. This cost only grows when the sample count is increased to decrease noise from the Monte Carlo approximation, as seen in Figure 9.

$$L_o = \frac{1}{N}\sum_{i=0}^{N} \frac{\frac{n\cdot\omega_i \times n_l\cdot -\omega_i}{|X\omega_i|} v(x \leftrightarrow x_i) L_i(X,\omega_i) f_r(X,\omega_i,\omega_o)}{pdf(x_i)} \tag{32}$$

Where N is the number of samples and $pdf(x_i)$ is $\frac{1}{LightArea}$.

Tone-mapping: Once the final solution has been calculated for a given pixel using the above lighting equations, the value must be tone mapped from HDR to SDR. In this project a basic Gamma tone-mapper is used, seen in EQ (33) then the SDR output is applied to the given pixel.

$$SDR = min(HDR^{\frac{1}{2.2}}, 1) \tag{33}$$

3.9 Materials

The materials play a factor in the final colour that comes from the scenes lighting, specifically materials determine the $f_r(X, \omega_i, \omega_o)$ components of EQs (30, 31, 32). This project utilises two different material types:

Diffuse: For the Diffuse material the colour given to the object is simply taken and divided by PI. This value is capped at 1 as the material cannot reflect more light than it receives. This can be seen in EQ (34)

$$C_{diffuse} = min(\frac{C_{mat}}{\pi}, 1) \tag{34}$$

Mirror: In order for mirror to determine what is opposite it, an extra ray is used. This ray is fired at a normal of the object hit and its origin is at X. If this secondary ray intersects another object, the lighting calculation is done again for the second object and the resultant colour is given back to the mirror and applied. If the object the mirror material is attached to has a colour of its own; it is added as a tint to the previous resultant colour with its intensity dependant on the colours alpha value. An example of the mirror material can be seen in Figures 7 and 8.

4 Evaluation

Both Multi-threading and utilising a BVH helps a lot with increasing performance. In Figure 6, we can see that utilising BVH and Multi-threading alone both help performance and when using both in conjunction the delta only grows further. These benchmarks are from a scene with three models comprised of 458 Tris in total and is lit with a single point light. The average Frame Time for each entry on the graph in Figure 6 are as follows:

- No BVH, No Threading: 6,911ms
- Only BVH: 1824ms
- Only Threading: 722ms
- BVH and Threading: 206ms

Some examples of scenes in the Ray-tracer can be seen below in Figures 7, 8 and 9:

5 Conclusion

In conclusion through utilising the outlined techniques above a Ray-tracer has been created that can use either primitives or models in a scene where the colour of every pixel is calculated based upon the collisions of the ray fired from that pixel, material properties and scene lighting.

Bibliography

Ghete, Mihai Calin (2007). "Mathematical Basics of Monte Carlo Rendering Algorithms". In: *Vienna University of Technology*. URL: https://www.cg.tuwien.ac.at/courses/Seminar/SS2007/MontecarloRendering-Ghete.pdf.

Kajiya, James T. (1986). "THE RENDERING EQUATION". In: *Siggraph*. Vol. 20. 4, 143–150.

Laguna, Gail (2018). "NVIDIA Unveils Quadro RTX, World's First Ray-Tracing GPU". In: URL: https://nvidianews.nvidia.com/news/nvidia-unveils-quadro-rtx-worlds-first-ray-tracing-gpu.

Scratchapixel (2014a). "Bilinear Interpolation". In: URL: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/interpolation/bilinear-filtering.

– (2014b). "Introduction to Acceleration Structures". In: URL: https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part2.

– (2014c). "Ray-Sphere Intersection". In: URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection.

Seymour, Mike (2014). "Disney's new Production Renderer 'Hyperion' – Yes, Disney!" In: URL: https://www.fxguide.com/fxfeatured/disneys-new-production-renderer-hyperion-yes-disney.

Shirley, Pete (2016). "Ray Tracing in One Weekend". In:

Tomas Möller, Ben Trumbore (1997). "Fast, Minimum Storage Ray-Triangle Intersection". In: *Journal of Graphics Tools*. Vol. 2. 1, 21–28.

Whitted, Turner (1980). "An Improved Illumination Model for Shaded Display". In: *Communications of the ACM*. Vol. 23. 6, 343–349.

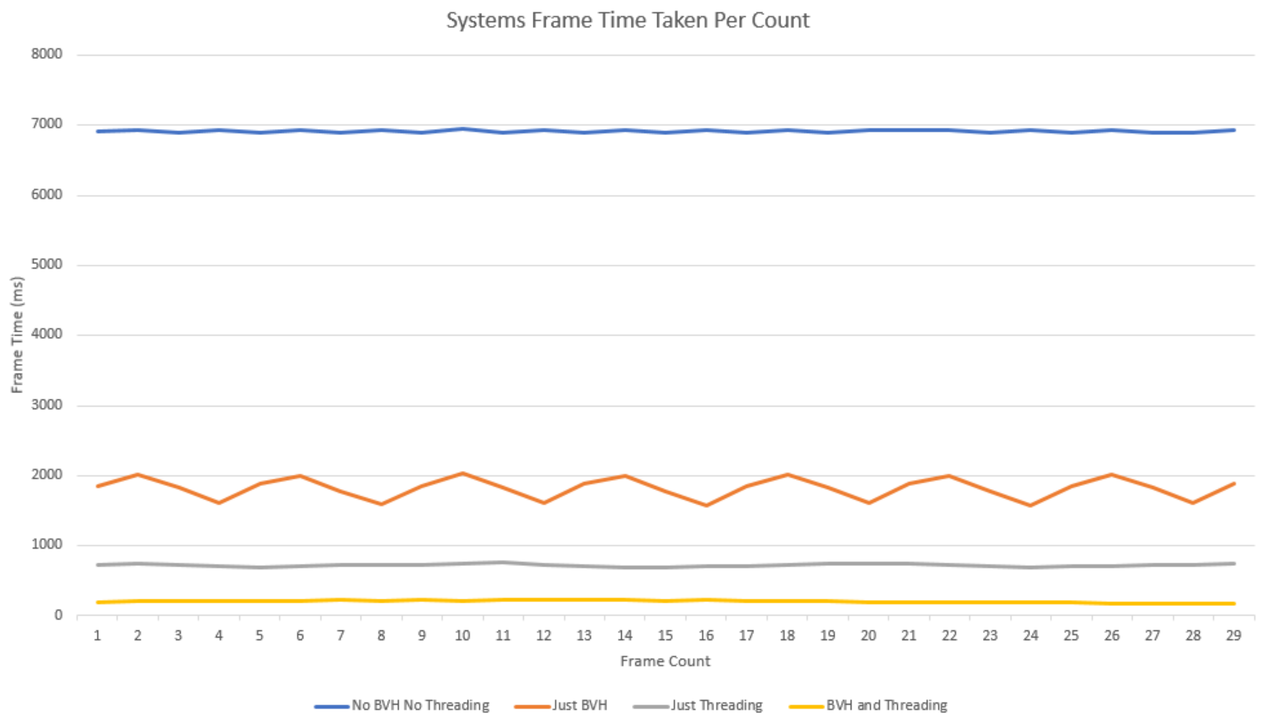


Figure 6: *Graph showing each systems average performance.*

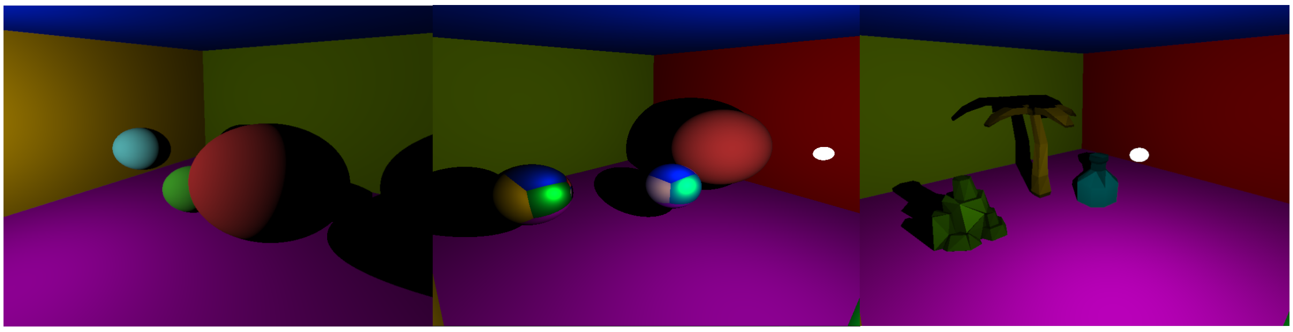


Figure 7: *Various scenes lit with a point light.*

Wiche, Roman (2018). “How to create awesome accelerators: The Surface Area Heuristic”. In: URL: https://medium.com/@bromanz/how-to-create-awesome-accelerators-the-surface-area-heuristic-e14b5dec6160.

Wolfe, Alan (2016). “Path Tracing – Getting Started With Diffuse and Emissive”. In: *The Blog at the bottom of the sea*. URL: https://blog.demofox.org/2016/09/21/path-tracing-getting-started-with-diffuse-and-emissive/.

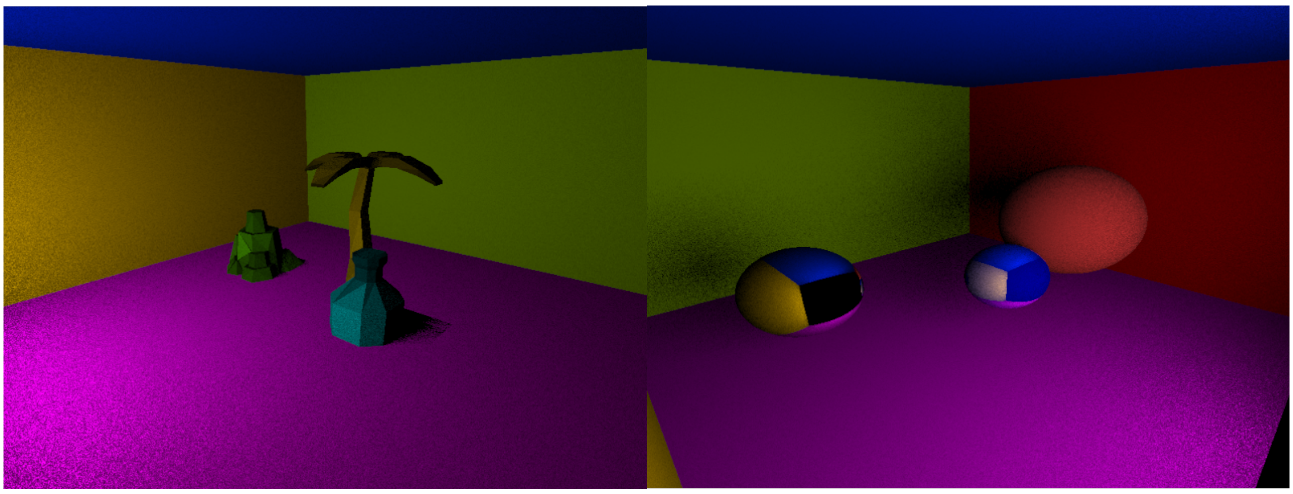


Figure 8: *Various scenes lit with an area light.*

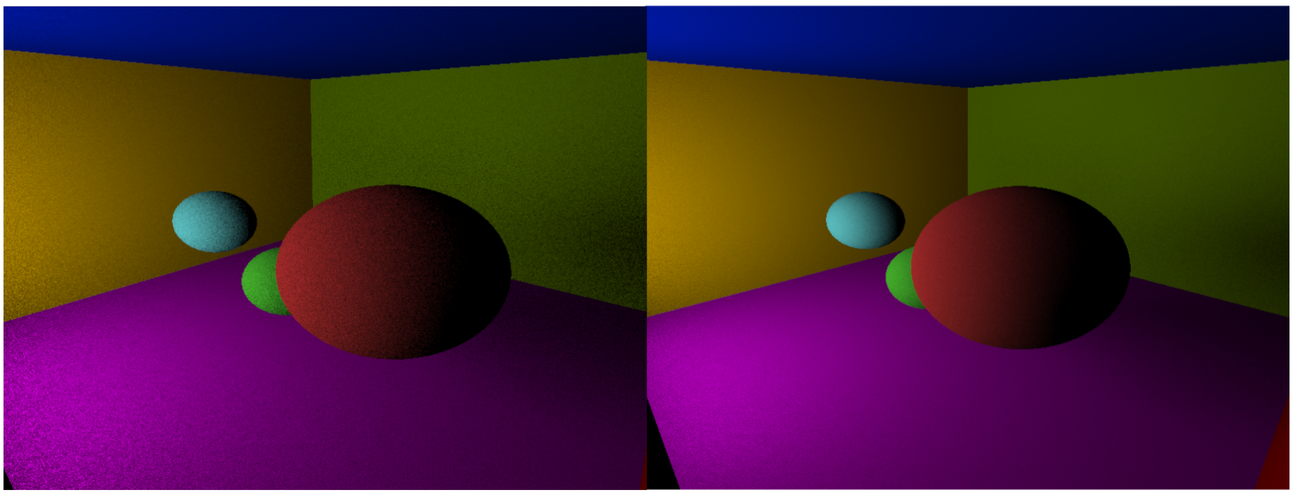


Figure 9: *Scene with area light with 8 and 72 samples respectively.*