Vinh Truong

Physically Based Rendering

Implementation of Path Tracer

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

28 April 2015

Author(s) Title Number of Pages Date	Vinh Truong Physically Based Rendering 43 pages 28 April 2015
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer Juha Kopu, Senior Lecturer

The main topic of this thesis was to implement a computer program that can render photorealistic images by simulating the laws of physics. In practice the program builds an image by finding every possible path that a light ray can travel. Technique presented in this thesis will naturally simulate many physical phenomenons such as reflections, glass materials, soft shadows, indirect lighting etc.

This thesis explains step-by-step how a pixel in an image gets its color by tracing a light ray through an arbitrary scene. The thesis also explains how to make the renderer run faster by optimizing certain data structures and exploiting parallellism in CPU and in GPU by using the OpenCL framework. Using these techniques it is possible to reduce the time spent in rendering from several days to a few minutes.

The thesis explains the theory behind path tracing algorithm, architecture of the program and its implementation details. Finally the thesis presents variety of images that the program is capable of generating and thoughts about future development.

Keywords	computer graphics, ray tracing, path tracing, rendering, photorealism, OpenGL, OpenCL

Tekijä(t) Otsikko Sivumäärä Aika	Vinh Truong Fysiikkaan perustuva renderointi 43 sivua 28.04.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Miikka Mäki-Uuro, lehtori Juha Kopu, lehtori

Insinöörityön pääaiheena on toteuttaa tietokoneohjelma joka simuloi fysiikan lakeja fotorealistisien kuvien piirtämiseen. Käytännössä ohjelma yrittää löytää kaikki mahdolliset polut joita valonsäde voi matkustaa ja kokoaa kuvan näistä poluista. Työssä esitetty tekniikka simuloi luonnollisesti monia fyysisiä ilmiöitä esimeriksi heijastukset, lasimateriaalit, pehmeät varjot, epäsuoravalaistus jne.

Työssä selitetään vaihe vaiheelta kuinka kuvan yksittäinen pikseli saa värinsä seuraamalla valonsädettä mielivaltaisen skenen läpi. Työssä käydään myös läpi kuinka renderointia voidaan nopeuttaa optimoimalla tiettyjä tietorakenteita ja hyödyntämällä rinnakkaislaskentaa prosessorilla sekä näytönohjaimella käyttämällä OpenCL-rajapintaa. Näillä tekniikoilla on mahdollista lyhentää renderointiin kuluvaa aikaa useista päivistä muutamiin minuutteihin.

Työssä käydään läpi teoriaa path tracing -algoritmista, ohjelman arkkitehtuuri ja sen toteutuksen yksityiskohdista. Lopuksi esitetään erilaisia kuvia joita ohjelma pystyy tuottamaan ja pohdintaa kuinka ohjelmaa pystyy kehittämään tulevaisuudessa.

Avainsanat	tietokonegrafiikka, säteenseuraaja, polunseuraaja, renderointi, fotorealistisuus, OpenGL, OpenCL

Table of Contents

List of Abbreviations

AABB	Axis-Aligned Bounding Box
BVH	Bounding volume hierarchy
CPU	Central processing unit
GCC	GNU Compiler Collection
GPU	Graphics processing unit
GUI	Graphical user interface
IDE	Integrated development environment
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
SAH	Surface area heuristic
STL	C++ Standard Template Library

1 Introduction

3D graphics is a fascinating topic that involves a wide variety of disciplines such as visual arts, mathematics, physics and computer programming to generate a believable image on a computer monitor. There are many fields that depend on 3D graphics with different constraints, one of the most important one being time. Simply speaking, most computer generated images will approach photorealism as the time spent on rendering them increases. For example, in video games the amount of time spent in generating a single image or a frame is only a couple of milliseconds, which means that sacrifices in realism have to be made in order to achieve interactivity. On the other hand, if accurate simulation of light is the main priority, the amount of time required to generate such an image could be many hours on a regular consumer hardware. Photorealism is desirable in many cases, for example, when 3D graphics have to blend in with live action footage as in movies with human actors. Another example would be in architectural designs where an architect would like to know how a given lighting setup would work in some indoor environment. Of course it is also needed by 3D artists for creating photorealistic models.

`phoe_ray` is a computer program created by the author of this thesis. It can be used to render a good looking image of a user provided 3D scene. The primary technique used by `phoe_ray` to accomplish this task is ray tracing, which is currently the most popular way to generate photorealistic images. There are multiple ways to augment the basic ray tracing algorithm, one of which is called path tracing. It is also one of the main subjects of this thesis. Other subjects include a brief overview of the theory behind ray tracing, philosophy and design decisions behind `phoe_ray`, implementation details of most significant features in `phoe_ray`, results that are generated using `phoe_ray` and finally some general thoughts on the development process.

2 Theory

This theory chapter gives a brief overview of the techniques and concepts behind the creation of photorealistic images.

2.1 Ray Tracing

Ray tracing algorithm is used to gather color information from a scene in order to render it to an image. A simplified illustration of this can be seen in Figure 1. The algorithm starts by iterating over every pixel on the image, generating a ray (”view ray” in the figure) that starts at the position of the camera, goes through the position of the pixel and towards to the scene. Then it traces that ray through the scene until it either hits a solid opaque surface or escapes to empty space. If the ray hits something, the algorithm will choose the closest hit point to the camera, which naturally sorts the objects back to front. The color of the pixel is therefore defined by where the ray has arrived.

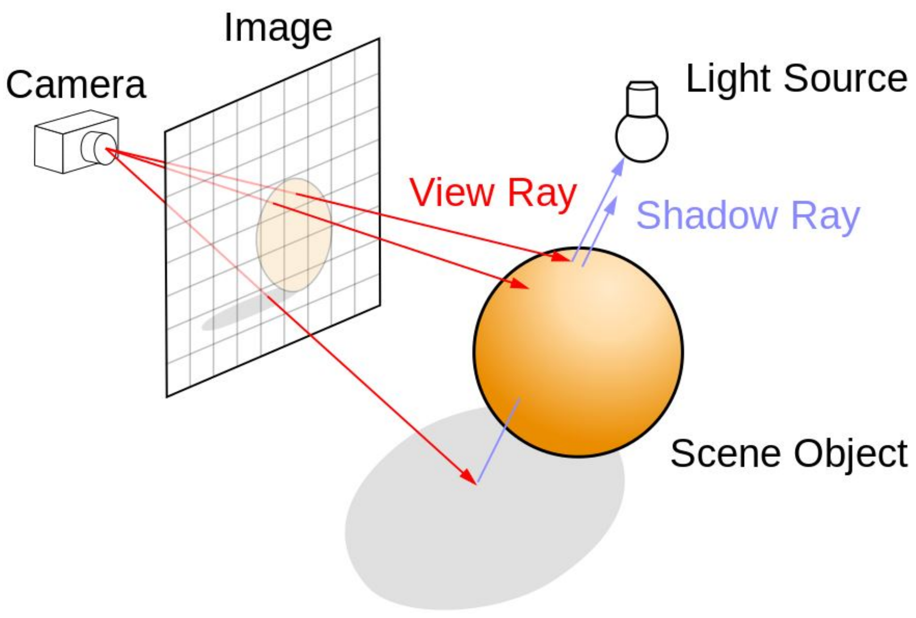


Figure 1: An illustration of ray tracing algorithm [1].

In order to determine where the shadows should be drawn, an additional ”shadow ray” will be created at the intersection point of the ray and its direction is aimed towards a light source in the scene. This can be seen in Figure 1. For example, when one of the rays hit below the sphere, a shadow ray is created and tested against the light source in the scene. This ray will hit the sphere on its way to the light source, therefore the color of the pixel that the initial ray was representing is set to black. The other two rays in the Figure 1 have a clear visibility to the light source, therefore the pixel is colored according to the surface of the sphere.

Additionally, if the ray hits a shiny object like a mirror, the algorithm reflects the ray about the surface normal at the intersection point and recursively invokes itself. This is also true for transparent surfaces, but in addition to reflection, this ray might be refracted instead.

The ray tracing algorithm explained here is also known as ”recursive ray tracing algorithm” invented by Turner Whitted in 1979 [2]. It handles reflections, refractions and simple shadows quite well and it produces a clean result relatively quickly. However, in its basic form it only considers direct lighting and it is not powerful enough to simulate effects such as indirect lighting or ”global illumination”. In the case of Figure 1, this algorithm would produce a perfectly sharp shadow below the sphere. Perfectly sharp shadows are only possible if the light source has no area, which do not exist in real life. This shadow is also perfectly black, because the algorithm does not consider any light that might have been reflected from other surfaces.

2.2 Global Illumination

Global illumination, also known as indirect illumination is a phenomenon where every surface is considered in lighting calculations. This is different from direct illumination which only considers surfaces that are directly visible to the light sources. While direct illumination is responsible for most of the lighting in the scene, it is not physically accurate. The effects of global illumination can be observed by standing in a dark room that has a small window shining light through it. If only the direct illumination is considered, most of the room would be completely black. In real life however, the room would be quite bright, because the light coming through the window would bounce from walls and furnitures before it gets absorbed. This effect can be seen in Figure 2, which shows the difference between direct illumination and global illumination. As seen in the figure, the sphere behind the cube completely invisible if no indirect lighting is simulated.

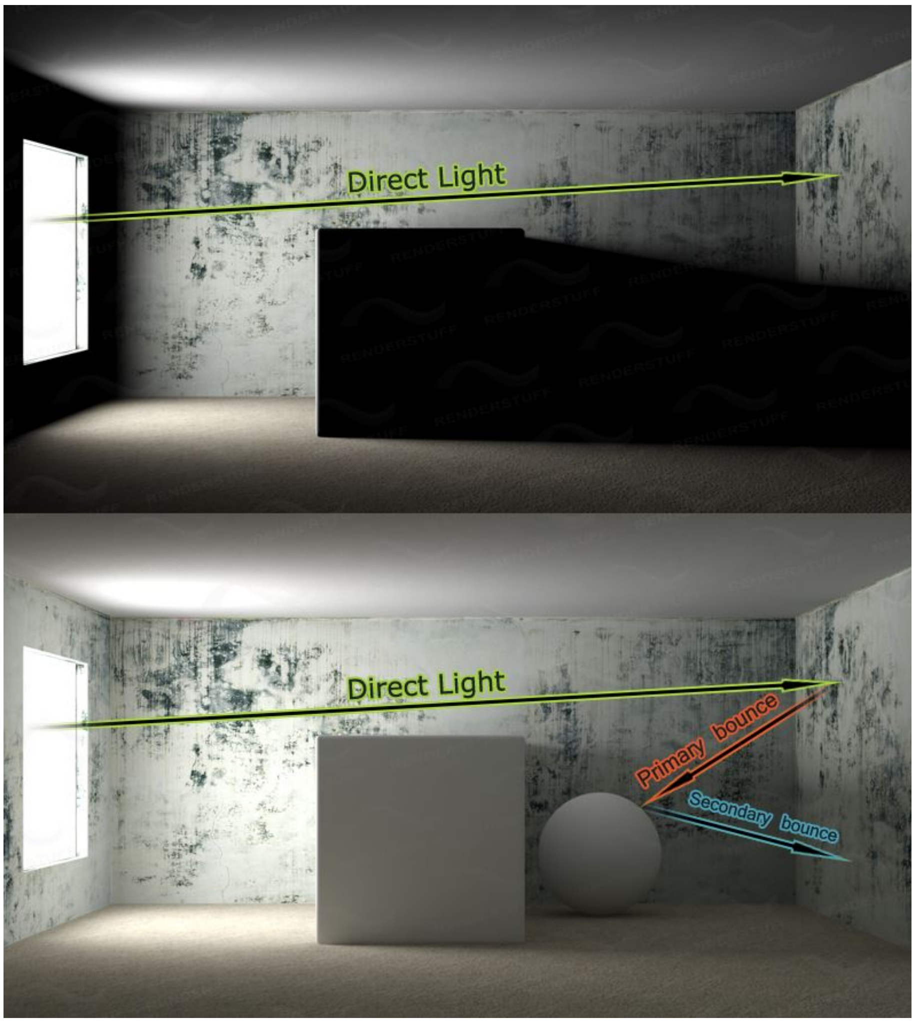


Figure 2: Comparison between only having direct lighting and having both direct and indirect lighting [3].

James Kajiya in 1986 [4] introduced the rendering equation into world of computer graphics. The rendering equation is an integral equation that describes a relationship between the outgoing light intensity of a surface element and the incoming light intensity from every other surface in the scene. In practice, this equation can not be solved analytically. One way to solve the rendering equation, is to approximate it by augmenting the original ray tracing algorithm with randomness, which leads to path tracing.

2.3 Path Tracing

Path tracing is similar to recursive ray tracing, the main difference being that instead of terminating the ray tracing routine after hitting an opaque surface, the path tracing routine continues from the hit position by randomly choosing a new direction. This is illustrated in Figure 3. It can be seen that after the initial hit on the red sphere, the algorithm will find a completely random direction for the new ray. This results in very different paths for the ray. The algorithm repeats this until the ray escapes to the outer space or enough bounces have been done. In this context, a single completely traced path is defined as a ”sample”. This sample has the information of every surface along the path. The color of the pixel is determined by using this information.

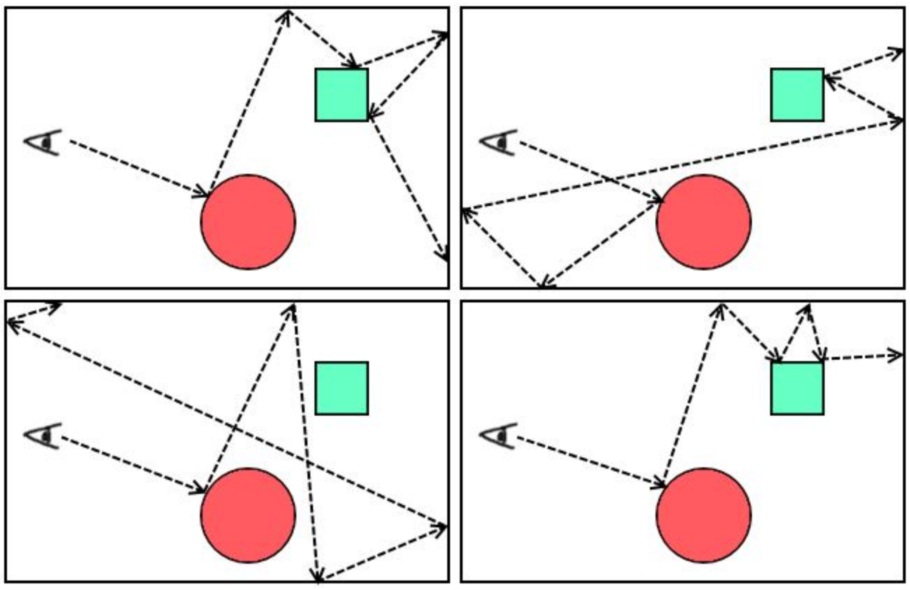


Figure 3: Valid paths for a ray.

The way of which the random directions are chosen is based on a hemisphere that is oriented towards the surface normal of the intersection point. This is known as the hemisphere sampling. This can be visualized in `phoe_ray` as seen in Figure 4.

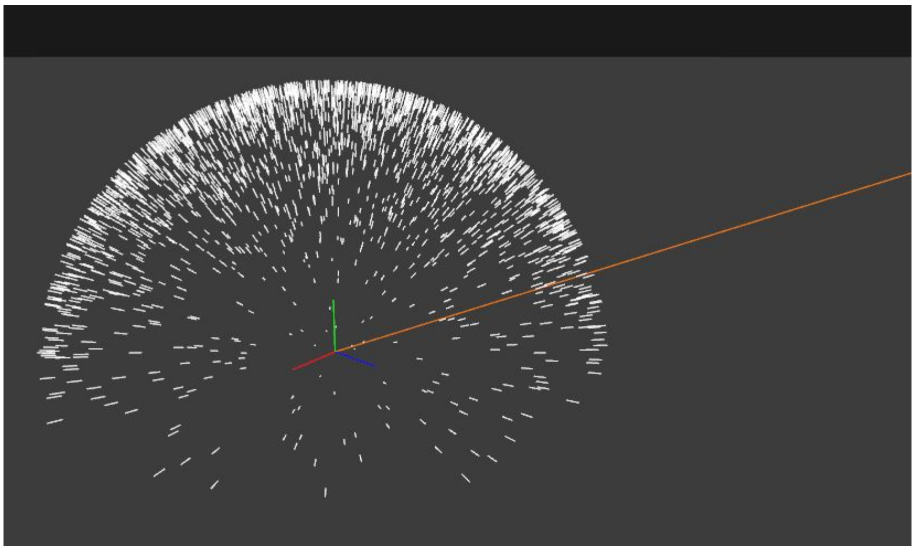


Figure 4: Hemisphere sampling visualization in `phoe_ray`.

In the Figure 4, the orange line represents the incoming ray to the surface. The green line represents the normal vector of the surface and together with red and blue vectors they form a basis to a local coordinate system. Using this coordinate system, it is now possible to choose a random direction that is evenly distributed on the surface of a hemisphere. These random choices are represented as white lines in the figure.

Hemisphere sampling has an additional detail that can be used to simulate roughness of the material. The definition of roughness can be seen in Figure 5. The roughness value essentially adjusts the cone of which the directions are sampled from. This definition of roughness in this context is arbitrary and specific to the implementation of `phoe_ray`.

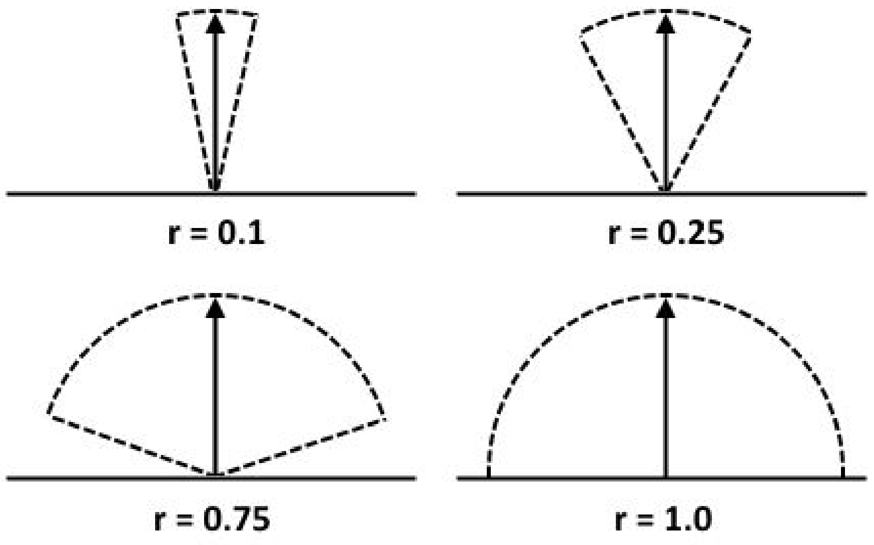


Figure 5: Different values of roughness r defined in `phoe_ray`.

As seen in Figure 5, when the roughness r approaches zero, the sampled vectors approach the normal vector that the hemisphere is currently aligned to. This hemisphere could be aligned to some other vector, such as reflection vector or refraction vector. Now, by randomly sampling the new direction vector off the hemisphere that is aligned to the reflection or the refraction vectors, it is possible to simulate rough mirror or rough glass.

This method of randomly choosing directions is known as the Monte Carlo simulation. Monte Carlo simulations require a large number of samples in order to converge to the solution. In the context of path tracing, once all the required samples are completed, they are averaged and saved as a color. An insufficient amount of samples will manifest as noise in the image. The exact number of samples required for a clean image depends on the scene. Usually after 2000 samples the image starts to become clean. The effect of the number of samples can be seen in Figure 18, Figure 21 and Figure 22.

3 `phoe_ray` Architecture

`phoe_ray` was developed by using a bottom-up, exploration-style approach which means that the design of the final program was not known at the beginning. Instead of pre-planning and writing design documents, the idea is to only focus on experimentation and implementation of new features. If an idea does not seem to contribute to the program, it can be removed early in development. Alternatively, the ideas that have proven to work in practice gets more attention to performance and readability etc.

`phoe_ray` is written entirely in C++ with large emphasis on procedural C-style. This means only using basic C features such as `struct`, `malloc`, pointers, arrays and functions. The reason why `phoe_ray` was not designed like a normal object-oriented program is because in practice when the program is being developed using the exploration approach, it is not known in advance which parts of the program are good subjects for concepts such as classes, interfaces and managers. If these are decided in advance, this by definition is pre-planning and resembles in writing design documents. This only works if the program behaviour and its features are known exactly by experience and previous work, which was not the case in `phoe_ray`. Writing programs in procedural C-style frees the developer from thinking about these concepts and focus only on the problem the program was supposed to solve. That being said there are a few convenient features that `phoe_ray` uses in C++ such as operator overloading for vector and matrix operations.

3.1 Libraries

`phoe_ray` uses several standard and third-party libraries for platform-specific features and minor conveniences such as mathematics and graphical user interface. Every library used in `phoe_ray` can be seen in Table 1.

Table 1: Libraries used in `phoe_ray`.

Name	Usage case
GLFW	Window, OpenGL Context, OS inputs and events.
GLEW	Searches and defines OpenGL types and functions to be used in source code.
GLM	3D vector and matrix operations.
OpenCL	A framework for writing and running general programs on a GPU.
stb_image	Load and save `.png` image files.
ImGui	GUI framework that can run on OpenGL.
C++ STL	`std::vector`, `std::thread`.
C standard library	File IO, printing, basic math, string manipulation and memory management.

These libraries work on multiple platforms, which allows `phoe_ray` to be supported at least on Windows and Linux. OpenCL is an exception, because its support not only depends on the platform, but also the hardware vendor of the GPU. Even though OpenCL is supposed to work across multiple hardware vendors, in practice there are differences in OpenCL implementations that can cause the program not run at all or run unreliably. Unfortunately these problems are hard to solve without owning one of each hardware vendors GPUs and testing on each one separately. `phoe_ray` was developed and tested on a NVIDIA GPU.

3.2 Asset Loading

`phoe_ray` supports `.png` image files and `.obj` 3D object files. Image files are used as textures. Textures are mapped across the surface of the object using UV-coordinates which are specified by the 3D object file. The exact details of UV-coordinates and texture mapping are out of scope for this thesis.

`.obj` format is used to describe the scene geometry and materials used in the scene. The geometry of the 3D object is defined by vertices (3D points in space) and triangle indices that span across those vertices. In addition, it describes UV-coordinates for each vertex which are used for texture mapping. The `.obj` format can be also used describe multiple mesh objects in the scene, for example a chair might be a different object than a table. This allows the program to discriminate between individual objects and describe materials in per-object basis. Objects material describes its color and surface properties such as emissivity or glossiness and whether or not it has a texture.

3.3 Preview Mode

In addition to path tracing modes, `phoe_ray` also supports previewing the scene before any path tracing begins. Preview mode displays a simplified version of the scene and therefore runs in real-time. Because path tracing is several thousand times slower than rendering a OpenGL scene, a preview mode is necessary for setting up the scene before path tracing. During the preview mode users can move and rotate the camera and change any materials in the scene. By default when path tracing is not happening, `phoe_ray` is on the preview mode.

3.4 Rendering Modes

`phoe_ray` supports two rendering modes: normal and live. Normal mode is the traditional way of how most ray tracer -based rendering works: user decides to render an image, all user interactions except for halting are disabled until the entire image is fully rendered. This simplest one to implement and it was the one that `phoe_ray` supported first from the beginning.

Live mode is when user interactions such as camera movement and material changes is allowed during rendering. To make this work, any previous progress made in rendering must be erased and entire rendering process has to be halted and restarted. Halting and restarting happens in less than few milliseconds to preserve interactive frame rates. Halting is non-trivial, because both path tracer implementations in `phoe_ray` have multiple threads running concurrently.

3.5 Path Tracing

Path tracer is the main feature of `phoe_ray`. As explained in Section 2.3, the path tracer uses the Monte Carlo methods for solving the rendering equation. This process involves thousands of millions of ray-triangle intersection tests for each rendered still image. Therefore it is necessary to have an acceleration structure to very quickly discard any unnecessary tests. Acceleration structures used in ray tracing reduces the number of intersection tests from $O(n)$ to $O(log(n))$. This is true, because instead of testing every single object in the scene, the acceleration structure cleverly splits the search space in half after each test. `phoe_ray` uses bounding volume hierarchy or BVH as its acceleration structure. The details of constructing and traversing the BVH are explained in the Section 4.3.

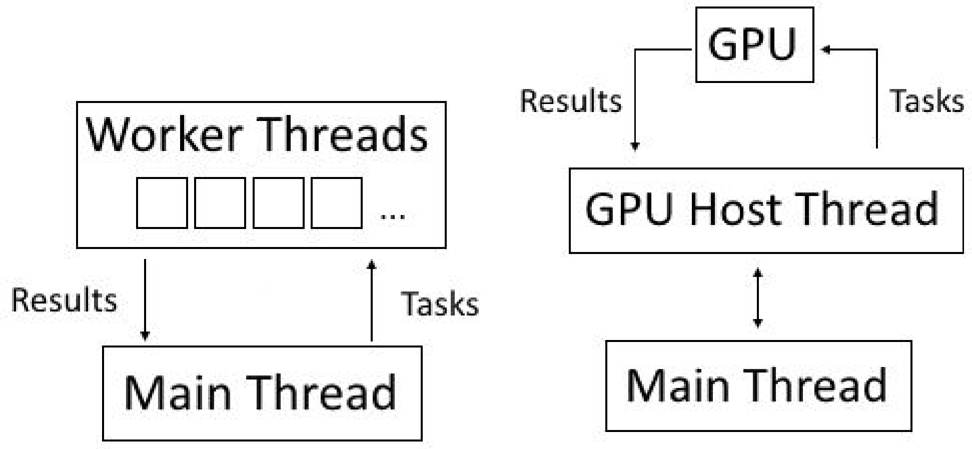


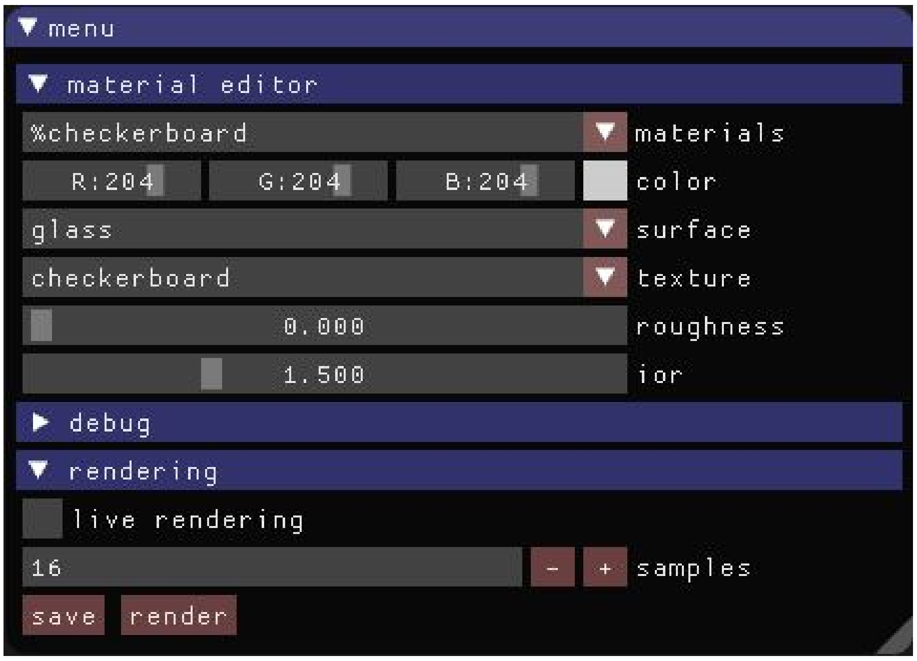
Figure 6: The two execution models used in `phoe_ray`.

`phoe_ray` implements two different versions of the same path tracer, one for the CPU and one for the GPU. While the path tracer algorithm itself and data structures surrounding it are identical on both implementations, the platform on which the two path tracers run on are completely different. The CPU version splits rendering job for multiple CPU cores and the results are gathered by the main thread. The GPU version runs almost entirely on the graphics card hardware. It has a CPU host thread assigned to it for doing CPU-GPU synchronization and gathering results for the main thread. An overview of both implementations can be seen in Figure 6. Details of both implementations are explained Sections 4.6 and 4.7.

3.6 Graphical User Interface

Graphical user interface or GUI in `phoe_ray` is relatively simple. It has several key roles:

- Displaying time information for currently active rendering process.
- Allow user to browse and edit materials.
- Allow user to toggle between rendering modes.
- Allow user to save the rendered image.
- Visualization of different systems in `phoe_ray` for testing purposes.



```
▼ menu
▼ material editor
%checkerboard ▼ materials
R:204  G:204  B:204  color
glass ▼ surface
checkerboard ▼ texture
0.000 roughness
1.500 ior
▶ debug
▼ rendering
live rendering
16  -  +  samples
save  render
```

Figure 7: GUI in `phoe_ray`.

ImGui library is used to implement GUI in `phoe_ray`. An example of usage of ImGUI in `phoe_ray` can be seen in Figure 7. As seen in the figure, ImGUI handles combo boxes, sliders, check boxes, buttons and grouped elements. At the time of writing, it is one of the simplest ways to get GUI on an OpenGL-based program. The library itself is only a few header files. There are no external design tools or markup languages, because the GUI is directly defined in the source code.

4 `phoe_ray` Implementation

Implementing `phoe_ray` took about 30 days of approximately 8 hours of work per day. `phoe_ray` is compiled using C++ compiler in Microsoft Visual Studio. Project files for Visual Studio are generated using CMake. As mentioned earlier in Section 3.1, every library used in `phoe_ray` runs on multiple platforms. CMake allows developers to easily generate project files for different IDEs and compilers. For example, CMake can generate Makefiles for Linux operating systems on both Clang and GCC compilers and Visual Studio project files on Windows.

4.1 Command Line Arguments

`phoe_ray` supports command line arguments and has default values for some of them. Table 2 contains every argument supported by `phoe_ray`:

Table 2: Command line arguments supported by `phoe_ray`.

Argument	Default value	Description
`-w`	1024	Screen width
`-h`	576	Screen height
`-s`	16	Samples per pixel
`-f`	-	3D object file path. User must provide this.

The user might launch the program in a following way: `phoe_ray.exe -w 1280 -h 720 -s 500 -f meshes/house.obj`. If some value other than the file path is missing or is invalid, a default value is chosen. `phoe_ray` will not start if the file path is not provided or if it does not exist.

4.2 Asset Loading

After the user has entered correct command line arguments, `phoe_ray` begins to load assets. As mentioned in Section 3.2, all assets are either stored in `.png` files or `.obj` files.

Asset loading is done in several phases. The first phase loads everything into memory and stores counts of every type of data. The second phase does preprocessing of the loaded data. Interestingly, almost all memory allocations in `phoe_ray` happens in this stage. Also every type of data, materials, textures, triangles etc. are all stored in simple C-arrays.

On GPU implementation, the preprocessing phase is mostly identical, except the data is copied into special data structures that are aligned to 16-byte boundaries. This is done by adding some extra padding in required locations. An example of this type of padding can be seen in Listing 1.

```
struct GPUTriangle {
    vec4 positions[3]; // 16 + 16 + 16 bytes
    vec4 normals[3];   // 16 + 16 + 16 bytes
    vec2 uvs[3];       // 8 + 8 + 8 bytes
    uint materialId;   // 4 bytes
    uint padding;      // 4 bytes
};
// sizeof(GPUTriangle) = 128, evenly divisible by 16
```
Listing 1: Data structure of a GPU triangle.

As seen in the Listing 1, it is helpful to comment very explicitly how many bytes each field will occupy in the `struct`. Incorrectly aligned data structures was one of the most common errors during the development of `phoe_ray`.

4.3 Bounding Volume Hierarchy

As mentioned briefly in Section 3.5, `phoe_ray` uses bounding volume hierarchy as its acceleration structure for ray-triangle intersect tests. This section explains what BVH is, how it is constructed and how to traverse through this hierarchy.

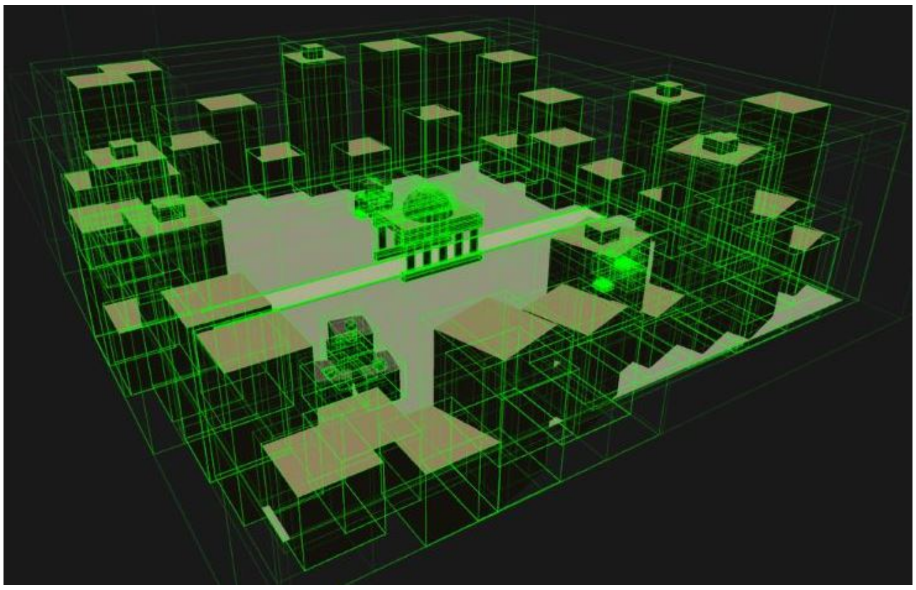


Figure 8: A visualization of BVH data structure.

BVH is a tree data structure, where each node in the tree has a bounding volume that encompasses all its child nodes and their bounding volumes. Top most node has a bounding volume that encompasses the entire scene and leaf nodes only contain a single primitive. In the context of ray tracing, if a ray misses a node, this guarantees that this ray missed every child node under the tested node. In practice it is very likely that ray will miss objects in the scene, because objects rarely intersect each other. Therefore the BVH will always perform much faster compared to brute force methods.

A visualization of BVH can be seen in the Figure 8. Each node in the BVH is colored as green in the figure. It can be seen that the BVH gets more complicated and dense in areas that have a lot of geometry.

4.3.1 Construction

Construction of BVH resembles the construction of binary trees, except the input sequence is not always split at the middle. In fact, the choice of this split position is very important. Poorly chosen split position (for example, always splitting at the middle) directly affects the performance of BVH. In traditional binary search over a list of integers, sorting is necessary in order for the binary search to work. In the case of BVH, it is not obvious how an arbitrary group of objects in 3D space can be sorted in an useful way. BVH construction strategy used in `phoe_ray` is from the book Physically Based Rendering [5]. Figure 9 shows how the BVH construction might happen in a 2D situation and Figure 10 shows what the Figure 9 looks like as a tree.

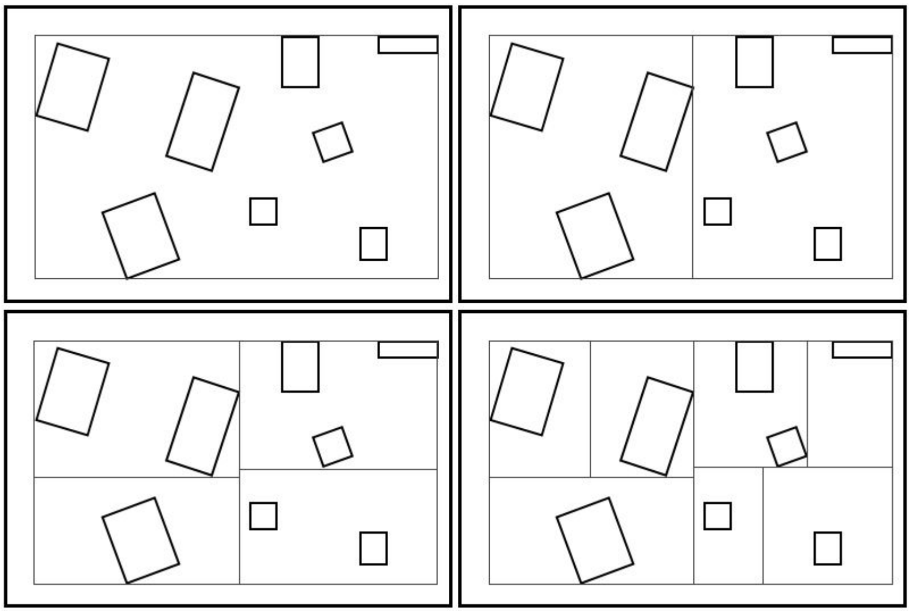


Figure 9: An example of BVH construction phase.

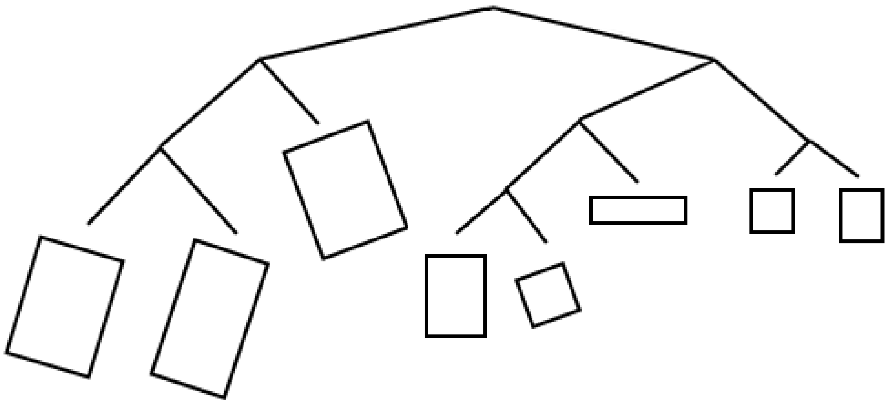


Figure 10: How the Figure 9 looks like as a tree.

First, the BVH construction algorithm generates an axis-aligned bounding box or an AABB for every primitive that is going to be stored in the BVH. AABB is a bounding box that is aligned to the three axes in three dimensions. It being axis-aligned simplifies many calculations, because it is possible to represent the AABB by only using six values, minimum and maximum distances for each axes. The algorithm also calculates the point that lies in the center of the AABB. This point is known as the centroid of the AABB. AABBs and centroids are used later in the construction algorithm.

The second phase is recursive and starts from the beginning with a complete list of primitives. The algorithm starts by computing an AABB that encompasses every primitives centroid point in the list. Then it determines the dimension in which to make the initial split. This dimension is chosen by finding the maximum extent of the AABB, which is essentially the longest side of the AABB. For example, an AABB defined as: $((0,0,0),(1,2,3))$, the Z-dimension will be chosen as the split dimension, because the AABB is longest in that dimension. Using this split dimension the algorithm starts to partition the primitives by using surface area heuristic or SAH. The reason why the surface area of a primitive is a good heuristic relates to the probability of rays hitting a given primitive. Larger primitive has a greater probability of being hit compared to a smaller one.

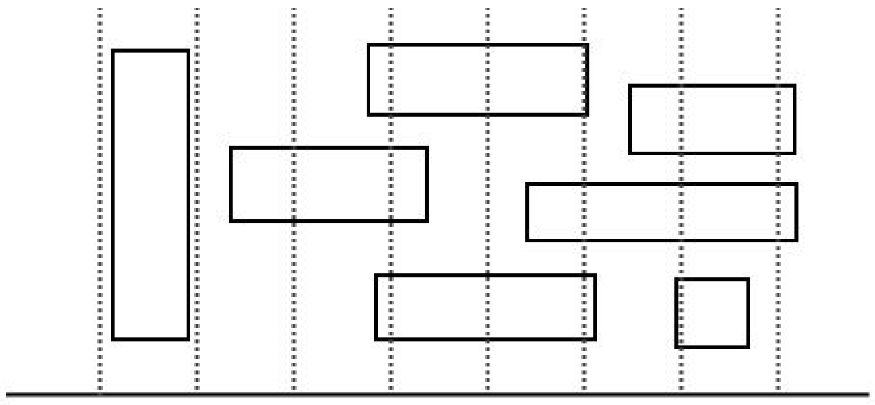


Figure 11: An example of trying out different split positions in SAH partitioning algorithm.

The SAH partitioning essentially tries find a split position where the resulting two child nodes after a split has been made has the smallest combined surface area compared to every other possible split position choices. One way to do this is by simply trying out every split position. This would guarantee the best possible split position, but in practice is too slow. Slightly faster approach is to choose a small number of evenly distributed split positions and try all of them. This is what `phoe_ray` does, it tries twelve different split positions and chooses the best one. Twelve is small enough number that even brute forcing does not take very long and produces good enough results in practice. An illustration of this technique can be seen in Figure 11.

Once the split position has been found, the split is done and the second phase is performed recursively on the two lists that came as a result of doing the split. The second phase stops going any further once a recursive call starts with only one primitive in the list.

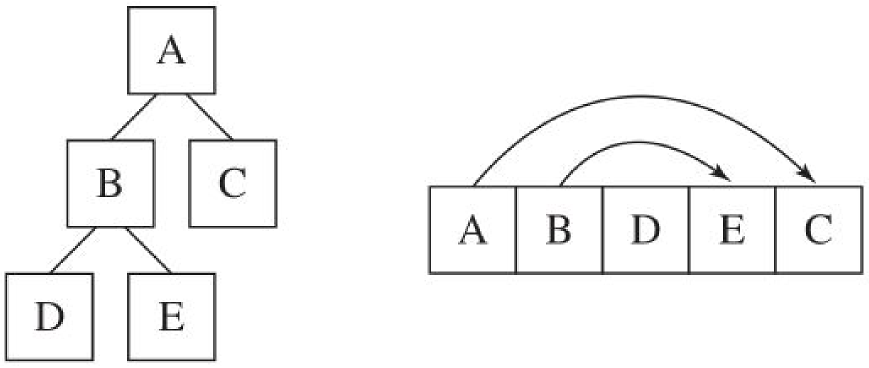


Figure 12: Flattening of a tree structure into a linear array format.

The final phase takes completely partitioned BVH tree form and flattens it to a pointer-less array form. An example of this can be seen in Figure 12. Nodes are stored in depth-first order. This makes it good for traversal purposes, because every node is next to each other in memory. Doing so improves cache, memory and overall performance.

4.3.2 Traversal

The traversal algorithm keeps a to-do list of node to be tested next. Pseudocode of this algorithm can be seen in Listing 2.

```
while(true) {
    if (intersectBBox(ray, currentNode)) {
        if (currentNode->primitiveCount == 0) {
            // leaf node
            intersectTriangle(
                ray,
                currentNode->primitive,
                &hit);
            if (todoList->count == 0) break;
        } else {
            addNode(currentNode->childs, todoList)
        }
    } else {
        if (todoList->count == 0) break;
    }

    currentNode = nextNode(todoList);
}
```

Listing 2: BVH traversal algorithm.

As seen in Listing 2, the traversal algorithm only stops once the to-do list is completely empty. This means that even if the algorithm already found a valid ray-primitive intersection, it will continue to look for other intersects along the same ray, because it might find an intersection that is closer to the camera. This naturally results in an image that has its elements perfectly depth sorted.

4.4 Preview Mode

Preview mode was briefly introduced in Section 3.3. It is implemented in modern OpenGL with programmable shaders. The main requirement of the preview mode is that it must run in smooth, interactive frame rates, which in this case means at least 60 Hz. Since most monitors update at least 60 times a second or 60 Hz, that should be the target frame rate for the smoothest possible user experience in a given monitor configuration. That said, `phoe_ray` does not explicitly limit the frame rate to 60 Hz. It uses vertical synchronization or VSync to prevent the preview mode to render more frequently than the monitor refresh rate allows. Rendering more frequently than a monitor can update is not only wasting resources, but also can cause image tearing where a half-rendered image is displayed on the monitor.

Because the preview mode is only supposed to serve as a tool to configure the scene for the path tracer, there is no need to make the preview mode look good or realistic. In practice it is good enough to have the scene at least make sense in the mind of the user (for example object shapes and colors should at least match the path traced image).

During the asset loading phase, in addition to building the structures such as BVH, `phoe_ray` also builds OpenGL buffers for rendering loaded 3D objects. Every 3D object has a vertex buffer associated with it. Vertex is a simple structure that describes a point in space. Its declaration can be seen in Listing 3.

```
struct Vertex {
    vec3 positions;
    vec3 normal;
    vec2 uv;
};
```

Listing 3: Structure of a vertex.

Every vertex buffer is uploaded to the GPU and their handles are stored in memory for rendering.

4.4.1 Camera

One of the main benefits of having a preview mode is giving the user an ability to set up the camera in the scene. This is like a photographer trying to aim the camera at the object that he/she is trying to photograph. `phoe_ray` allows this type of interaction in preview mode. In `phoe_ray`, the camera can be moved up, down, left, right, forward and back by using WSADQE-keys respectively. In addition to movement, the camera can also be oriented by using a mouse.

Camera movement requires three vectors that defines a local coordinate system to the camera. These vectors are: $\vec{f}_{camera}$, $\vec{u}_{camera}$ and $\vec{r}_{camera}$ (forward, up and right respectively). Using these three vectors makes it very easy to perform 3D movement, because now updating the position only requires one vector addition.

Orienting the camera using the mouse is harder to implement, because the objective is to map 2D mouse movement to a 3D rotation. `phoe_ray` uses spherical coordinates to do this. Converting from spherical coordinates to Cartesian coordinates is done by the following formulae:

$$\begin{aligned} x &= r \sin\theta \sin\varphi \\ y &= r \cos\theta \\ z &= r \sin\theta \cos\varphi \end{aligned} \tag{1}$$

where $\varphi \in [0, 2\pi)$ and $\theta \in [0, \pi]$. Orientation computations are only concerned with unit vectors which means that $r = 1$. To begin, the objective here is to take screen coordinates (x_{screen}, y_{screen}) and map them to (φ, θ) by normalizing both values to get $(x_{normalized}, y_{normalized})$. This is done by dividing screen coordinates with screen dimensions. Next the (φ, θ) is computed by using these normalized values. This is done by the following formulae:

$$\begin{aligned} \varphi &= 2\pi x_{normalized} \\ \theta &= \pi y_{normalized} \end{aligned} \tag{2}$$

Once the (φ, θ) is computed, they can be converted back to Cartesian coordinates using the Equation 1 to get the $\vec{f}_{camera}$ vector. Using the $\vec{f}_{camera}$ vector, it is now possible to derive both the $\vec{u}_{camera}$ and the $\vec{r}_{camera}$ vectors by doing two vector cross products:

$$\begin{aligned} \vec{r}_{camera} &= \vec{f}_{camera} \times \vec{u}_{world} \\ \vec{u}_{camera} &= \vec{r}_{camera} \times \vec{f}_{camera} \end{aligned} \tag{3}$$

where $\vec{u}_{world} = (0, 1, 0)$. $\vec{r}_{camera}$ is normalized between the cross products.

This entire computation is done in every frame update. It is insignificant enough to the overall performance that it is unnecessary to optimize any further.

4.4.2 Rendering

Preview mode rendering is closely related to basic video game rendering, except everything in the scene is static. `phoe_ray` implements a simple OpenGL rendering engine for the preview mode. The rendering procedure can be shown in the following pseudocode:

```
while (true) {
    sortObjectsByDepth();

    clearScreen();

    foreach (object in scene) {
        renderObject(object);
    }

    swapBuffers();
}
```

Listing 4: Rendering loop.

This loop will be executed forever until the user decides to terminate the program. Every time `swapBuffers()` is called, the front and back buffers are swapped (double buffering) and sleeps for a while. Sleep duration is determined by the monitor refresh rate to guarantee smooth interactions as explained in Section 4.4.

In order to make sure that the transparent objects are rendered correctly, every object in the scene have to be sorted back to front in relation to the position of the camera. This is done on every frame, because the camera might move due to user interaction.

4.5 Path Tracer

Path tracer implementation in `phoe_ray` can be expressed as a three step process as seen in Listing 5:

```
for (int sample = 0; sample < samplesPerPixel; sample++) {
    for (int y = 0; y < screen->height; y++) {
        for (int x = 0; x < screen->width; x++) {
            // Step 1
            Ray cameraRay;
            cameraRay = generateCameraRay(x, y, screen);
            // Step 2
            Color color;
            color = calculateRadiance(ray, scene, camera);
            // Step 3
            constructImage(output, accumulator,
                           x, y,
                           color, sample);
        }
    }
}
```

Listing 5: Path tracing routine.

There are several interesting observations when looking at the path tracing routine:

1. Each pixel on the screen is independent to every other pixel on the screen. This means that it is trivial to make this code run in parallel on any number of threads. This is also true on GPU implementation.
2. Time spent rendering is directly related to the number of samples per pixel and the resolution of the rendered image.
3. Scene has to stay constant during the rendering process. Any changes to the scene during rendering invalidates any progress achieved so far.
4. Rendering does not have to be fully finished in order to be displayed on the screen. This allows the renderer to give user some early feedback of the rendering process which is useful, because the image is partially recognizable even after only a few samples and therefore allows the user to cancel the process early if something is incorrect.

The three most important steps in the path tracing routine are: `generateCameraRay()`, `calculateRadiance()` and `constructImage()`. All three steps are explained in detail in their respective subsections 4.5.1, 4.5.2 and 4.5.3.

4.5.1 Generating Camera Rays

Like every other object in the scene, the camera in `phoe_ray` is also defined in world space. As seen in Section 4.5 Listing 5, for every pixel a corresponding camera ray is generated. The objective is to take pixel coordinates $(x, y) \in [0, screenDimension]^2$ and output a camera ray that starts at cameras position and points forwards towards the scene. There are several requirements that must be satisfied by the camera ray generator in `phoe_ray`:

1. Generated camera rays should result in an image that matches the preview mode output.
2. Camera rays have to respect perspective projection with a configurable field of view.
3. Ray directions should have a small random offset to effectively eliminate aliasing on the output image.

In order for the camera rays to match the preview mode output, they have to be converted to the screen space. This could be done in a single expression for each coordinate:

$$
\begin{aligned}
x_{screen} &= \left(2\left(\frac{x + 0.5 + x_{offset}}{screenWidth}\right) - 1\right) \times aspectRatio \times \tan\left(\frac{fov}{2}\right) \\
y_{screen} &= \left(1 - 2\left(\frac{y + 0.5 + y_{offset}}{screenHeight}\right)\right) \times \tan\left(\frac{fov}{2}\right)
\end{aligned}
\tag{4}
$$

where $(x_{offset}, y_{offset}) \in [-0.5, 0.5]^2$. Following steps explains how the expressions are derived:

1. Add 0.5 to pixel coordinates (x, y). This ensures that the ray will pass through middle of the pixel.
2. Add a random offset $(r_x, r_y) \in [-0.5, 0.5]^2$ to (x, y) that will slightly offset the ray off the center of the pixel. This eliminates aliasing from the output image, because every subsequent ray now samples a slightly different point in space.
3. Map pixel coordinates (x, y) to normalized device coordinates $(x_{ndc}, y_{ndc}) \in [0, 1]^2$ by dividing pixel coordinates with the screen dimensions. Note that the y-axis in NDC space points downwards.

4. Map the coordinates in NDC space (x_{ndc}, y_{ndc}) to screen space $(x_{screen}, y_{screen}) \in [-1, 1]^2$. This can be done like this: $(2x_{ndc} - 1, 1 - 2y_{ndc})$. Note that the y-axis in screen space points upwards.
5. x_{screen} is multiplied by the aspect ratio of the screen which is $\frac{screenWidth}{screenHeight}$, assuming $screenWidth \geq screenHeight$.
6. To account for the field of view, (x_{screen}, y_{screen}) is multiplied by $\tan\left(\frac{fov}{2}\right)$.

Finally, coordinates in screen space are now transformed into world space by using a 4x4 camera-to-world matrix which is an inverted version of world-to-camera matrix (which is also known as the coordinate system defined by cameras $\vec{f}_{camera}$, $\vec{u}_{camera}$ and $\vec{r}_{camera}$ vectors) derived in Section 4.4.1.

4.5.2 Calculating Radiance

Calculating radiance is the most important and time consuming process in `phoe_ray`. As seen in Section 4.5 Listing 5, this determines the radiance, or the color of the pixel on the image. A simplified version of this function can be seen in Listing 6.

```
 1 vec3 calculateRadiance(Ray* ray, Scene* scene) {
 2     vec3 color = vec3(1.0f);
 3     RayHit hit;
 4
 5     for (int depth = 0; depth < maxDepth; depth++) {
 6         if (intersect(ray, scene, &hit))  {
 7             // Color information
 8             if (hit.material.textureType == IMAGE) {
 9                 color *= sampleTexture(hit);
10             } else {
11                 color *= hit.material.color;
12             }
13
14             // Surface information
15             if (hit.material.surfaceType == EMISSIVE) {
16                 color *= hit.material.emission;
17                 break;
18             } else {
19                 ray->origin = hit.position;
20                 ray->direction = sampleDirection(hit);
21                 color *= lambert(hit, ray->direction);
22             }
23         } else {
24             color *= sampleSkyDome(ray);
25             break;
26         }
27     }
28
29     return color;
30 }
```

Listing 6: Calculates radiance.

Line 2

`color` variable is initialized to $(1, 1, 1)$ to indicate maximum possible radiance. This variable will be modified throughout the process, usually becoming smaller as the depth increases.

Line 3

`hit` variable keeps track of multiple attributes that can be gathered from ray-triangle intersection tests. These attributes are: position, normal, UV-coordinate and material.

Line 5

`maxDepth` in `phoe_ray` is set to 6. 6 is a good balance between correctness and performance in most simple scenes. Above a certain point each subsequent iteration contributes very little to the final color.

Line 6

Traverses through the bounding volume hierarchy and tries to find the closest ray-triangle intersection and fill the `RayHit` structure, if successful. Ray-Triangle intersection algorithm used here is called Möller-Trumbore intersection algorithm, named after its inventors Tomas Möller and Ben Trumbore [6]. Details of its derivation are out of scope for this thesis.

Lines 7 to 12

This part determines the color of the hit point. This color can be uniform across the entire mesh, but if an image is assigned to the triangle, this texture will be sampled instead. Sampling is done with UV-coordinates.

Lines 14 to 21

This part determines what to do the next iteration. In the case of emissive surfaces, the path tracing terminates. Every other surface type does hemisphere sampling based on roughness value. Color is modulated by the Lambert's cosine law, if applicable.

Lines 24 to 25

If intersection test fails, the only possibility is that the ray went into open space. One solution is to simply set the output color to be black. In this case `phoe_ray` experiments with image-based lighting and samples the sky dome texture instead. Because it is multiplying the existing color variable, the sky dome contributes to the lighting.

4.5.3 Image Construction

Once the radiance value is calculated for a given pixel, the value gets added into an accumulator buffer. After this, the final output color for a given pixel is calculated by following steps:

1. Average the pixels radiance values in the accumulator buffer.
2. Clamp the value between $[0, 1]$.

3. Raise the value to the power of 0.45 for gamma correction. The reason for gamma correction is related to how computer monitors display color. The exact details of this are out of scope for this thesis.
4. Multiply the value by 255.0 and cast it to `uint8`.

Final color value is then written into the output buffer. This can then be displayed on the screen or output as an image file.

4.5.4 Rendering Modes

As mentioned in Section 3.4, `phoe_ray` supports normal and live rendering modes. Implementing the normal rendering mode is straightforward: run the entire path tracing process from start to end normally.

The user can halt during the path tracing process and start again if desired. By making the time between the user issuing the halting command and actually halting very small (less than few milliseconds), it becomes feasible to implement the live rendering mode. The live rendering mode is essentially the same as the normal rendering mode, the only difference is that any changes to the scene or to the camera automatically halts and restarts the path tracing process. This can potentially give the user greater interactivity and feedback, assuming that the user has a powerful enough computer to run it. `phoe_ray` achieves fast halting times by inserting halting conditions into strategic locations in the code, such as between individual pixel samples and samples.

4.6 CPU Implementation

Even though the GPU implementation of `phoe_ray` is several times faster than the CPU implementation when running on the hardware that `phoe_ray` was being developed on, there are good reasons for sticking to the CPU implementation:

- It is easier to develop new features on CPU implementations, because of simpler debugging tools.

- It is guaranteed to work on almost any computer and on any operating system.
- It serves as a reference implementation for any other hardware accelerated implementations. Their correctness is determined by comparing them to the CPU implementation.

`phoe_ray` has been tested on Intel Sandy Bridge quad-core processor and AMD Phenom II hexa-core processor. Rendering performance scales linearly as core count increases on both CPU architectures which is expected.

4.6.1 Multithreading

Multithreading in `phoe_ray` is implemented by using the thread pool pattern. In the thread pool pattern, all worker threads are launched when the program starts and are only terminated once the program ends. There is an array of task buffers. The worker threads in the thread pool wait for a condition variable signal which is issued by the main thread. Once a worker thread receives a signal, it picks a task buffer that is allocated to it. It then begins to complete tasks sequentially until there are no tasks left. Once a task is completed by a worker thread, it increments a counter that represents the number of recently completed tasks. This is used to signal the main thread that some work has been completed. This is easier to see in pseudocode Listing 7.

```
void workerThread(int threadId, Task** buffers) {
    while (running) {
        waitUntilSignal();

        Task* myBuffer = buffers[threadId];

        foreach (task in myBuffer) {
            doPathTracing(task);
            recentlyCompletedTasks++;
        }
    }
}
```

Listing 7: Worker thread routine.

Important thing to note here is that the `doPathTracing()` could halt at any point, because of user interaction. This is handled carefully in `phoe_ray` to make sure the threads are correctly put on the idle state.

4.6.2 Load Balancing

`phoe_ray` implements a simple screen partitioning scheme, where the screen is split into small 8x8 pixel blocks. These blocks are then stored in a random order. This scheme was originally intended to make the rendering process look more visually pleasing, but it turned out to be an effective way to load balance tasks across the worker threads. The main reason why this is true is that in a given camera view to a scene, some parts of the screen might see the sky directly and other parts might see a dense pile of objects. In the case of four threads, if the screen were split into four equal sized quadrants and each quadrant is assigned to a thread, it is common that some threads will spend very different amounts of time to render those quadrants. Therefore, by splitting the screen into smaller 8x8 blocks and randomly distributing them, there is a higher probability that the workload will be evenly distributed across all the threads. This usually results in significantly reduced rendering times.

4.7 GPU Implementation

GPU implementation in `phoe_ray` is built on top of OpenCL framework. OpenCL offers simple abstractions for managing memory and running programs on a GPU. It is a cross-platform library that works on many popular hardware vendors devices, like Intel, NVIDIA and AMD. OpenCL comes with an OpenCL C compiler which compiles programs for the GPU. OpenCL C is a based on the C programming language, but features like C standard library and recursive functions are removed. An interesting note about OpenCL, is that most functions in OpenCL are thread-safe. `phoe_ray` does not benefit from this fact, but it is very powerful compared to OpenGL where most functions modify global states. More information about OpenCL and OpenCL C can be found in the book OpenCL Programming Guide [7].

In order to begin working with OpenCL, one has to perform the following steps in order:

1. Create a OpenCL context. OpenCL context is required in most operations.
2. Find a compatible compute device. In `phoe_ray`'s case this would be the GPU in the system.
3. Create a command queue for the compute device. Every GPU command has to be pushed into a queue for it to be executed. This includes buffer reads, buffer writes and running kernels.
4. Read kernels from source files and compile them.

Every step must be successful in order to continue. Every OpenCL function has a way for giving feedback if something went wrong. A function might return `NULL` or an error code.

Once OpenCL framework is set up correctly, it is time to allocate buffers and copy data over to the GPU. As mentioned briefly in Section 4.2, everything that is going to be sent over to the GPU have to be padded to satisfy 16-byte boundaries. `phoe_ray` does this during the asset loading phase.

4.7.1 Kernels

In the context of GPU programming, programs written for the GPU are often called ”kernels”. The main difference to regular programs is that kernels are submitted to hundreds of GPU cores and executed in a very parallel manner. In `phoe_ray`, kernels are written in OpenCL C language. Kernels have to be compiled before they can be used. The compilation phase can fail and it is possible to see compiler errors like in a traditional compiled language. Kernels in `phoe_ray` are compiled with the `-Werror` flag to treat every warning as an error. Here is a list of kernels in `phoe_ray` and a description of their role.

`ray_generator.cl`
    This is the `generateCameraRay()` as seen earlier in Section 4.5.1.

`pathtracer.cl`
    This is the `calculateRadiance()` as seen earlier in Section 4.5.2, except it also

includes hemisphere sampling functions, the ray-triangle intersection test and the BVH traversal function.

`compositor.cl`

This is the `constructImage()` as seen earlier in Section 4.5.3.

`clear_buffers.cl`

This is a special kernel that resets every pixel buffer to zero.

It is not necessary to have multiple `.cl` files, but in terms of readability it is useful to split the kernel code into multiple files, based on their role.

4.7.2 Execution Model

GPU implementation in `phoe_ray` has one worker thread assigned to submit tasks to the command queue and collect results once a task is done. Once every task is done, the worker thread starts from beginning until all samples are done. Before the worker thread is spawned, the main thread runs the `clear_buffers.cl` kernel and updates buffers that might have changed afterwards (user might have moved the camera or changed materials). These buffers include the material buffer and the camera attribute buffer. Once this is done, one worker thread is spawned and it will start to submit tasks immediately. Here is a pseudocode that explains what the worker thread does:

```
void gpuWorkerThread() {
    while (samplesDone < samplesPerPixel) {
        foreach (task in tasks) {
            push("ray_generator.cl", task, commandQueue);
            push("pathtracer.cl", task, commandQueue);
            push("compositor.cl", task, commandQueue);
        }
        wait(commandQueue);
        recentlyFinishedTasks++;
    }
}
```

Listing 8: GPU worker thread routine.

`task` in this context is similar to CPU implementations 8x8 pixel blocks, except here it is a group of 8192 pixels. In OpenCL it is possible to submit a group that is equal to the number of pixels on the screen, but in practice this might cause the operating system to freeze completely for several seconds. This usually crashes the graphics driver and subsequently the program as well. Based on empirical testing, it was found that values around 8192 do not cause this problem.

5 Results

The following figures showcase material types supported by `phoe_ray` and how the roughness modifier affects the material.

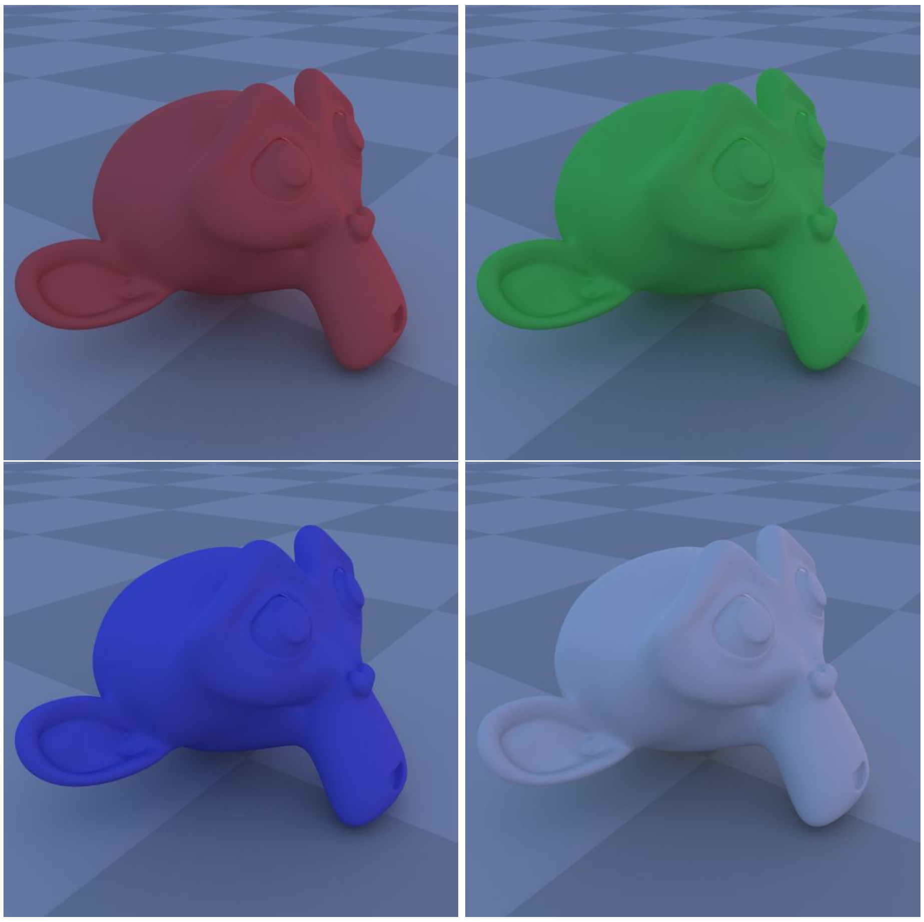


Figure 13: Diffuse material in variety of different colors.

One of the most impressive feature here is the soft shadows underneath the monkey's head. Similarly there are subtle shadows around the monkey's nose, eye regions and its left ear. Compared to rasterization-based graphics, the soft shadows are often implemented specifically in order to have them at all, while the path tracing algorithm handles them naturally. Diffuse material in `phoe_ray` does not use roughness value.

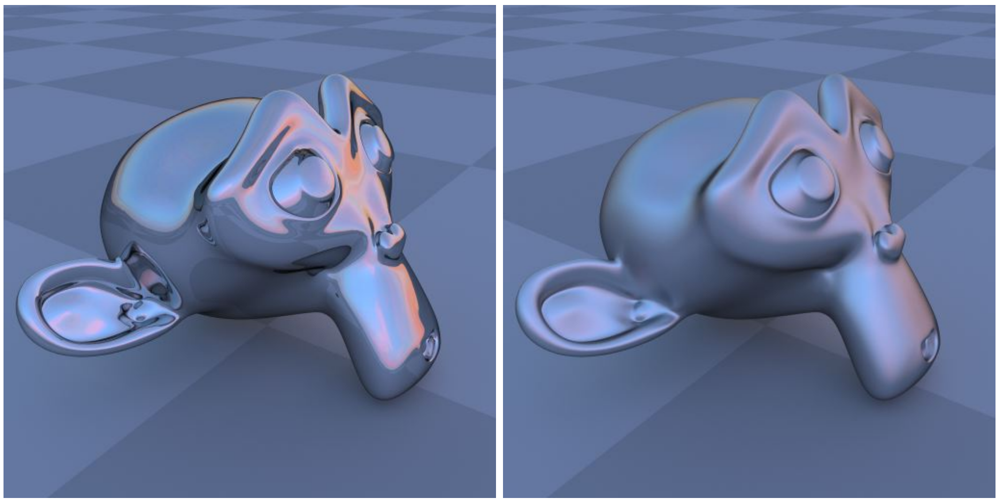


Figure 14: Glossy material (left: roughness = 0.0, right: roughness = 0.4).

Glossy material in `phoe_ray` is acts like a mirror. Here the roughness value has a noticeable effect on the materials reflectiveness. Increasing the roughness value is similar to a badly scratched mirror. This effect is also something that the path tracer simulates naturally without significant effort compared to rasterization-based methods.

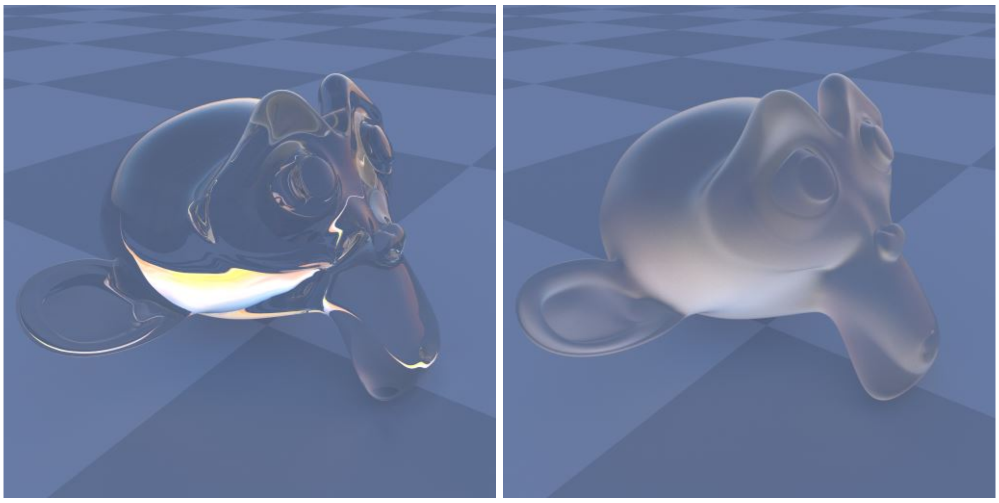


Figure 15: Glass material (left: roughness = 0.0, right: roughness = 0.4).

Glass material in `phoe_ray` can achieve surprisingly varied results depending on the roughness value and the index of refraction. A value of 1.4 is used as the index of refraction in Figure 15. Frosted glass -type of material can be achieved by changing the roughness value.

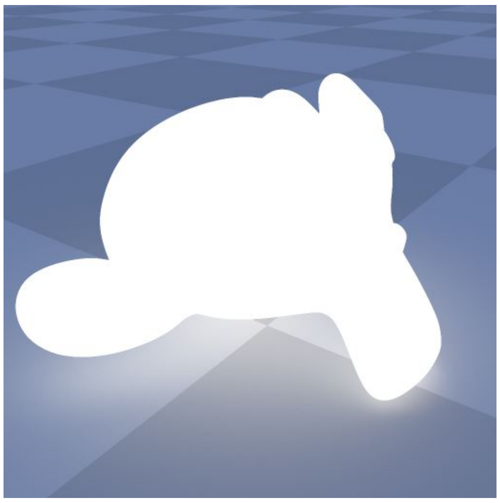


Figure 16: Emissive material.

As emission power increases, the emissive material becomes completely white. It is possible to emit any color when the emission power and the material color is set correctly.

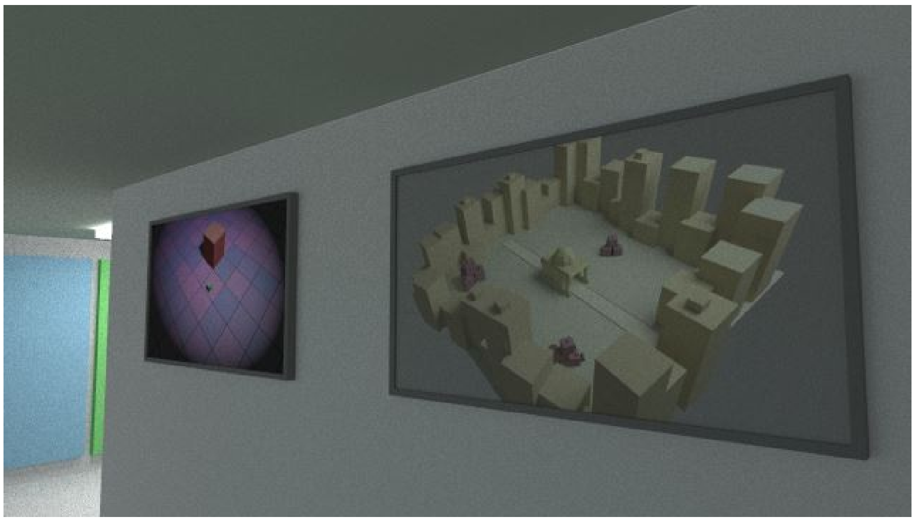


Figure 17: Textured paintings.

`phoe_ray` supports simple texture mapping. These two paintings are read from `.png` image files.

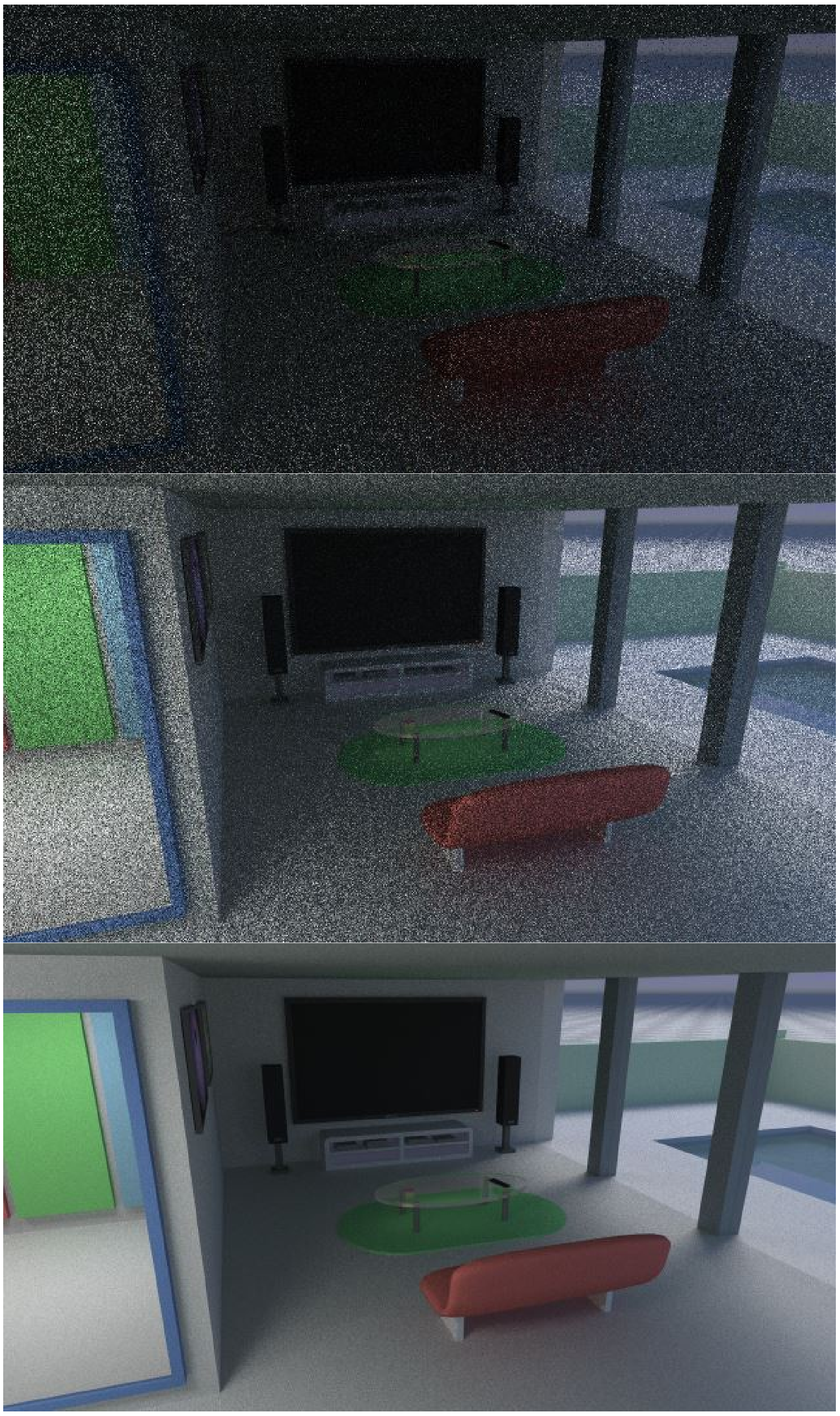


Figure 18: Effect of sample count (first = 1 sample, second = 16 samples, third = 1024 samples).

Figure 18 demonstrates what happens when an image is rendered with different sample counts. The image starts out as extremely noisy and barely recognizable and as the sample count increases, it starts to get clearer. Number of samples required to get an image without noticeable noise varies greatly depending on the scene. Typically scenes that have an extremely large light source (like the sun) tend to require much less samples compared to scenes that have smaller light sources (like a light bulb).

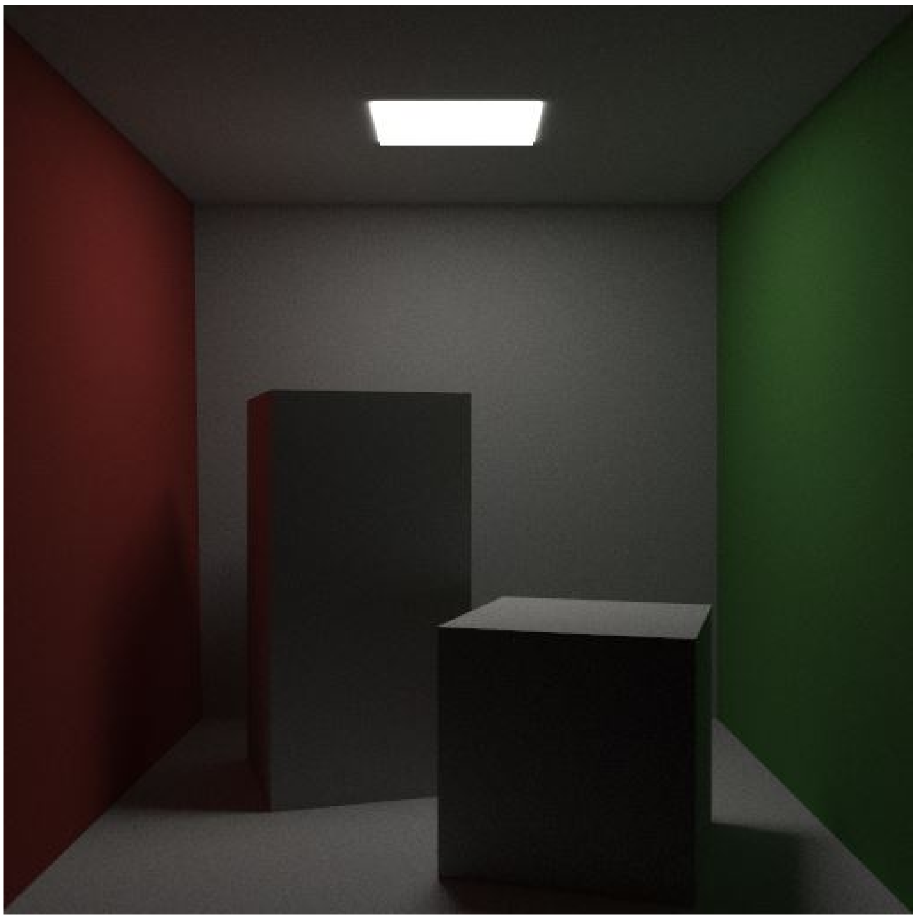


Figure 19: The famous Cornell Box.

Figure 19 showcases the famous scene in computer graphics research called Cornell Box which can be downloaded here [8]. Here it is possible to see the effects of global illumination, which is what path tracer was originally designed to solve. Boxes in the middle are completely white, but their sides closest to colored walls have wall colors reflected to them. This is purely due to indirect lighting, because these sides are not directly visible to the lamp on the ceiling.

All images in this chapter were rendered with a following hardware configuration:

- CPU: Intel i5-2500k
- GPU: Nvidia GeForce GTX 560 Ti
- RAM: 8 GB of DDR3 Memory
- OS: Windows 8.1

Figure 20 displays some benchmarking results from `phoe_ray`. The benchmarking here was done using the hardware described above and the scene being benchmarked here is the scene with the monkey head. 32 samples were used and the resolution is set to 960 by 544.

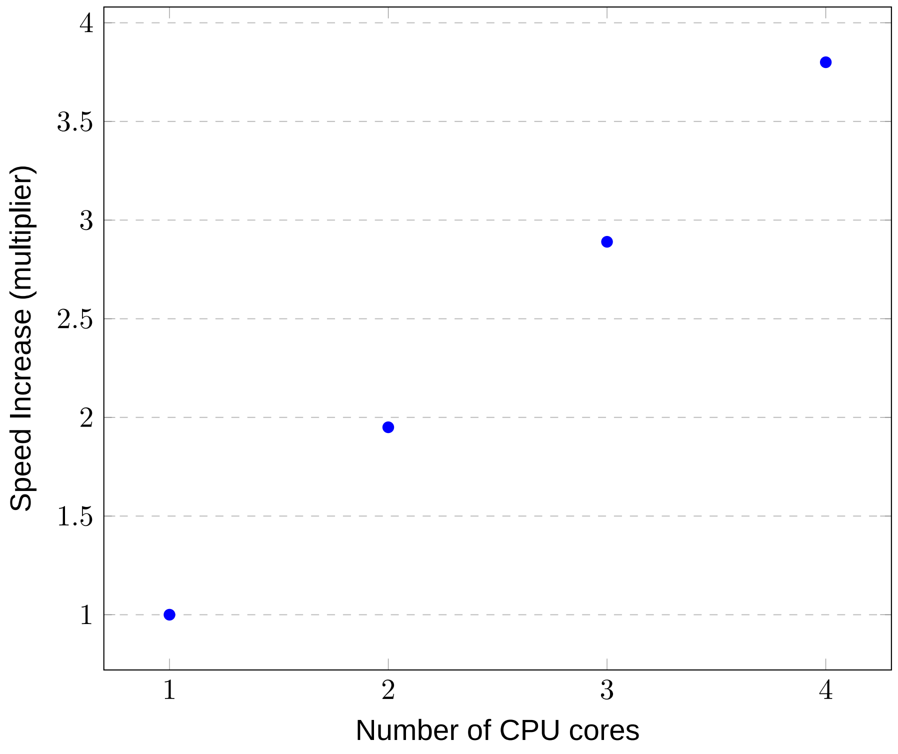


Figure 20: How the number of CPU cores increases the speed of rendering.

As seen in Figure 20, as the number of core increases, the speed increases grows linearly. The linear growth should continue indefinitely, because there are no thread communication in `phoe_ray`.

The next two figures 21 and 22 shows how the variance of pixels in an image reduces as the number of samples increases in the monkey scene (as seen in Figure 13) and the house scene (as seen in Figure 18), respectively. The setup used here is similar to the one used in Figure 20, except the number of samples varies from 1 to 128.

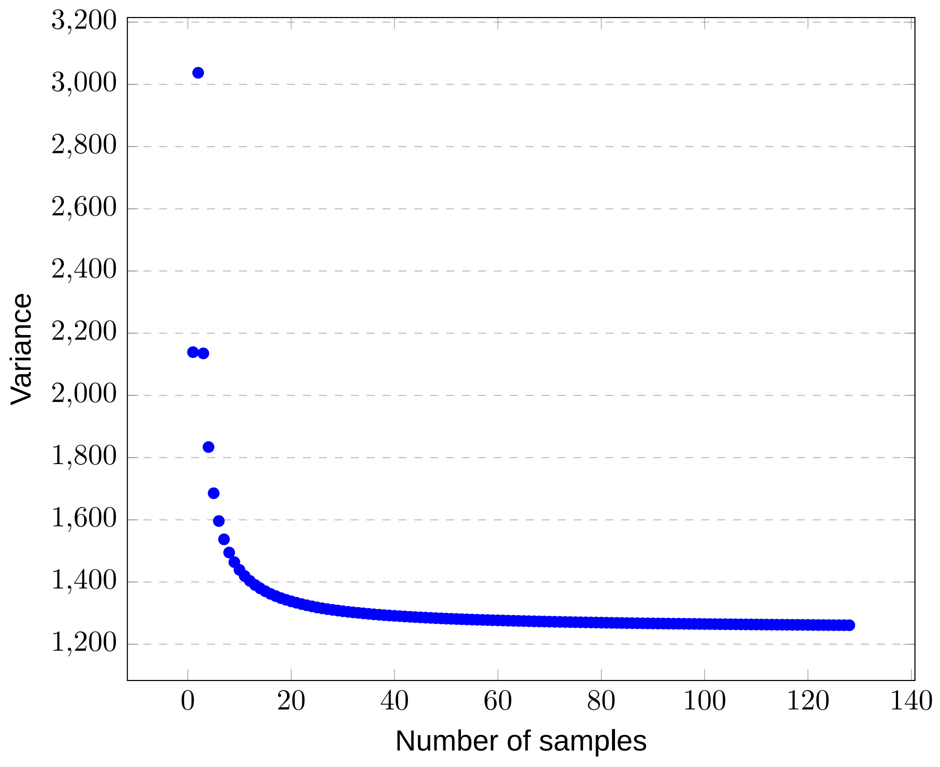


Figure 21: How the sample count decreases the variance of the image (monkey scene).

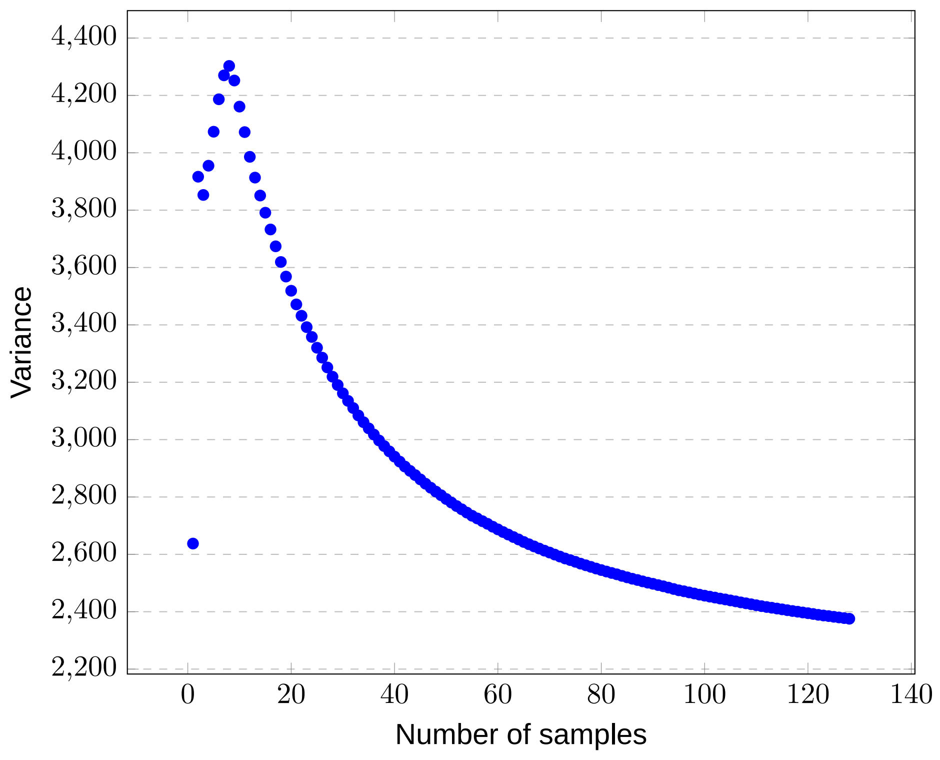


Figure 22: How the sample count decreases the variance of the image (house scene).

The actual values for variance in figures 21 and 22 are not important, the rate of convergence however, is. It is clear that in the monkey scene, the variance converges much faster compared to the house scene. This is because the monkey scene is completely visible to the sky which is an extremely large light emitter, while in the house scene almost nothing is directly visible to the sky. Most lighting in the house scene is from indirect lighting, which requires more samples.

6 Conclusion

The most valuable features in `phoe_ray` that turned out to be essential to the development include: visualization of data structures for clarity, prioritizing quick iteration times for fast debugging and having a preview mode with a GUI on top of it which also affects debugging and testing efficiency. A feature such as the live rendering mode is very useful in this context, but also in general it is always worth the time to reduce friction in getting things done, even if it only remains as a developer-only feature.

There are many exciting ways to extend and to augment existing features in `phoe_ray`. E.g. features such as being able to render multiple frames in sequence to produce motion, as in animation. This includes character animations and physics simulations, too. Other interesting topics are new types of materials, such as human skin, tree leaf, milk and wax. Simulating these realistically requires a general support for subsurface scattering. There are also ways to improve the current diffuse and glass materials to make them even more realistic, the current models used in `phoe_ray` are quite simple compared to what is available. In terms of optimization possibilities, the GPU implementation could be made even more efficient by implementing the bounding volume hierarchy specifically for the GPU.

In all, `phoe_ray` can easily be considered as a success. Every feature imagined by the author at the beginning of the project were implemented and even more. Many new concepts and techniques were learned during the development, the main one being able to program the GPU for large calculations, such as path tracing.

References

1 Ray tracing (graphics);. Available from: http://en.wikipedia.org/wiki/Ray_tracing_(graphics) [cited April 13, 2015].

2 J D Foley, Turner Whitted. An Improved Illumination Model for Shaded Display. SIGGRAPH ’05 ACM SIGGRAPH 2005 Courses. 1979;Article No. 4.

3 Best V-Ray settings - Indirect illumination;. Available from: http://renderstuff.com/vray-indirect-illumination-best-settings-cg-tutorial/ [cited April 13, 2015].

4 James T Kajiya. The rendering equation. ACM SIGGRAPH Computer Graphics. 1986;Volume 20 Issue 4.

5 Matt Pharr, Greg Humphreys. Physically Based Rendering, Second Edition: From Theory To Implementation; 2010.

6 Tomas Möller, Ben Trumbore. Fast, Minimum Storage Ray/Triangle Intersection. Journal of Graphics Tools. 1997;Volume 2 Issue 1.

7 Aaftab Munshi TJ Benedict Gaster, Dan Ginsburg. OpenCL Programming Guide; 2011.

8 Cornell Box Data;. Available from: http://www.graphics.cornell.edu/online/box/data.html [cited April 12, 2015].