



**Utrecht University**

Master Thesis

---

**Porting and Maintaining a Volatile GPU Path Tracer  
Codebase to the CPU as a Fallback for Large Scenes**

---

Supervisors: dr. ing. Jacco Bikker, dr. ir. A. Frank van der Stappen

Basar Oguz

*ICA-6084990*

*Faculty of Science*

*Department of Information and Computing Sciences*

*16 August 2019*

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Objectives . . . . .	4
1.2	Research Questions . . . . .	4
1.3	Content Overview . . . . .	5
<b>2</b>	<b>Preliminaries</b>	<b>6</b>
2.1	Physically Based Rendering . . . . .	6
2.1.1	The Rendering Equation . . . . .	6
2.1.2	Path Tracing . . . . .	7
2.1.3	Acceleration Structures . . . . .	8
2.2	Case Study: LightHouse . . . . .	9
2.2.1	Application Layer . . . . .	10
2.2.2	Render System . . . . .	10
2.2.3	Render Core . . . . .	11
<b>3</b>	<b>Literature Review</b>	<b>13</b>
3.1	Software Portability . . . . .	13
3.1.1	Early influential work . . . . .	13
3.1.2	Relevant Methods and Techniques . . . . .	14
3.1.3	Portability Metrics . . . . .	17
3.2	High Level CPU Path Tracing APIs . . . . .	18
3.2.1	Embree . . . . .	19
<b>4</b>	<b>Research Methodology</b>	<b>21</b>
4.1	Porting and Implementation . . . . .	21
4.2	Experimental Setup . . . . .	21
4.2.1	Rendering Very Large Scenes . . . . .	21
4.2.2	Performance Evaluation . . . . .	22
4.2.3	Portability Measurement . . . . .	22
<b>5</b>	<b>Implementation</b>	<b>23</b>
5.1	Architecture and System Design . . . . .	23
5.2	Geometry Loading . . . . .	24
5.3	Path Tracer . . . . .	26
5.3.1	Wavefront Formulation . . . . .	27
5.3.2	Buffers . . . . .	28
5.4	Shading . . . . .	29
<b>6</b>	<b>Results and Evaluation</b>	<b>31</b>
6.1	Rendering Very Large Scenes . . . . .	32
6.2	Rendering Performance . . . . .	35

6.3	Portability and Maintenance Cost . . . . .	38
<b>7</b>	<b>Conclusion</b>	<b>41</b>
7.1	Future Work . . . . .	41
<b>8</b>	<b>References</b>	<b>43</b>

# 1. Introduction

Rendering and visualizing virtual scenes is a task solved diversely across industries. Movie and visualization industries typically demand high realism and pay less attention to performance, whereas interactive web applications and games indispensably demand images to be rendered in real-time. Recent hardware developments have allowed for the use of physically based rendering algorithms such as path tracing in real-time for applications. However, due to memory limitations in GPUs, large scenes that do not fit in device memory need to be rendered using the CPU.

Software porting is the practice of producing a software unit in a new environment, based on an existing version. The need to port software may emerge from a wide range of situations, including but not limited to supporting a wider range of hardware architectures, compilers or operating systems. Our research focuses on a particular subset of software porting, between two distinct hardware architectures, namely the Central Processing Unit (CPU) and the Graphical Processing Unit (GPU). The existing software that our case study was conducted on is a production quality, photorealistic image renderer, LightHouse, that initially runs on the GPUs.

Ultimately, the rendering software, LightHouse, should be designed in such a way that the main GPU rendering core can be ported to different environments and maintained with minimal effort. Our target environment to port the main rendering core is the CPU, which will only be used for scenes that contain geometry that exceeds the memory of the GPU. The porting project consists of four steps: scoping, analysis, porting and testing. Due to different processor architectures, instruction sets and lack of support for specialized GPU programming languages on the CPU side, it is not possible to port the software without significant effort. In order to keep the logic of the existing and the ported cores synchronized, careful analysis and re-implementation of the source code is essential.

## 1.1. Objectives

Given the existing and volatile GPU path tracer codebase, LightHouse, this project aims to develop improved methodologies to continuously port and maintain an integrated CPU fallback system. It is ultimately expected that the two software units produce the identical output with common software features while targeting different hardwares. Another primary objective is keeping the software maintenance low-cost. Additionally, it is expected that the ported version exhibits a performance level consistent with state of the art CPU ray tracing performance. Formal definitions and metrics of software portability and maintenance will be considered to achieve and verify our objectives.

## 1.2. Research Questions

There are three main research questions this study aims to answer, respectively in the fields of *software design*, *maintenance* and *performance*.

**RQ1** - Can a ported CPU path tracer be a reliable fallback strategy to the source GPU

path tracer for large scenes?

**RQ2** - How does the performance of a ported CPU path tracer compare to the source GPU path tracer?

**RQ3** - Given a volatile GPU path tracer codebase, how can a ported CPU path tracing codebase be maintained?

### 1.3. Content Overview

This document contains two sections on the theoretical background of the thesis project. Chapter 2 contains the preliminaries, in which we describe the path tracing algorithm which plays an important role in our case study. Chapter 3 establishes the state of the art on *software portability* and *CPU-based rendering*. Chapter 4 explains design and reasoning behind the experiments, whereas Chapter 6 presents the results of these experiments. Chapter 5 gives a detailed explanation of our ported software implementation. The thesis concludes with Chapter 7 that provides an answer to the research questions and a discussion future work.

## 2. Preliminaries

In this chapter, we provide background information on the problem field by explaining the specific algorithm that will be ported from GPU to CPU. Rendering is the process of producing a digital image, given a 3D scene. In order to simulate how light behaves and interacts with matter in nature, physically based rendering techniques have been developed. However obviously superior these methods seem, they have only recently been adopted by performance critical applications such as games because of their high computational cost. The group of algorithms that capture these principles for realistic lighting are typically called *global illumination algorithms*. The renderer in our use case uses *path tracing*, a global illumination algorithm that uses random sampling to approximate the physically accurate rendering equation.

### 2.1. Physically Based Rendering

Rendering refers to the process of creating a 2D image from the representation of a 3D scene. There are many ways to achieve this with varying degrees of realism, performance and detail. Performance critical applications such as games have typically used rasterization based methods due to its relatively light computational cost. However, due to the increased compute capability of recent hardware, ray tracing based rendering methods are increasingly adopted by real-time applications.

Rasterization based rendering methods project the scene geometry towards the screen plane, processing each triangle independently. In order to accurately model reflections, refractions and global illumination, approximations and heuristic methods are used.

Physically based rendering methods are typically based on *ray tracing* algorithms, first described by Arthur Appel in 1968 in a paper titled "*Some Techniques for Shading Machine Renderings of Solids*" [App68]. The common goal of these methods is photorealism; the synthesis of images that are indistinguishable from the photograph of the same scene [PJH16]. Ray tracing aims to render the scene by following the path or a light ray through the scene. As the ray interacts with and bounces in and out of objects in the environment, it simulates photorealistic effects such as reflections, refractions and global illumination.

#### 2.1.1. The Rendering Equation

In order to simulate the way light behaves in the nature, an integral equation has been proposed by Kajiya, which was adopted for use by the graphics research community, and production renderers [Kaj86]. The integral equation models light leaving a point on a surface, by expressing it as the sum of the incoming light energy and the light emitted by that surface.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\omega} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) \cos \theta_i d\omega_i \quad (1)$$

- $L_o(x, \omega_o)$  the outgoing radiance towards the direction  $\omega_o$ .

- $L_e(x, \omega_o)$  is the radiance emitted towards the  $\omega_o$  direction.
- $f_r(x, \omega_i, \omega_o)$  is the bidirectional reflectance distribution function (BRDF) which is a function of the incoming ( $\omega_i$ ) and outgoing ( $\omega_o$ ) directions for a given position to calculate how much of the incoming light is reflected towards the outgoing direction.
- $L_i(x, \omega_i)$  is the amount of incoming radiance from the  $\omega_i$  direction.
- $\cos \theta_i$  term functions to convert radiance to irradiance, and affects all angles except 0.

It is important to note that this integral is evaluated over the hemisphere of the point  $x$ . The intensity of the incoming light is attenuated by the bidirectional reflectance distribution function (BRDF) corresponding to the material of the surface [Nic65].

The model so far does not account for materials that *transmit* light through. In order to capture how a material affects the light passing through it, the concept of bidirectional transmittance distribution function (BTDF) was introduced. BTDF, in principle reverses the surface and reports how much light is transmitted to an angle in the hemisphere that lies below the surface at the point  $x$ . BRDF and BTDF are combined in the BSDF, which is the *bidirectional scattering distribution function*.

### 2.1.2. Path Tracing

The evaluation of the recursive rendering equation is the goal of photorealistic renderers. Solving this high-dimensional integral would require the calculation of incoming light energy on every point from an infinite number of directions. This is computationally exhaustive and impossible to implement. Kajiya proposes an algorithm called *path tracing* to stochastically evaluate the integral by using Monte Carlo integration [Kaj86].

Path tracing is a stochastic experiment that approximates the integral in Equation 1 by taking light energy samples along random paths. A ray starting from the origin of the camera is traced through the center of each screen pixel, intersecting the geometry in the scene. At every intersection, the light rays are extended along a random path. The paths are terminated if they hit a light source or reach a maximum recursion depth.

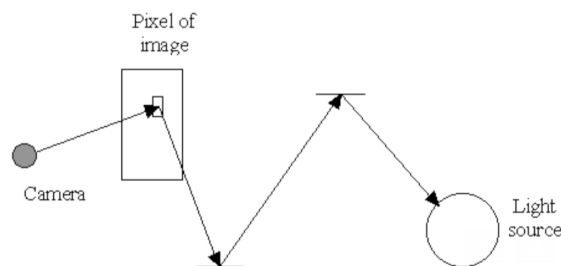


Figure 1: The random walk of light rays generated by the path tracing algorithm

Tracing a single path per pixel results in a noisy image. To produce a noise-free image, continuous samples are taken from every pixel and they are averaged in time. This way, the image *converges* to the actual representation of the scene.

Increasing the amount of samples taken per pixel introduces a heavy performance penalty, because each sample is collected with a recursive computation. Many techniques have been introduced to address this performance issue by either altering the path tracing algorithm itself or by post-processing the image. Multiple importance sampling is a method introduced by Veach [Vea98] based on the observation that uniformly distributed random samples of shadow rays produce high variance (noise) in the image. The solution they propose is to use two different probability density functions (pdf) to generate two random bounces—one for directly sampling lights and one for *indirect illumination*—at each intersection.

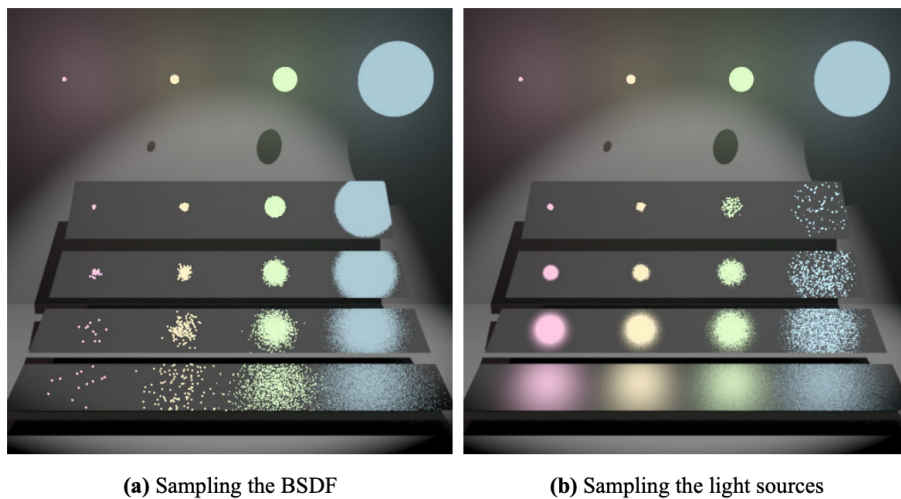


Figure 2: A comparison of the two improvements on the path tracing algorithm [Vea98]. (a) Sampling the bounces using the BSDF of the surface to simulate material properties. (b) Explicitly sampling the light sources by shooting rays towards their direction

Figure 2 demonstrates the mentioned improvements to the path tracing algorithm. The noise in sparsely sampled images is reduced significantly by multiple importance sampling in Figure 2 (b).

### 2.1.3. Acceleration Structures

Each ray in a path tracer is potentially tested against scene geometry for intersection. For practically every scene but the most trivial ones, testing every ray against every primitive in the scene geometry introduces a severe performance penalty. The algorithmic cost of ray-scene intersection is  $O(n)$  in a scene with  $N$  primitives. In order to make path tracing applications real-time, *acceleration structures* that compartmentalize the scene geometry into hierarchical boxes are used. Acceleration structures improve the algorithmic complexity of scene intersection to  $O(\log N)$ .



These structures typically organise the geometry into a defined hierarchy. The principle of acceleration structures is by traversing this hierarchy—which is usually a tree-like structure—from top to bottom, reaching only relevant primitives. The scene subdivision may either be spatial or object based. *Grids* and *KD-trees* are spatial subdivision techniques, whereas *bounding volume hierarchies* (BVH) are object subdivisions.

One of the most straightforward implementations of acceleration structures is a *grid*. A grid uniformly divides the 3D space of the scene into equal-sized cubes called *voxels*. Although it is very fast to build grids, they do not adapt to local geometric complexities. Therefore some voxels contain large amounts of primitives whereas some voxels are empty. To address this issue, the KD-tree was introduced, that subdivides space based on a *surface area heuristic* (SAH). The SAH is responsible for minimizing the cost to split a plane and therefore finding out the best plane to split the current node. The cost of splitting the current node into nodes A and B are represented by the following cost equation introduced by MacDonald and Booth in their paper called "Heuristics for ray tracing using space subdivision" [MB90].

$$C(A, B) = t_{traversal} + p_A \sum_{i=1}^{N_A} t_{intersect}(a_i) + p_B \sum_{i=1}^{N_B} t_{intersect}(b_i) \quad (2)$$

Where  $t_{traversal}$  is the cost of traversing a node,  $p_A$  and  $p_B$  denote the probabilities that the ray passes through the proposed A and B nodes. This probability is proportional to the surface area of the node.  $N_A$  and  $N_B$  are the primitives in A and B. The sums are summing the costs of intersecting every triangle in the respective volumes. SAH can be used in many acceleration structures including spatial and object-based acceleration structures.

Spatial subdivision algorithms fundamentally suffer from either not sufficiently adapting to geometric complexity (in grids) or the possibility that a primitive may reside in multiple voxels (KD-trees). BVHs were introduced by Turner Whitted to address the issues with spatial partitioning [RW80]. The BVH typically consists of a tree that stores the bounding box of the scene as its root node. The leaf nodes store the bounding box of a small amount of primitives. The interior nodes are recursively divided by minimizing the cost described in Equation 2. A BVH generally is a binary tree, although higher branching factors are also possible, such as the MBVH that contains four nodes at every level. A dense BVH like this is more efficient with vectorized traversal.

## 2.2. Case Study: LightHouse

The software product that we will operate on for our experiments is a production quality renderer called LightHouse, built at Utrecht University. The renderer was built with the primary aim to be used in commercial online configurators for products such as baby strollers, furniture and clothing. In the primary use case, the renderer is responsible for producing high-quality images of the configured product in a few seconds. Figure 3 demonstrates two scenes consisting of various materials and complex geometry rendered with LightHouse.

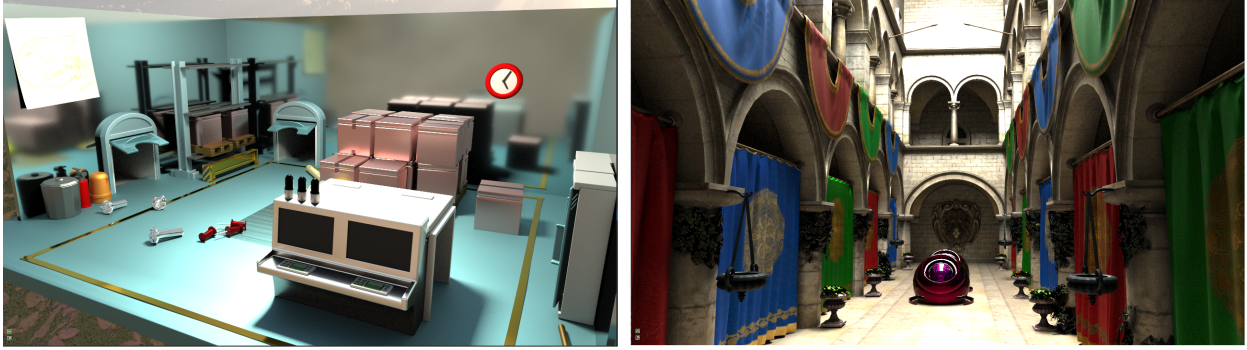


Figure 3: Two images rendered with the GPU core of LightHouse. The left image shows the Pica Pica scene, and the right image displays the Sponza scene.

The host environment for LightHouse is Microsoft Windows, running on x64 CPUs. This rules out Linux, ARM CPUs and mobile environments. The software architecture of LightHouse consists of three layers: application layer, render system and render core. The purpose of separating these tasks into distinct units is clarity, extensibility and increasing the portability by isolating the path tracing algorithm into a single software unit: the render core. This functional separation enables LightHouse to target multiple hardware architectures, most significantly the CPU and the GPU. The range of targetable architectures is practically infinite given that the reference render core is portable to the target architecture.

The data of the scene is divided into two parts as *host side* and *device side* data. These two different representations of the same data are required to target different hardware by preparing the device side data specific for that particular hardware. The device that is used to render in the primary render core of LightHouse is a NVIDIA GPU that supports the NVIDIA OptiX API.

### 2.2.1. Application Layer

The application layer is responsible for setting up the render target, usually a 2D texture consisting of pixels that displays the output of the rendering, and handling user input. The user input determines the location of the camera in the scene, and along with render commands, these are passed to the render system to be processed. This layer is located on top the render system and the render core respectively in the layer stack, and only communicates with the render system. Therefore, the render core is abstracted away from the application layer, making the application agnostic in nature about the architecture it is running on.

### 2.2.2. Render System

The render system owns the host-side representation of the scene, including objects, lights, camera(s) and animation. It is responsible for importing the scene data and converting it to a format that the designated render core can read and process. The data representation in this layer is the base reference for the scene and depending on the core it will be converted

to the respective device-readable format. The application layer interface is responsible for sending the current camera location and render commands to the render system.

### 2.2.3. Render Core

The render core is abstracted away from the application layer and only communicates with the render system layer through the render core API. This layer owns the device-side representation of the scene data. It implements the path tracing process using an appropriate ray tracing API for the rendering device. The device may be a GPU or a CPU of varying architectures and vendors. In order to support rendering in both, different implementations of the render core will exist in the renderer, including the one implemented in our research.

```
class CoreAPI
{
public:
enum CoreID
{
    MAIN_CORE = 0,
    PORTED_CORE_1,
    PORTED_CORE_2,
    RASTERIZER
};
static CoreAPI* CreateCoreAPI( const CoreID id );
virtual CoreStats GetCoreStats() = 0;
virtual void Init() = 0;
virtual void SetTarget( ... ) = 0;
virtual void Setting( ... ) = 0;
virtual void SetTextures( ... ) = 0;
virtual void SetMaterials( ... ) = 0;
virtual void SetLights( ... ) = 0;
virtual void SetSkyData( ... ) = 0;
virtual void SetGeometry( ... ) = 0;
virtual void SetInstance( ... ) = 0;
virtual void Render( ViewPyramid v, Convergence c ) = 0;
virtual void Shutdown() = 0;
};
```

Listing 1: The LightHouse render core API.

The `CoreAPI` that each render core is responsible for implementing is presented in the Listing 1 as an abstract class. It is important to note that although the primary core implements the path tracing algorithm, the API does not restrict the usage of other rendering algorithms. The `Render` function takes as input the view pyramid as shown in Figure 4 that defines a camera view. The second argument, *convergence*, determines whether to accumulate samples or render every frame from scratch. Thus, this argument is only relevant

for path tracing cores.

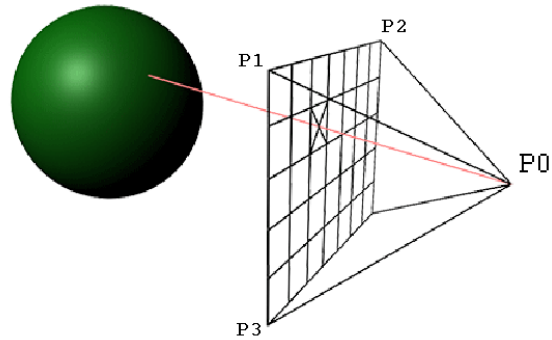


Figure 4: The view pyramid that defines a camera view. P0 marks the position of the eye. P1, P2 and P3 respectively mark the top left, top right and bottom left corners of the screen.

The **Init** function is called once during application launch, and it is intended for core specific initialization. Contrarily, the **Shutdown** function destroys the render core and frees all resources. The functions that are prefixed with **Set** encapsulate the data transfer between the render system and the rendercore.

## 3. Literature Review

In this chapter we introduce key concepts of *software portability* and *efficient CPU path tracing* with a literature review that summarizes the theoretical background up until to the state-of-the-art. We start, in Section 3.1, with an overview of software portability in general and narrow down the implications and practices to the use case of this thesis project. We evaluate the proposed metrics of portability and how portability of a software unit can be measured, followed by a survey of efficient CPU path tracing principles in Section 3.2.

### 3.1. Software Portability

The concept of software portability varies in meaning for different industries and use cases. The task of porting software may refer to anything between physically moving and installing software to another computer, and completely rewriting the source code of a program to make it work on a different platform. The different types of tasks required to port software create the levels of porting. The degree of effort made in order to port software is a measure of how portable the software is. Portability is a non-functional software requirement in the broader field of systems engineering because portability concerns itself with the software architecture rather than the logic. Ideally, the software logic is intact in a ported program.

#### 3.1.1. Early influential work

In his article "Guidelines for Software Portability" (1978), Andrew S. Tanenbaum proposes an initial formal definition of portability with respect to programming languages, device characteristics, machine architecture and documentation [TKB78]. Tanenbaum proposes the guidelines for creating portable programs based on the following definition of portability, written a year earlier by Poole et al [PW75]:

"Portability is a measure of the ease with which a program can be transferred from one environment to another; if the effort required to move the program is much less than that required to implement initially, *and* the effort is small in absolute sense, then that program is highly portable."

It is important to note that the case of rewriting the program from scratch is not excluded from this definition. A common pitfall indicated by Tanenbaum, that is relevant for this project is the difference in machine architecture between the *host* and the *target* machine architecture. In his context, the system that the initial development was made for is referred to as the host machine, and the system that the program is desired to be ported to is called the target machine. He states that the architecture of the host machine influences the program to be ported and this influence may either be explicit (choice of application programming interface) or be implicit (the algorithms that are chosen) [TKB78]. In another relevant insight, the article states that "optimality is not portable", but portability does not necessarily imply inefficiency. In other words, a program may be hand-tuned to the distinct

attributes of a given system, thus making it slow to the point of being useless on different systems that it could be run on.

### 3.1.2. Relevant Methods and Techniques

Although there exists no commonly agreed upon definition of software portability, a working definition proposed by James D. Mooney will be used due to its focus on *source portability* [Moo90]:

”A software unit is portable (exhibits portability) across a class of environments to the degree that the cost to transport and adapt it to a new environment in the class is less than the cost of redevelopment.”

A *software unit* is a term that could refer to an entire application, a component of a program, or just a single file or class in the program. For the purposes of this study, a software unit is seen as an abstraction layer, such as the render core presented in Section 2.2. A smaller, specialized software unit that resides in the render core is the shading code described in Section 5.4. *Environment* refers to everything the software is interacting with through auxiliary interfaces. Examples of environment components are operating systems, processing units, networks and software libraries. The concept of environment is interchangeable with the notion of *platform*. The highly relevant environmental variable in this project is the processing unit, where the GPU is the native environment and the CPU is the target environment for the porting exercise. The term *class* of environments is used to denote the entire range of possible components that the target environment has, rather than listing the entire variants of that particular class that fits a certain criteria (e.g. a list of all CPUs that support SSE instruction sets).

Mooney notes that portability is not a binary attribute and that each software unit has a quantifiable *degree of portability* to a particular class of environments. This degree of portability is based on the cost of porting, and it is not absolute; it only has a meaning with respect to a specific class of environments. There is a defined set of costs and benefits associated with portable software development and these costs and benefits take a variety of forms that will be discussed later.

In addition to the degrees of portability, there are various *levels of portability*. Based on the assumption that software goes through multiple representations through its lifecycle. Generally, the representations move from high-level to low level, between a human-readable format and an executable in machine language. The three most common levels of portability have been listed by Mooney in his article, *Developing Portable Software* [Moo04]:

- *Source portability*. It is the most common level of portability. The software is ported from a human-readable high level representation by the manual adaptation of a software developer. The ported version is then compiled for the target environment. This level of portability is the one practiced in the scope of this research.
- *Binary portability*. Used when the high level representation is inaccessible, or should remain intact. The executable is ported directly to a new environment. Although it

requires less effort than source portability, it usually applies to a small amount of cases and environments.

- *Intermediate-Level Portability.* In some cases, a software representation between source and executable binary could be ported. Java bytecode and assembly language are examples of such an intermediate representation layer.

The process of porting can be divided into two tasks; *transportation* and *adaptation* according to Mooney [Moo04]. Adaptation refers to the phase in which the source of the program is being modified, either manually or by a form of automated translation. Transportation is more about the physical movement of the software and its artifacts, and is more relevant on software developed for embedded systems. Finally, the terminology Mooney presents defines *redevelopment* as the alternative to porting. It refers to developing a program from scratch based on the original software specification. The comparison between redevelopment and porting is the most essential part of the limited body of work on software portability.

In the design phase of a program, several issues need to be addressed to reach a high level of portability. The main effort in this stage is coming up with a design such that the effort put in this stage makes the program easily portable to different platforms. Mooney proposes four questions that aims to divide the portability discussion into smaller issues [Moo01]. Answering these software specific questions establishes a close analysis of the project:

1. For what class (or classes) of environments should future portability be considered?
2. What degree of portability is desired for various environments in these classes?
3. What extra development costs, if any, are acceptable in order to achieve these portability goals?
4. What reduction in the quality of implementations, if any, is acceptable to achieve the desired portability?

The first question defines a certain set of environments that could be targeted, therefore imposes *constraints* on portability. The second question reveals the potential *benefits* in the form of improved attributes due to the final design of the system. The remaining questions are useful in displaying the potential *costs* of portability. The costs expected here can be in two different forms, either due to increased effort in the initial development or reduced quality in the end product.

It is important to note that the questions raised above are relevant to the initial development stage of software. If portability analysis was not done and the software was not designed with portability considerations, the cost porting may be greater than the cost of redesign or redevelopment [Moo00]. In order to design for portability, following some design strategies are necessary. These guidelines—presented in the paper *Issues in the Specification and Measurement of Software Portability*—are also applicable to existing software such as the case study of this project, so that they can be used to enhance a software towards being more portable.

1. Identify specific environment interfaces required (procedure calls, parameters, data structures, etc.). For each interface, either:
2. Encapsulate the interface completely in a suitable module, package, object, etc. Anticipate the need to adapt this interface to each target system; or:
3. Identify a suitable *standard* for the interface, which is expected to be available in most target environments. Follow this standard throughout the design. Anticipate the need to provide a software layer to bridge the gap for environments which do not implement the standard satisfactorily.

These guidelines are highly relevant for increasing the source portability of particular software units such as the render core described in Section 2.2.3.

An example for the third element in the list above could be the selection of a suitable programming language. For instance, choosing a language that has fewer facilities to isolate functional units would hinder these design goals.

Examples to the portability design goals stated above have been presented by Ian Sommerville in his textbook, *Software Engineering*, as two different architectural design patterns: Model-View-Controller (MVC) and *layered architecture* [Som11]. The MVC design pattern isolates presentation and interaction from the data. It is typically used in web-based environments because its purpose is to support multiple ways to interact with the same data.

*Layered architecture* refers to separating the system into layers with specific functionality associated with each layer. Typically, each layer communicates with the layer above it through interfaces. The lowest-level, therefore, typically encapsulate core functionality, hence the term render *core* in Section 2.2.3. Garlan et al. state that *reuse*, such as the easily re-implementing an abstracted layer is a very significant benefit of layered architecture [GS93]. Just as in abstract data types, different implementations of the same layer can be used interchangeably, given that they agree on the interfaces to adjacent layers. This idea is paramount to our research, and interchanging layers—possibly even at runtime—is one of the important goals of this project.

In a study of portability as a non-functional quality attribute, Kazman et al. report that *separation*, *abstraction* and *resource sharing* affect portability favourably, while *compression* and *replication* are hindering factors [KB94]. The unfavourable attributes typically promote performance. Resource sharing refers to allowing multiple components of the system to use the same data representation. An example of this could be seen in the scene data in the render system, which can be used by multiple render cores (Section 2.2.2). This may seem to introduce a trade-off between performance and portability, however as Mooney states, it is possible avoid the trade-off as long as initial design was made with the discussed portability principles [Moo90].

An underestimated benefit of porting has been demonstrated by Ray et al., in a case study of cross-system porting of forked projects [RK12]. By applying an automated software analysis tool to 18 years of BSD family software, they compared the quality of ported edits to initial development commits. Their findings indicate that ported code is usually less error-prone than non-porting changes. This finding suggests that either programmers are more likely to selectively port well-established features from other projects, or that porting



results in bug fixing.

### 3.1.3. Portability Metrics

In order to assess the portability of a software unit, several methods have been proposed. Metrics are useful in justifying design decisions to project managers, and demonstrating the outcomes of the trade-offs. In his initial work, Mooney proposes the following conceptual entry point for the portability metrics [Moo00]. The main assumption is that the degree of portability of a certain software unit should be greater than zero ( $DP > 0$ ).

$$DP = 1 - (\text{cost of porting})/(\text{cost of redevelopment}) \quad (3)$$

In his later work, this definition is expanded using more detailed terminology. *Process metrics* are used to measure development and maintenance processes, and these quantify the costs of these activities. According to Mooney, process metrics may denote expenses in dollars, person-days, work hours, or lines of code [Moo01]. For the purposes of this study, development cost activities will be measured both in lines of code and work-hours. A more formalized version of Equation 3 becomes:

$$DP(su) = 1 - \frac{C_{port}(su, e_2)}{C_{rdev}(req, e_2)} \quad (4)$$

The reasoning starts with the assumption to develop a software unit,  $su$ , for an initial environment,  $e_1$ , based on its initial requirements,  $req$ . This cost is represented by  $C_{dev}(req, e_1)$  and is based on four components for *design, coding, testing and debugging* and *documentation*:

$$C_{dev}(req, e_1) = C_{des}(req) + C_{cod}(req, e_1) + C_{td}(req, e_1) + C_{doc}(req, e_1) \quad (5)$$

It is important for portability to note that all the components, except the design phase that is ideally independent of the environment, depend on the target environment. Given the development cost above, the cost to redevelop a software unit, based on the original requirements,  $req$  for an environment  $e_2$  becomes:

$$C_{rdev}(req, e_2) = C_{rdes}(req) + C_{rcod}(req, e_2) + C_{rtd}(req, e_2) + C_{rdoc}(req, e_2) \quad (6)$$

The cost to port the same software unit for the target environment  $e_2$  does not have a design component, because the design is intact. It consists of the manual modification, testing and debugging, and documentation:

$$C_{port}(su, e_2) = C_{mod}(su, e_2) + C_{rtd}(req, e_2) + C_{rdoc}(req, e_2) \quad (7)$$

It is expected that the porting costs in the testing and documentation components are less than redevelopment costs. If portability is achieved in the design phase, it is also expected that  $C_{mod} \ll C_{rdes} + C_{rcod}$ , and this is the greatest benefit achieved by portable design.

In their paper *A Study of Software Portability Evaluation*, Hakuta et al. propose a method for evaluating porting cost based on what they call *porting impediment factors* [HO97]. These impediment factors determine whether program modification is necessary for porting. They include system-environment disparity, which is the difference between source and destination environments, and program factors, such as to which extent portability was taken into account during the initial development. These factors are weighted and summed together to form the *portability impediment index*,  $\alpha_p$ .

On the other hand, porting *cost factors* are external factors such as human factors and environmental factors. Human factors refer to the knowledge and experience of the developer doing the porting, whereas the environmental factors denote software development tools, test cases and workstation environment. Together with portability impediment index, the porting cost in total man-hours,  $Y$  is found by the following formulation [HO97]:

$$Y = C \cdot \prod_{i=1}^n 10^{\beta_i \alpha_i} \cdot X^{\beta_s} \quad (8)$$

where  $X$  denotes the program source code size in kilobytes, and  $C$ ,  $\beta_i$ ,  $\beta_s$  are constants obtained by regression analysis, and can be taken from a look-up table. The number of factors that effect portability,  $n$  is 3 and the factor set is defined as follows:

- $\alpha_1 = \alpha_p$  (Portability impediment index)
- $\alpha_2 = \alpha_H$  (Human factors)
- $\alpha_3 = \alpha_E$  (Environmental factors)

Considering the generality of Mooney’s approach, it is easier to adapt it to the case study we have with this thesis project and it will be used in the porting experiment.

### 3.2. High Level CPU Path Tracing APIs

Recent increases in the compute capacity in consumer grade CPUs due to Moore’s law and simpler functional unit replication allowed for graphics engineers to exploit this capability for real-time ray tracing purposes. However, utilizing the full compute capacity of a CPU with an algorithm that exhibits data-dependent branching, and irregular memory access patterns is nontrivial [WSB<sup>+</sup>14]. Moreover, different architectures require different low-level optimizations, adding another level of complexity to the task of producing a renderer that can target multiple architectures and workloads.

To alleviate these challenges, companies such as Intel, NVIDIA and Microsoft have made the following observations and acted upon them to produce high-level ray tracing frameworks:

- Rendering software in general can be built from a small group of specialized software units [WSB<sup>+</sup>14]. These units are common building blocks of the ray tracing pipeline and encapsulate routines such as acceleration structure building, traversal and ray casting.

- CPU architectures were built for data-dependent branching [WSB<sup>+</sup>14]. Ray tracing algorithms typically exhibit this kind of behaviour in the acceleration structure traversal step. The challenge with the CPUs however is achieving high throughput.
- There is a lot of headroom in the performance of state-of-the-art ray tracing applications that exhibit only interactive frame rates (1-10ms). The ideal target performance of such applications would need to reach the real-time realm (30-60ms) to become eligible for the use of the gaming industry.
- The recent research (e.g. single ray vectorization and packet tracing) on the field has not yet made its way into industry’s existing renderers. This is mostly due the new methods being highly specialized, thus requiring deep domain knowledge to implement.

### 3.2.1. Embree

Embree was introduced in 2012 and open-sourced in 2014 as a ray tracing framework for x86 and x64 CPUs. The observations made above make Embree a suitable candidate for use in our renderer. Specifically, Embree is a suitable candidate API for this project because it is a product specifically targeted at ”professional rendering workloads” [WSB<sup>+</sup>14]. What is meant by professional workloads is that they include large models and have a combination of primary rays with high spatial coherence and secondary rays with low spatial coherence. Large models in this context refer to scenes with millions of triangles, including the type of scenes that do not fit into GPU memory.

The spatial coherence of rays mentioned above refers to ray coherence concept explained by Ohta et al. [OM87] as rays having nearly the same origin and nearly the same direction. Rays that are spatially coherent have higher probability to intersect with nearly the same object. On the contrary, rays with low spatial coherence tend to diverge into different directions due to bounces from rough surfaces or complex geometry, causing highly irregular data access and low data locality. It is typically the *shadow rays* and *secondary rays* that exhibit low spatial coherence because they are dependent on geometry, whereas the primary rays are generated from the camera, through the pixels, therefore inherently having high spatial coherence.

Other examples of high-level ray tracing APIs include OpenRT [DWBS03], OptiX by NVIDIA and Radeon-Rays by AMD. OpenRT aimed to provide an interface for both the application and core services of a typical renderer. However, it was discontinued and therefore ineligible for use. DX12 is developed by Microsoft to run on GPU, however is not well-documented and still experimental. It could be seen as a weaker alternative to OptiX maintained by NVIDIA [PBD<sup>+</sup>10]. It is also known that OptiX runs on the CPU and a study conducted by Andr Bico compares Embree with OptiX on a global illumination algorithm [? ]. Although their performance is comparable, Embree provides advantages over OptiX such as using the same code base as any C/C++ application and the ability to apply its features to an already existing CPU renderer and obtain performant code. Finally, Embree

is deemed "the best solution if the goal is to achieve the fastest performance without many development considerations" [? ].

# 4. Research Methodology

## 4.1. Porting and Implementation

In order to conduct experiments on performance, reliability and portability, our primary goal is to port the reference render core—currently working on the GPU—to a version that renders using the CPU. The ported render core will be used as a framework for conducting the experiments explained in Section 4.2. A detailed explanation of the porting process is given in Section 5.

## 4.2. Experimental Setup

To answer the research questions proposed in this thesis, several experiments will be conducted using the ported core described in Section 4.1. There are two broad categories of experiments. We will start with an assessment of performance and quality. After this, we will investigate portability of the CPU fallback. The following sections will expand on the experimental setup designed to test these attributes.

Performance evaluation for rendering systems with a tightly matching set of features has to be conducted with similar scenes from similar viewpoints. The performance of the renderer is highly dependent on the type and the size of the geometry. Therefore, commonly acknowledged computer graphics scenes such as the Pica Pica, Sponza and Amazon Bistro will be used. These workloads are chosen because they are representative for professional rendering environments.

### 4.2.1. Rendering Very Large Scenes

We aim to answer **RQ1** by investigating the viability of the ported CPU render core in large scenes. In order to test the reliability of the CPU fallback, a scene that is larger than the device memory of the GPU will be rendered using the Embree Core we have implemented. A survey of modern consumer grade GPUs from the two top-grossing GPU vendors NVIDIA and AMD indicates a memory range of 4-6GB's on devices manufactured in 2018 (AMD Radeon RX 550X and NVIDIA GeForce GTX 1060 respectively) [nvi][amd]. The GPU platform used in our tests is a GeForce GTX 1060 with 6GB of memory, which will not be able to render a scene that exceeds 6GB using the GPU render core of LightHouse.

A scene that exceeds 6GB can not be rendered by the primary render core of LightHouse. We will use a version of the Powerplant scene with an amplified number of triangles that does not fit into the GPU device memory but fits into the 16GB RAM of the system. The system is expected to produce images of the scene at a very low framerate, however if the resulting frames are being drawn on screen, the fallback core will be deemed successful, because these images are not otherwise producible using the GPU render core of LightHouse.

In order to verify that the Embree core has similar functionality to the reference primary render core, we will setup an additional experiment. Three scenes that fit both into CPU

and GPU memory will be rendered from identical viewpoints using both render cores. The resulting images will be compared. It is expected that similar output is produced. Images produced after 1, 10, 100 and 1000 samples from each render core will be compared. The comparison will be done using the Root Mean square Error (RMSE) and Structural Similarity Index metrics [WBSS04].

#### 4.2.2. Performance Evaluation

To answer **RQ2**, the Embree core will be tested against the primary render core for performance. The two cores will be tested against different workloads and measurements will be made on the performance of the key functional blocks of the rendering pipeline.

The performance critical stages of rendering that will be measured are *BVH construction* and *render performance* in rays per second. The BVH construction is typically done once for static scenes, therefore it is only performance critical in dynamic scenes where the BVH needs to be updated every frame. The measurements will be made in milliseconds for three different scenes with increasing complexity.

The render performance will be evaluated in rays per second including the time it takes for ray traversal, shading and sampling. The same three scenes used for *BVH construction* will be used. Four sets of measurements will be made including one set of measurements with diffuse-only shading using both render cores. Through this experiment, the computational cost of shading in both cores will be isolated and conclusions will be drawn from the performance characteristics of the Embree render core. This way, possible improvements as well as possible use-cases for the renderer will be investigated.

#### 4.2.3. Portability Measurement

We will conduct a practical programming experiment to measure how portable the resulting software architecture of LightHouse renderer is. The experiment setup will be as follows. After completing the implementation of the Embree render core, a new functionality will be introduced to the primary render core. This new functionality is tentatively selected as a new bidirectional scattering distribution function (BSDF). Adding a new BSDF to the system is typically done to support more advanced or different materials, and it is one of the most common functional additions to professional renderers.

The new BSDF introduced to the primary render core needs to be ported to the Embree render core. The porting process will be measured in various aspects, including the porting effort in total number of lines of code, as well as the percentage of unique code produced. Mooney’s portability analysis presented in the Portability metrics will be used to quantitatively analyse this process, as well as reporting qualitative observations and findings.

# 5. Implementation

In this chapter, the implementation of the CPU based render core is presented. The primary goal in the implementation phase is to create a CPU based render core, that produces identical results to its GPU counterpart. Qualitative findings and observations from the development stage will be reported throughout this chapter in detail. The results and the validation of this intention are presented and discussed in Chapter 6.

The implementation will be explained in the following sections. Section 5.1 gives an overview of the architecture and system design of LightHouse and the design decisions made with a focus on portability. Section 5.2 presents our approach in handling the scene data received from the LightHouse render system. Next, in Section 5.3, the ray generation routines, together with the wavefront path tracing algorithm and its implementation in our render core will be demonstrated. Finally, our ported implementation of the reference shading code will be introduced in Section 5.4.

## 5.1. Architecture and System Design

The approach to implementing the CPU-based render core is based upon the *source portability* principles stated in section 3.1.2. In order to distinguish the two render cores participating in the porting effort, we will establish a source and target relationship between them. The GPU-based render core will be referred as the *source core* (the original render core that is to be ported), whereas our ported CPU-based implementation of it will be called the *target core*.

We start the software design by examining the initial portability considerations made during the engineering of the source core. The answers to the four questions raised by Mooney in Section 3.1.2 that enable a close analysis of the source core are as follows:

1. With its initial design, the source core is implemented with C++/CUDA and targets NVIDIA GPUs. Future portability was anticipated for the CPUs, GPUs by other vendors such as AMD, and web browsers through the utilization of WebGL.
2. The degree of portability planned for these environments was source-level. Due to CUDA being a hardware specific API, it is not possible to cross-compile the source codebase to target architectures, nor is it possible to run the executables of the source core in different architectures.
3. It was anticipated that each new aforementioned target core, could cost at least two-full months of development work of a single porting engineer.
4. The reduction in the quality of implementations for the sake of portability was not applicable to the source core, as it was highly optimized for high performance real-time rendering.

In the light of these findings based on the existing software, we have conducted a portability analysis based on the three guidelines proposed by Mooney and explained in Section 3.1.2.

1. The interface that enables the portability of the render core is the **CoreAPI** between the render system and the render core.
2. The interface is encapsulated in an abstract class. Each unique render core is responsible for implementing it.
3. The *standard* for the interface is set with the function signatures. The interface functions represent only the core rendering tasks found in Section 2.2.3.

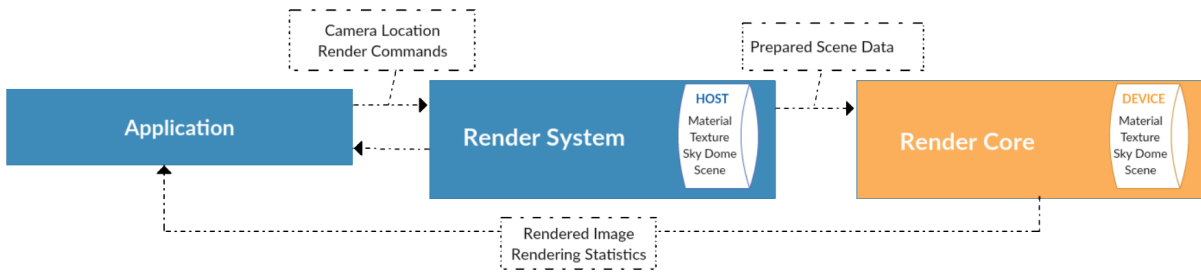


Figure 5: An overview of the main software components of LightHouse. Dashed blocks contain the type of data passed through these interfaces. Render system and core(s) contain their own copies of scene data.

The entry point to implementing the target render core is readily provided by the **CoreAPI** interface between the render system and the render core. The render system is core agnostic, meaning it does not impose any implementation restrictions or assumptions. Synchronizing the scene data and answering the **Render** commands issued by the application is the key responsibility of the render core. Thus, we have taken the OptiX Prime core as our source core and focused our porting effort on replicating its functionality by implementing the requirements of the **CoreAPI**.

## 5.2. Geometry Loading

In order to serve a render command issued from the application layer, our target render core needs to hold its own representation of the scene. The application layer is responsible for loading the specified scene into main memory. In its current state, the application layer supports an open-source file formats; *glTF*, and two proprietary formats; *obj* and *fbx*. The render core, on the other hand, is file format agnostic.

The representation of the scene that lays in the render system is referred to as the *host scene*, and its counterpart in the render core is called *core scene*. Other helper objects that contribute to the scene representation and also have dual representations are materials, textures, lights, sky dome and meshes. The host scene holds separate lists of these objects. The scene data (each scene element separately) is sent to the core through the **CoreAPI** as shown in Figure 5. Changes are tracked with flags in these objects, and scene elements that are changed (by some form of animation or removal) during runtime are sent to the render



core with a *synchronize* command. For static scenes, this transfer occurs only once. The render cores are responsible for implementing the receiving end of the scene data transfer, such that a core specific representation of the scene can be built.

The host mesh holds a double representation of a triangle mesh. It contains a list of vertices and indices storing the connectivity data. Although this much is enough to construct a mesh, the host mesh holds an additional list of triangle objects. Efficient intersection on both the source and target render cores require only vertices and indices. However, the shading code requires the triangle objects. These triangle objects store additional data required for shading such as vertex normals and material indices. Thus, in line with our research goals, we have opted to store both representations on our implementation of the geometry initialization. The vertex and index data are fed to Embree, whereas the triangle objects are stored in core for the shading explained in Section 5.4.

The logic of creating a scene from triangle meshes is consistent between OptiX and Embree, and thus, between the source core and the target core. A scene is an internal container for multiple geometries of potentially different types in both APIs. Both require vertex and index data to create internal geometry buffers, and both provide a command for triggering the use of geometry buffers to construct acceleration structures, hence, preparing the scene for intersection. Figure 6 presents a side-by-side comparison of the two APIs in order to demonstrate the logical similarity of loading geometry to both cores.

<pre> // create scene scene = rtcDeviceNewScene(device)  // add mesh to scene geomID = rtcNewTriangleMesh(scene)  // set Embree data buffers rtcSetBuffer(scene, geomID,              RTC_VERTEX_BUFFER, vertexPtr) rtcSetBuffer(scene, geomID,              RTC_INDEX_BUFFER, indexPtr)  // commit changes rtcCommit(scene) </pre>	<pre> // create scene rtpContextCreate(context)  // create OptiX geometry buffers rtpBufferDescCreate(context, INDEX,                    indexPtr, &amp;indicesDesc) rtpBufferDescCreate(context, VERTEX,                    vertexPtr, &amp;verticesDesc)  // create model rtpModelCreate(context, &amp;model) rtpModelSetTriangles(model,                     indicesDesc, verticesDesc)  // commit scene rtpModelUpdate(model) </pre>
---	--

Figure 6: A side-by-side comparison of the geometry loading phase of Embree and OptiX. The left side shows Embree and the target core, whereas the right side shows the source core on the GPU side. The code was deliberately changed to pseudo-code to emphasize the functional similarities.

### 5.3. Path Tracer

The source core implements the path tracing algorithm explained in Section 2.1.2 with robust techniques such as next event estimation and multiple importance sampling (MIS). The path tracing integrator is unidirectional, meaning the rays are issued from the eye only and not from the light sources. The integrator uses Monte Carlo sampling. In order to achieve real-time frame rates on its target architecture (NVIDIA GPUs), the source core implements the Wavefront formulation of the path tracing algorithm explained in Section 5.3.1. This means that the path tracing algorithm is decomposed into multiple tasks. These tasks are encapsulated in separate CUDA kernels, and data such as rays and intersections are passed between these kernels multiple times in a single iteration.

In order to keep the codebase as similar as possible, in accordance with the portability guidelines presented in Section 4.2.3, we have chosen to port this algorithm, making as few changes as possible, to our target core on the CPU side. Thus, kernels that are responsible for actions such as *ray generation*, *shading*, *occlusion* and *intersecting* were ported from separate CUDA files (kernels) to C++ functions. Figure 5.3 lists the kernels found in the source core.

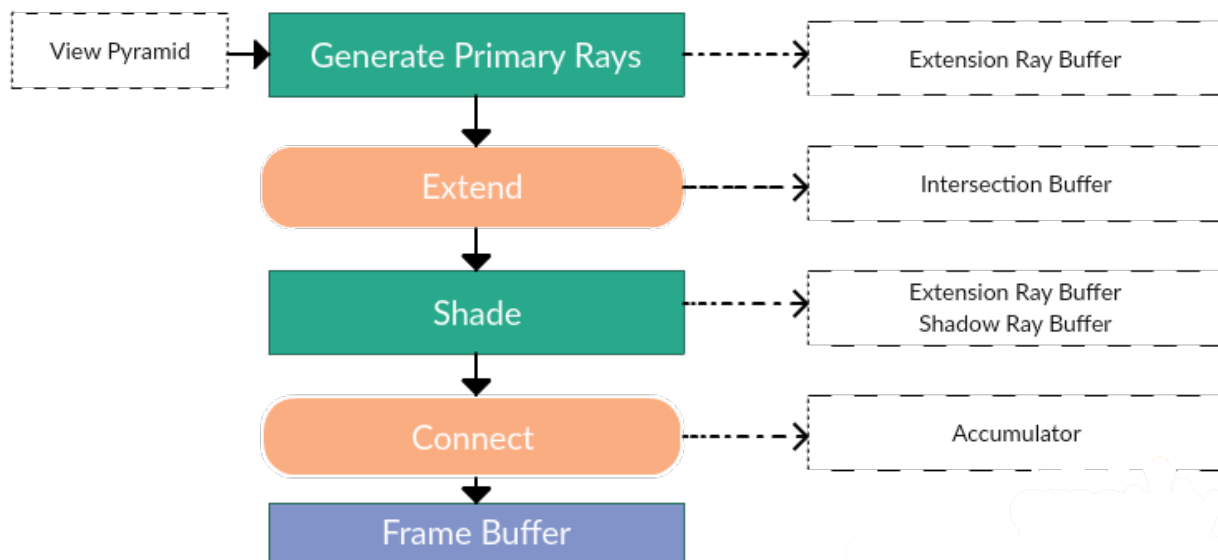


Figure 7: Path tracing kernels. Green kernels are implemented with CUDA in the GPU core(s), and ported to C++ in the CPU core. Orange kernels are implemented with Embree ray tracing kernels. The dashed boxes on the right sides of the kernels show where they write to.

The kernels that are shaded with orange in Figure 5.3 were implemented with OptiX Prime, a fast low-level API for tracing rays on GPUs. These kernels are essentially ray/primitive intersection tasks. The **Extend** kernel takes a ray or a batch of rays as input, and outputs the closest intersection point data that the ray has with the scene. The intersection data contains the distance to the eye, the ID of the triangle that was hit, and the u, v coordinates that denote a point inside that triangle. We have chosen Embree Ray Tracing

Kernels to use as the counterpart to Optix Prime in our target core, as it is capable of completing the same task on CPUs [WSB<sup>+</sup>14].

The OptiX Prime API call for this task is `rtpQueryExecute`, that executes a *query* of type `RTP_QUERY_TYPE_CLOSEST`. The participants of this query are the primary rays generated from the eye, or the extensions of these rays that bounce from the scene during the random walk, depending on the path depth. This kernel writes its output to a buffer of intersections. Our ported implementation of this kernel uses the Embree API call `rtcIntersect`. Similarly, the participants of this intersection are scene, intersection context, and the ray. The `rtcIntersect` function finds the closest hit for a single ray. If an intersection is found, hit distance is written into the `tfar` member of the ray and all hit data is set. When no intersection is found, the ray/hit data is not updated.

The **Connect** kernel is the second orange kernel in the source core that uses OptiX Prime. The kernel operates on a buffer of shadow rays, also known as occlusion rays. This shadow ray buffer contains rays that are spawned from the shade kernel during the random walk. They originate from hit points that the Extend kernel generates and are directed towards random points on lights. The shadow rays need to be tested for intersections against the scene before for occlusions writing the radiance throughput associated with them. If no intersection is found, the associated throughput is written to the accumulator. The source core implements this with another `rtpQueryExecute`, with a `RTP_QUERY_TYPE_ANY` flag. The target core implements this query with a `rtcOccluded` call of the Embree API.

### 5.3.1. Wavefront Formulation

In order to maximize thread utilization of the source core, the path tracing algorithm was implemented with the *wavefront* approach proposed separately by Laine et. al [LKA13] and van Antwerpen et al. [Ant11]. This means that the entire path tracing workload is separated into highly specialized kernels, and the data buffers that are passed amongst these kernels (rays, intersections and throughputs) are compacted by keeping atomic counts on active rays. The kernels can be seen in Figure 5.3, and the buffers that they write to can be seen in the dashed boxes to their right.

CPUs and GPUs have different characteristics, and the wavefront scheme is specific to GPUs. It involves data passing at the start and end of each kernel, which is unnecessary for the CPUs. The data buffers used to pass data between kernels in the source core are `CUDABuffer` objects. These buffers have either a host or a device pointer for the data they point to, and implement aligned resource allocation. Since CUDA is not a dependency in our target core, we opted to implement our `CoreBuffer` objects to store ray, intersection and throughput data. Our buffers only encapsulate a single pointer to the aligned data that they point to, since there is no separate device memory for the CPU. We have chosen to mimic the spots where data is passed from the GPU to the CPU or vice versa by merely passing pointers to functions. Thus, we have kept code similarity high, and avoided unnecessary data transfers.

Interestingly, programming with a SIMD (Single Instruction, Multiple Data) execution

model has gained traction in the recent years, especially in performance critical rendering tasks, where data parallelism can be exploited. A recent example of this in a proprietary software is *MoonRay*, developed by *DreamWorks Animation* in 2017, that is the first production path tracer that fully leverages SIMD vector units [LGXT17]. The increase in SIMD adoption is partially due to the release of Intels ISPC (Intel Single Program multiple data Compiler) compiler [PM12]. This compiler draws heavily from GPU programming languages, and enables automatic parallelization of scalar code, by distributing it over multiple SIMD lanes.

CPU based renderers written with ISPC, therefore, also suffer from control flow divergence as per their GPU counterparts. Although the penalty may be lower for CPUs due to narrower SIMD, the wavefront approach should be useful if our implementation is vectorized in the future. In its current state, the the acceleration structure construction and ray intersections are vectorized internally by Embree. However, ray generation, integration and shading phases are scalar. Vectorizing these processes as ispc kernels is a large prospect into render time speedups. Lee et al. report 2x with SSE4.2 (4 wide) and 2.5x with AVX2 (8 wide) instruction sets for total render time speed-ups in the vectorization of *MoonRay* [LGXT17].

### 5.3.2. Buffers

Due to the wavefront programming model explained in Section 5.3.1, intersections, rays, and shading data associated with rays, need to constantly be passed between kernels. We refer to the containers for these data as path tracing buffers. The source core leverages generic programming to implement these buffers. The class template is called `CUDABuffer`. This container has dual pointers for the data it stores; a pointer for the device and another one for the host. Because it uses CUDA calls to acquire device resources and to copy data back and forth between host and device, it is not possible to use the same container for the target core.

We have opted for porting the buffers to the CPU side by writing a similar template class called `CoreBuffer`. Our `CoreBuffer` implementation only acquires resources on the host side, and owns a single data pointer. It is favourable to use our own implementation, for instance, rather than a STL container, because it allows to make minimal changes in the kernel function signatures. More importantly, using these buffers allows for much more similarity between the source and target core implementations of the `CoreAPI` functions explained in Section 2.2.3. The buffers that contain scene data (such as material and texture buffers) are initialized and populated in the with `CoreAPI` calls from the render system. On the other hand, path tracing buffers are initialized once per render target, with compile time sizes that reflect the maximum amount of active rays.

## 5.4. Shading

The **Shade** kernel is responsible for processing the intersection data produced by the **Extend** kernel. The source core uses a programming model called *persistent threads* [AL09]. This means that the kernel uses a work queue to keep every Streaming Multiprocessor (SM) busy. The size of the this work queue is the amount of active paths in that particular iteration of the random walk. A function "hosts" the shade kernel, and runs until there is no more active paths available. This programming model does not translate to the CPU environment where there are no Streaming Multiprocessors at hand.

Porting a parallel algorithm directly to a parallel implementation, especially when the target architecture is vastly different, would have been error prone. Thus, we have opted for calling the Shade kernel sequentially for every active path. It would also significantly increase the porting cost by introducing an additional engineering challenge, which we strive to minimize. This way, the emphasis on correctness and similarity is conserved. The persistent threads syntax is replaced by a job index that keeps track of the extension ray index currently being shaded. This means that the host function that implements the persistent threads is effectively replaced by a loop that increments the job index.

The Shade kernel reads from and writes to several buffers. It reads from the extension ray, path state data and intersection buffers. After executing the shading code, the Shade kernel writes to the extension rays output, path state data output, shadow rays and shadow ray potentials buffers. The shading code contains a high amount of control divergence, hence, execution of the shading code may also not result in any writes to any of these buffers. Therefore, counters are essential to keep track of the amount of active extension and shadow rays. Essentially, we have ported the persistent threads implementation on the GPU, to sequential loops that use these counters to perform the correct amount of shading tasks.

This kernel writes to the extension ray out buffer with a randomly bounced ray, until the max path length is reached. Additionally, a shadow ray towards a random point on a random light is initialized and written to the shadow ray buffer. Due to stream compaction discussed in Section 5.3.1, the kernel contains atomic counters. Atomicity is not a concern in sequential CPU code where no race conditions exist. It is important to note that the buffers do not provide a size query, hence, we implemented a static counter object to keep track of the ray counts.

Additionally, the shade kernel calls other device functions that reside in different CUDA source files. The CUDA source files are grouped into functional groups, into files that contain functions regarding, materials, texture sampling and lights. It is not possible to compile these files with a C++ compiler, and it is not trivial to port these functions by simply adapting the function signatures to C++. These functions constitute the most non-portable component of the source core, such that, they contain intrinsic math functions, that have hardware support, and available only through the CUDA API. The intrinsic functions are documented, but the source code is inaccessible. We have opted for re-implementing these intrinsic functions from scratch. Listing 2 shows our implementation of an intrinsic function following a specification taken from the CUDA API. The code that was produced

with a reverse engineering effort as such, are considered entirely novel code blocks in the portability analysis.

```
_device_ int _float_as_int ( float x )  
// Reinterpret bits in a float as a signed integer.  
  
inline int RenderCore::FloatAsInt( const float fVal )  
{  
    union FloatInt {  
        float f;  
        int i;  
    };  
  
    FloatInt fi;  
    fi.f = fVal;  
    return fi.i;  
}
```

Listing 2: An example of porting an intrinsic function. The first two lines present the documentation, and the implementation of the function is listed below.

Contrary to functions that involve intrinsics, some functions exhibit high portability. For instance, the source core function that implements picking a random point on a light requires no intrinsic function calls. Porting this function solely involves changing the function signature to the C++ syntax, and connecting the light count variables to the CPU-side counters. It is a significant maintenance cost bonus that there is no need to unit test these functions, as they haven't gone through semantic changes.

## 6. Results and Evaluation

This section reports on the results experiments we have conducted on our target core implementation, with the experimental setup devised linked to our research questions in Section 4.2. The chapter is split into three sections. In each section, we start by summarizing the experiment, then we report the results obtained, and finally evaluate the results through a discussion on how they answer to our research questions.

The answer to **RQ1**, which questions the possibility of a reliable CPU fallback, is partially answered by our successful ported render core implementation presented in Chapter 5. We present a close analysis of the quality of output produced by both render cores in Section 6.1. The performance of our ported render core is compared with the source core in Section 6.2, thus, answering **RQ2**. Finally, we present detailed code metrics that evaluate the portability and maintenance costs investigated by **RQ3**.

Figure 8 presents the scenes used in the experiments. The first column presents an image of the scene, the second column lists its name and the third column lists the amount of triangles and the size of the scene including the texture data.

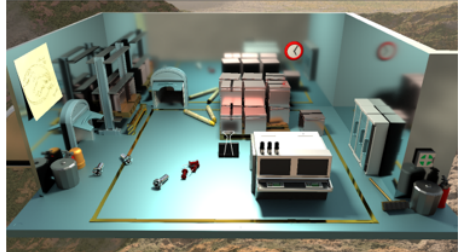


	<p><b>Pica Pica</b></p>	<p>76k triangles 12Mb Scene data</p>
	<p><b>Crytek's Sponza</b></p>	<p>270k triangles 330Mb scene data</p>
	<p><b>Amazon Bistro</b></p>	<p>3M triangles 7Gb scene data</p>

Figure 8: An overview of the scenes used in experiments.

## 6.1. Rendering Very Large Scenes

In order to answer **RQ1**, we need to validate that our target core implementation produces similar images to the source core. Our experiment setup is as explained in Section 4.2.1. In order to validate that our ported render core can render scenes that exceed the GPU memory, we have rendered the Amazon Lumberyard Bistro scene that consists of 3858088 triangles using our ported render core on the CPU [McG17].



Figure 9: The Amazon Lumberyard Bistro scene was rendered with our ported render core implementation. It is not possible to render this scene with the source render core as it exceeds the GPU memory.

In order to verify that our ported render core produces images that are quantifiably similar to the source core, we have produced rendered images of an identical scene, from an identical camera view pyramid. This means that the camera for both cores is at the same location, and the cores are rendering the same scene under identical lighting. It is important to note that the path tracing algorithm, relies on random sampling to estimate a higher order integral. Moreover, we use *jittering* and *blue noise*, that are also random processes, in primary ray generation, which causes every image to be rendered slightly differently. The usage of such random processes suggest that it is not possible to expect identical rendered images. Figure 10 presents two images produced by the two render cores after collecting 1024 samples. The images are visibly similar in quality.





Figure 10: A side-by-side comparison of an image produced after 1024 samples of the Sponza scene by our target render core on the left, and the source render core on the right.

Table 1 reports the data on the comparison of two images produced after collecting various amounts of samples from each render core. The Structural Similarity Index (SSIM) shows how similar two images in a range of 0.0 to 1.0. Higher SSIM values indicate higher image similarity. Root Mean Square Error (RMSE) is a metric that measures how different two images are due to factors such as noise. It is shown that in both Pica Pica and Sponza scenes, the images produced are increasingly similar to each other as the sample size increases. Although the SSIM values do not converge to 1, it is deduced that the two cores produce images that are %80 to %85 similar after collecting 1024 samples.

	Samples	SSIM Index	RMSE
Sponza Scene	1	0.697	56.3
	2	0.711	46.6
	4	0.719	40.3
	8	0.729	35.5
	16	0.737	32.5
	32	0.749	30.6
	64	0.761	29.5
	128	0.773	28.8
	256	0.785	28.4
	512	0.793	28.3
	1024	0.801	28.1
Pica Pica Scene	1	0.736	50.4
	2	0.751	43.2
	4	0.770	38.2
	8	0.792	35.3
	16	0.813	33.8
	32	0.835	32.9
	64	0.851	32.4
	128	0.864	32.1
	256	0.871	32.0
	512	0.876	31.9
	1024	0.878	31.9

Table 1: The Structural Similarity Index, Root Mean Squared Error and RGB distance measures between two images rendered with the source and target cores after collecting a given amount of samples.

The results display significant difference in images even after 1024 samples per pixel. We have subtracted the images rendered with two different cores and examined the result to identify where the difference lies. The difference image is shown in Figure 11. The highest intensity in the image is on the edges of objects. Thus, the difference in images can be attributed to the different results obtained from the intersection engine, which is OptiX for the source core and Embree for the target core.



Figure 11: The image difference of the two rendering cores on the Pica Pica scene after collecting 1024 samples per pixel.

## 6.2. Rendering Performance

As established in Section 4.2.2, we have investigated how the performance of our ported CPU render core compares to the source GPU render core. We setup an experiment that measures the amount rays traced per second in each core. This data provides an overall comparison of the rendering performance. Using a timer in the code, we have also calculated the time spent in various functional blocks of the render core. The experiments were conducted on a computer setup with a consumer grade Intel i7 CPU (2.2Ghz) and a NVIDIA GeForce GTX 1060 GPU.

Figure 12 shows an example output from the application during performance testing. The Pica Pica Scene, published by EA Games, consisting of 76250 triangles, was used in all performance measurements. This scene was chosen because is moderately sized, and with its wide range of materials and layout, it is representative of a production quality scene. The white overlays in Figure 12 denote the frames per second being rendered at the time of the capture. The left image was captured from the source (GPU) render core, and the right image was captured from the target (CPU) render core. In a first glance, it is evident that the source core exhibits real-time frame rates (40-45fps), whereas the target core exhibits offline frame rates (0.3-1fps). Thus, it is undesirable to use the target core in interactive scenarios. The overall rendering performance indicate that the target core is suitable for offline use.

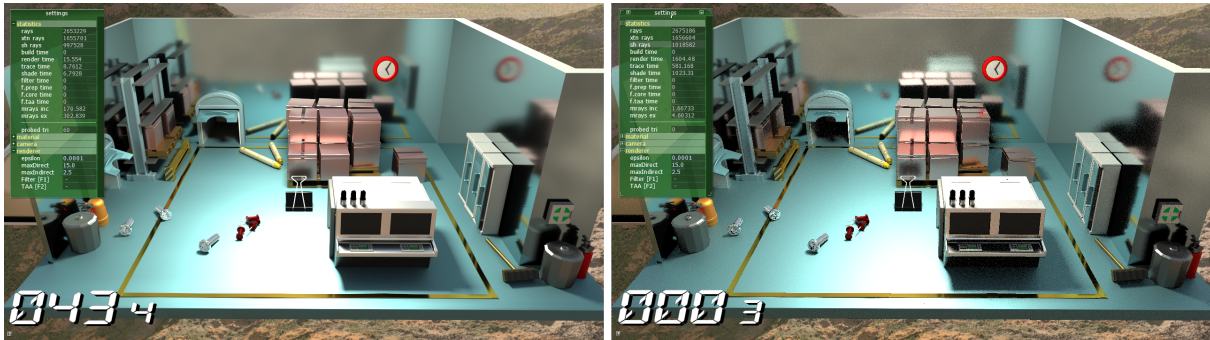


Figure 12: Side-by-side comparison of rendering performance. The white overlay shows the frames rendered per second. The left image is rendered with the source (OptiX) core, and the right image with the target core (Embree).

The Pica Pica scene, from the given viewpoint in Figure 12, is rendered with processing a total of 2.67 million rays per sample in both cores. The identicalness of the ray counts among cores is an additional verification to the software similarity between render cores. Before we take a closer look at the time spent in each functional block of the renderer, Figure 13 shows the decomposition of rays into two types; extension and shadow rays. Evidently, 61% of the rays are extension rays, which are tested for the nearest intersection in the scene. The remaining 38% are shadow rays that are tested for any intersection. The shadow rays are typically traced faster due to terminating on any hit.

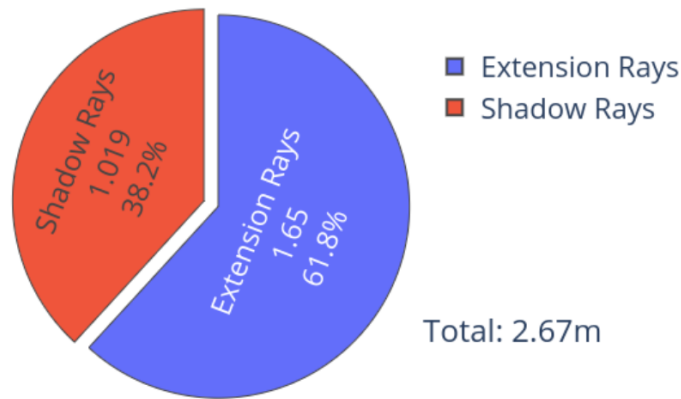


Figure 13: The distribution of the type of rays processed in each frame. Shadow rays are faster to process than extension rays. The ray amounts and distributions are identical in both cores.

A closer look into the performance characteristics of each core is presented through measuring the time spent in each functional block. There are two main functional blocks, *Shade* and *Trace*. Measuring the time spent in each block is important because it gives an insight into how performant the ported code is. The Trace block involves the time spent in API calls in each core, and the Shade block involves the time spent in the shading code that

we have ported. Figure 14 demonstrates the times spent in these blocks with the orange bars for the target core (Embree) and blue bars for the source core (OptiX). The Render block is the sum of the other two blocks.

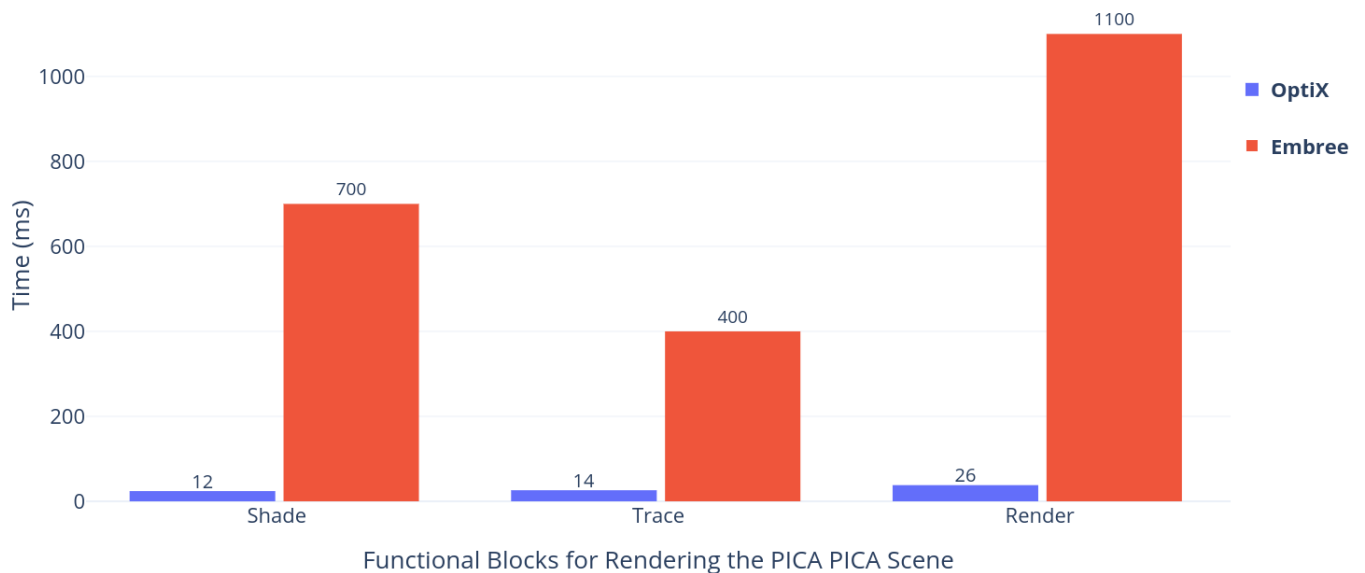


Figure 14: The time spent in functional blocks of rendering for the source (OptiX) and target (Embree) cores.

As seen in Figure 14, on average, it takes 1100ms for the target core to render, as opposed to the 26ms of the source render core. As hinted by the frame rates previously shown, it takes the target render core roughly 40 times more time to render each frame. Interestingly, while the source core spends 53% of its time in the Trace block, the target core spends 36% of its time in this block. This indicates that our ported shading code introduces a larger performance penalty than the Embree API calls in the Trace block. In other words, the ported shading code exhibits a larger room for optimization.

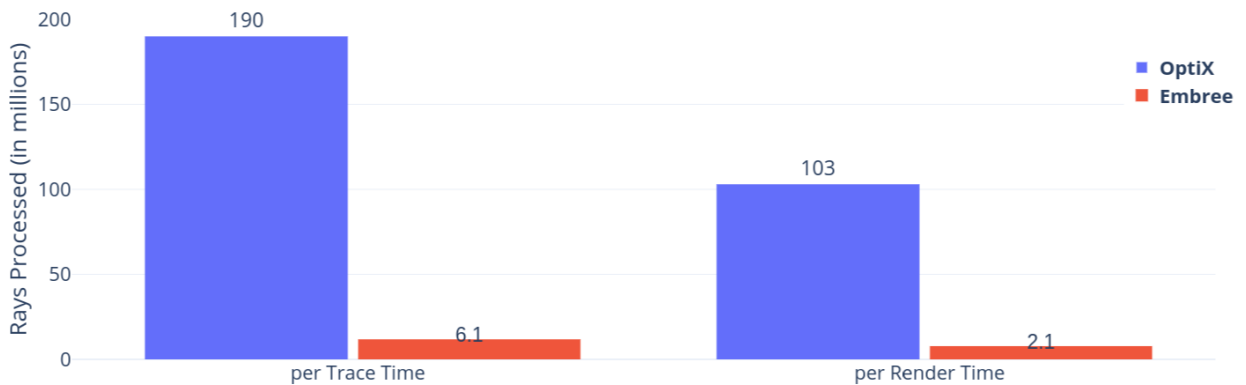


Figure 15: The amount of rays processed (in million rays per second) for each functional block source (OptiX) and target (Embree) render cores. The bars on the left measure million rays traced per second, and the bars on the right measure million rays rendered per second.

Figure 15 presents a comparison of the amount of rays processed in different time frames. The source core renders 103 million rays per second in total, shading and tracing combined. Compared to the 2.1 million rays per second of the target core, this indicates a 40-45x slow down on average rendering times. This is expected due to the serial implementation of the ported code. However, it is important that the performance loss in the Trace block is less than the performance loss in the Shade block. This shows that our ray tracing API Embree can potentially perform as well as OptiX in future parallel implementations.

Thus, our research question **RQ2** that investigates if the target core can exhibit similar performance to the source core is answered. The performance of the two cores are in different realms, meaning that while the source core renders in real-time, the target core renders in semi-interactive to offline frame rates.

### 6.3. Portability and Maintenance Cost

This section presents the results of the experiments explained in Section 4.2.3. In order to evaluate the portability of the source core, two cores have been compared to each other using Moss, an open source system for detecting software similarity by Stanford University [mos]. The service uses robust winnowing, a local fingerprinting algorithm for detecting local code similarity [SWA03]. The system is being provided as a web service. The Moss server is queried with submission scripts containing source files of the two render cores. Each query produces a HTML page that lists code similarity.

The number of shared lines is not the only metric that comprises the percent moss score between two blocks of code. The algorithm also takes into account the similar calculations done with different variable names. The moss score is an indicator to what percentage of a given code block is attributable to the reference code block. Thus, a high moss score indicates a high amount of code similarity. As we have established in Section 3.1.3, a functional block

	CPU Core			GPU Core		
	Total	Shared	% Moss Score	Total	Shared	% Moss Score
Render Core	2007	181	8	800	181	25
BSDF	100	73	86	141	73	62
Shading	216	101	51	246	101	37
Extend	43	0	0	109	0	0

Table 2: Lines of code in the source and the target cores.

is highly portable when the cost of porting is lower than the cost of redevelopment. The Moss percentage gives an accurate measurement on the degree of portability, because it is obtained by comparing the target core to the source core. This way, the cost of porting is measured against the cost of redevelopment through the code produced.

We present the results in Table 2. As expected the highest similarity score of %86 is reached with the target core BSDF implementation. This means that the smallest functional block, the BSDF calculation, is as closely coupled between cores as possible. This indicates that the costs for maintaining the source and target cores are minimized. On the other hand, the ported Extend kernel results in a %0 Moss score, which means that significant engineering effort was made to implement it, porting future changes to the source Extend kernel will require similar effort. Luckily, the components that have low Moss score are more central components and are subject to less change in the future. The porting effort for the low Moss score components can be seen as a one time effort. On the other hand, the BSDF and the Shading kernels are volatile, and changes to these can be ported with minimal effort to the target core.

Figure 16 shows a side by side comparison of the decompositions of the source code of both cores into functional groups. The number under the name of the functional group indicates the lines of code it contains, and the percentage shows how much of the codebase it constitutes. Due to the persistent threads implementation, the Extend takes up a larger portion of the source core codebase compared to the serial implementation the target core codebase.

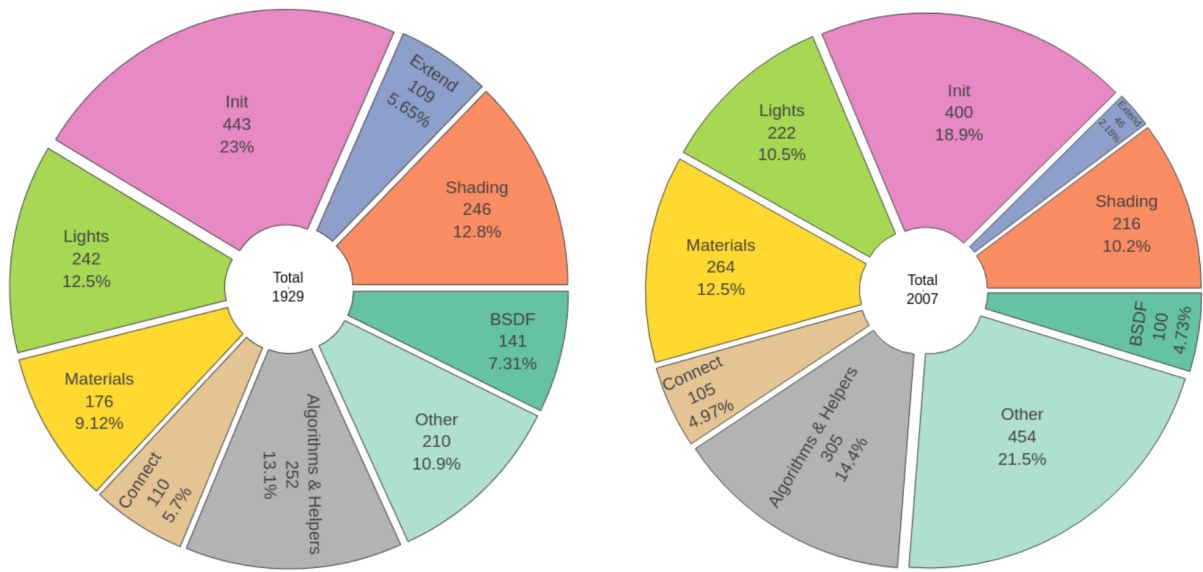


Figure 16: The decomposition of the render cores in numbers of lines of code. The left figure shows the source core and the right figure shows the target core.



# 7. Conclusion

In this master’s thesis, we initially introduced the problem of the inability to render very large scenes on the GPU due to device memory limits. We formulated three research questions in Section 1.2 that investigate the feasibility and the quality of a CPU fallback in such cases. We have conducted a literature study in Section 3 to establish the state-of-the-art on software portability and ray tracing on the CPU. In order to answer our research questions, we ported the GPU render core of a production quality photorealistic renderer, LightHouse, to the CPU.

We have investigated the feasibility of a CPU fallback, by rendering scenes that overflow the GPU memory, in our ported CPU implementation. We have successfully rendered a scene that does not fit into the GPU memory with our ported render core implementation. Thus, we conclude that a ported CPU path tracer can be a reliable fallback to the source GPU path tracer for large scenes. We have also found that images rendered with our ported target core differ from the images rendered with the source GPU by 10-15%. This difference was attributed to the difference in the intersection data provided by the intersection engines used in the source and target cores.

As for performance, our findings indicate 40-45x lower overall rendering performance in our ported render core. This puts our CPU render core into the offline renderers category, limiting its use in interactive scenarios. It is important to note that there is no significant advantage to using our ported core in system configurations that have a GPU that has sufficient memory, and our ported render core is intended as a fallback strategy.

We have successfully ported the source render core in such a way that the maintenance costs are minimized. The portability measurements suggest that the most volatile components of the render core have the highest software similarity. Thus, it is expected that future changes in the source codebase can be continuously ported to the target core with minimal effort. Through porting a new BSDF functional block to our target core, we have verified that the source and the target cores can be maintained with minimal effort.

## 7.1. Future Work

We have completed the porting case study with an implementation presented in Section 5. However, as the results in Section 6.2 demonstrate, the CPU render core is a long way from rendering in real-time frame rates. Increasing the performance of the CPU render core is a promising lead for future research. Achieving this goal while keeping the maintenance costs low is non-trivial, as the target architectures of the source and target cores vastly differ. However, implementing parallelized ray intersection routines could be seen as the next step of this research.

The source core contains advanced features such as filtering and temporal anti-aliasing. Porting these features to our CPU render core is another subject for future work. It is expected that these features can be ported with minimal effort due to the promising results we have obtained in our portability experiments.

Finally, automating the porting process between cores can be seen as the ultimate goal this research introduces. In functional blocks with high software similarity, a change in the source core could be automatically applied to the target core through the use of a pre-processor block or cross-compilation setup.

## 8. References

- [AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.
- [amd] AMD Radeon RX 550x | AMD. <https://www.amd.com/en/products/graphics/radeon-rx-550x>. Accessed: 2019-02-13.
- [Ant11] Dietger van Antwerpen. Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU. In Carsten Dachsbacher, William Mark, and Jacopo Pantaleoni, editors, *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. ACM, 2011.
- [App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.
- [DWBS03] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The OpenRT Application Programming Interface Towards A Common API for Interactive Ray Tracing . page 9, 2003.
- [GS93] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume Volume 2 of *Series on Software Engineering and Knowledge Engineering*, pages 1–39. WORLD SCIENTIFIC, December 1993.
- [HO97] Mitsuari Hakuta and Masato Ohminami. A study of software portability evaluation. *Journal of Systems and Software*, 38(2):145–154, August 1997.
- [Kaj86] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.
- [KB94] Rick Kazman and Len Bass. Toward Deriving Software Architectures from Quality Attributes. page 44, 1994.
- [LGXT17] Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. Vectorized production path tracing. In *Proceedings of High Performance Graphics*, HPG '17, pages 10:1–10:11, New York, NY, USA, 2017. ACM.
- [LKA13] Samuli Laine, Tero Karras, and Timo Aila. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 137–143, New York, NY, USA, 2013. ACM.
- [MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, May 1990.
- [McG17] Morgan McGuire. Computer graphics archive, July 2017. <https://casual-effects.com/data>.
- [Moo90] James D. Mooney. Strategies for Supporting Application Portability. *Computer*, 23(11):59–70, November 1990.
- [Moo00] James D. Mooney. Bringing Portability to the Software Process. 2000.
- [Moo01] James D. Mooney. Issues in the Specification and Measurement of Software Portability. 2001.
- [Moo04] James D. Mooney. Developing Portable Software. In Ricardo Reis, editor, *Informa-*

- tion Technology*, IFIP International Federation for Information Processing, pages 55–84. Springer US, 2004.
- [mos] Moss: Plagiarism detection. <https://theory.stanford.edu/~aiken/moss/>, url = <https://theory.stanford.edu/~aiken/moss/>, abstract = Moss., language = en-us, urldate = 2019-05-07, note = Accessed: 2019-05-07, keywords = Moss, file = Snapshot:/Users/basaroguz/Zotero/storage/WYWFSNB6/geforce-gtx-1060.html:text/html.
- [Nic65] Fred E. Nicodemus. Directional Reflectance and Emissivity of an Opaque Surface. *Applied Optics*, 4(7):767–775, July 1965.
- [nvi] GeForce GTX 1060 Graphics Cards from NVIDIA GeForce. <https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1060>. Accessed: 2019-02-13.
- [OM87] Masataka Ohta and Mamoru Maekawa. Ray Coherence Theorem and Constant Time Ray Tracing Algorithm. In Toshiyasu L. Kunii, editor, *Computer Graphics 1987*, pages 303–314. Springer Japan, 1987.
- [PBD<sup>+</sup>10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, pages 66:1–66:13, New York, NY, USA, 2010. ACM. event-place: Los Angeles, California.
- [PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, September 2016. Google-Books-ID: iNMVBQAAQBAJ.
- [PM12] Matt Pharr and William R. Mark. ispc: A spmd compiler for high-performance cpu programming. *2012 Innovative Parallel Computing (InPar)*, pages 1–13, 2012.
- [PW75] P. C. Poole and William M. Waite. Portability and Adaptability. In *Software Engineering, An Advanced Course, Reprint of the First Edition [February 21 - March 3, 1972]*, pages 183–277, Berlin, Heidelberg, 1975. Springer-Verlag.
- [RK12] Baishakhi Ray and Miryung Kim. A Case Study of Cross-system Porting in Forked Projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 53:1–53:11, New York, NY, USA, 2012. ACM.
- [RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional Representation for Fast Rendering of Complex Scenes. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '80, pages 110–116, New York, NY, USA, 1980. ACM. event-place: Seattle, Washington, USA.
- [Som11] Ian Sommerville. *Software engineering*. Pearson, Boston, 2011. OCLC: 758329000.
- [SWA03] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.
- [TKB78] Andrew S. Tanenbaum, Paul Klint, and Wim Bohm. Guidelines for software portability. *Software: Practice and Experience*, 8(6):681–698, November 1978.
- [Vea98] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD Thesis, Stanford University, Stanford, CA, USA, 1998.
- [WBSS04] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment:

From error visibility to structural similarity. *Trans. Img. Proc.*, 13(4):600–612, April 2004.

[WSB<sup>+</sup>14] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics*, 33(4):1–8, July 2014.