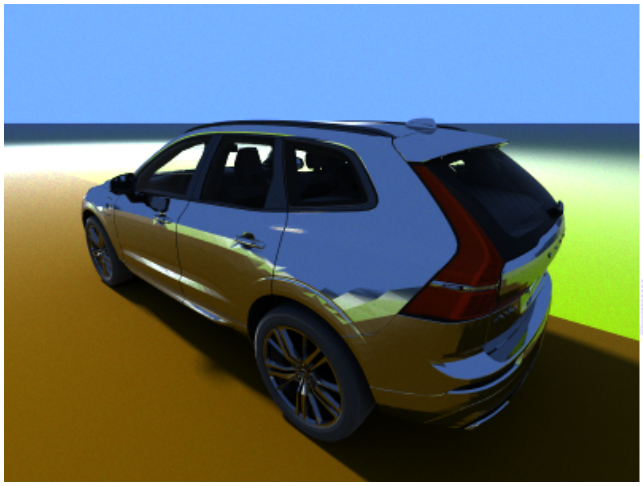
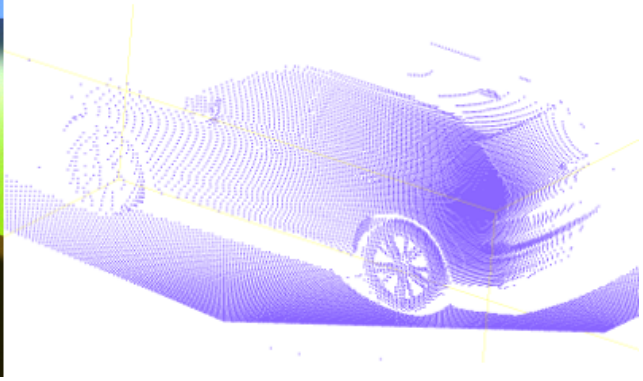
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG





Ray Tracing for Sensor Simulation using Parallel Functional Programming

Master's thesis in Computer science and engineering

Johan Johansson & Ari von Nordenskjöld

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Ray Tracing for Sensor Simulation using Parallel Functional Programming

Johan Johansson
Ari von Nordenskjöld

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Ray Tracing for Sensor Simulation using Parallel Functional Programming

Johan Johansson & Ari von Nordenskjöld

Supervisors: John Hughes & Mary Sheeran, Department of Computer Science and Engineering

Advisor: Göksan Isil, Volvo Cars

Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A Volvo XC40 rendered using our ray tracer in camera mode on the left and LIDAR mode on the right.

Typeset in LaTeX
Gothenburg, Sweden 2020

Ray Tracing for Sensor Simulation using Parallel Functional Programming

Johan Johansson
Ari von Nordenskjöld
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In the modern automotive industry, cars are tested in virtual environments in order to save time and money. Sensors are important components of modern cars which also need to be tested together with the rest of the car and its software. Ray tracing is an effective method for simulating how light interacts with an environment, and can be used for simulating the sensors used in modern cars.

We present an implementation of a physically based ray tracer that can flexibly simulate both vision and LIDAR sensors. This is enabled by the presence of certain features like MIS and spectral path tracing. The ray tracer is written in the parallel functional programming language Futhark, and runs in parallel on the GPU. For validation, we provide a framework of relative measurements which can be used to compare the ray tracer to other implementations, or interpreted as an error to minimise for systematically improving the accuracy of the implementation.

Keywords: physically based ray tracing, parallel functional programming, sensor simulation, LIDAR, camera, Futhark

Acknowledgements

First we want to thank our supervisors at Chalmers – Mary Sheeran and John Hughes – for their continuous feedback and their important guidance, which helped us set a good course and made our work more accessible. We are equally thankful to our supervisor at Volvo Cars – Göksan Isil – for his great support, dedication, and enthusiasm. He always made sure we had access to the right tools and expertise, which helped us keep the project concrete and aligned with Volvo's vision.

We're very thankful to Troels Henriksen, the author of Futhark, without whose incredible work on the language this project would obviously not have been possible at all. We are also grateful for his ready help when we had problems with the language, and his valuable feedback on our code.

Thank you also to the other people at Volvo who helped with the proposal – Anders Ödblom, Siddhant Gupta, Fredrik Björkqvist, Francesco Costagliola, Mikael Andersson, Tobias Johansson, and all the members of the Virtual Car team.

Johan Johansson & Ari von Nordenskjöld, Gothenburg, June 2020

Contents

Glossary

energy Light sources emit photons, each of which is at a particular wavelength and carries a particular amount of energy. x

irradiance The radiant flux received by a surface per unit area. 26

kernel A function that executes on the GPU device. 15

radiance The central quantity in radiometry, denoted by L. Measures irradiance with respect to solid angle – in other words, radiance is associated with a certain direction. 26, 29

radiant flux "also known as power, is the total amount of energy passing through a surface or region of space per unit time" [Pharr et al., 2016] . x

solid angle The extension of 2D angles in a plane to an angle on a sphere. Measured in steradians (sr). Like the circumference of a circle is 2π radians, the area of a sphere is 4π steradians. 28, 29

Acronyms

AABB axis-aligned bounding boxes. 13

BSDF bidirectional scattering distribution function. 21, 22, 25, 26, 29, 32

BVH bounding volume hierarchy. 13, 14, 46

MIS multiple importance sampling. 29

SSIM structural similarity index. 40

1
Introduction

Modern vehicles contain a lot of software that needs to be tested and verified. It is especially important to test software related to safety features and autonomous driving. In order to drastically increase the number of scenarios that can be tested, such software is tested within virtual environments. This makes it possible to identify corner cases where systems do not behave as expected, that might not otherwise have been found due to the costs and risks associated with testing vehicles on a large scale in the real world [Belbachir et al., 2012].

Sensors play a central role in the safety systems used in modern vehicles, and are essential for the implementation of autonomous driving systems [Hecht, 2018]. Figure 1.1 conceptually illustrates how sensors are used together with perception algorithms to enable autonomous driving.

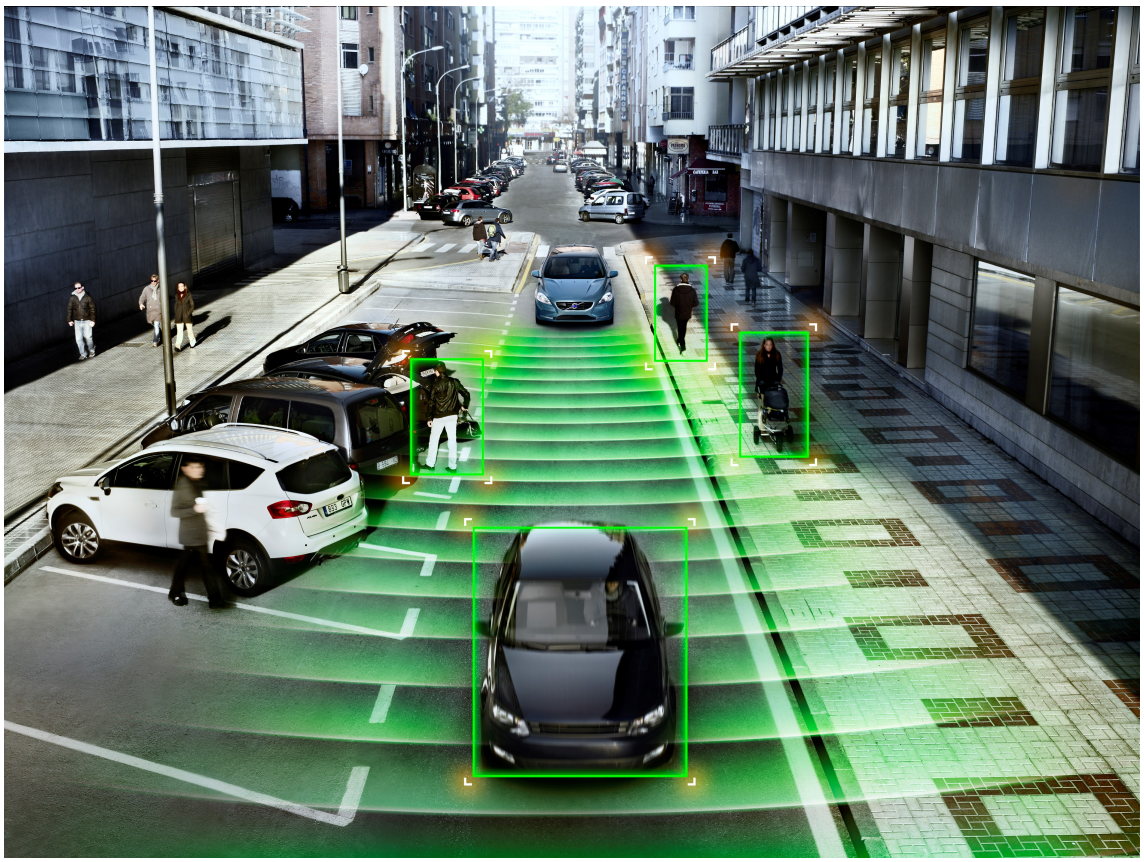


Figure 1.1: Sensors provide safety to prevent accidents and enable autonomous driving. Image by Volvo Cars.

It is important to test the sensors together with other software systems within the virtual environments. Modern vehicles are equipped with many sensors, but the vision and LIDAR sensors are the ones that are of primary interest for this thesis. The LIDAR sensor uses the backscattering of laser light to measure distances to objects in the environment. The frequencies used vary depending on the area of application, but in the automotive industry, infrared frequencies near the visible spectrum are commonly used. In the context of this thesis, the vision sensor is just a conventional camera.

Both cameras that use visible light and LIDAR depth sensors can be simulated using *ray tracing*. Majek and Bedkowski claim that "ray tracing algorithms are the natural choice for [depth sensors]", and show via experiments on several different virtual scenes that ray tracing, and specifically GPU ray tracing, can be efficiently used to simulate depth sensors [Majek and Bedkowski, 2016]. Ray tracing approaches are even more of an industry standard when it comes to simulating visible light. For example, the *Arnold* renderer is a physically based rendering system used to simulate light, and was used to generate imagery for movies like the critically acclaimed *Gravity* (2013) [Pharr et al., 2016, chapter 1.7].

The shortcomings of existing LIDAR ray tracers like the one shown by Majek and Bedkowski is that they tend to put performance before accuracy, and don't use physically based models. They don't take material properties into consideration or compute indirect lighting. Visual ray tracers like Arnold do take a physically based approach, but only operate in the visible light spectrum. Another issue with existing ray tracers is that they tend to be implemented with the low-level languages typically used for general purpose GPU programming – that is, OpenCL and CUDA. The low-level nature of these languages can distract from the domain specific details of ray tracing and make the implementation exceedingly complex.

In this thesis, we present a flexible ray tracer that uses physically based models to simulate both the transportation of visible light, and infrared LIDAR depth sensing. A programming language called Futhark is used for the implementation, which is a high-level language that abstracts away distracting low-level details, but through heavy optimization still produces well-performing GPU code. One of the primary goals of this project has been to produce code that is easy to read and understand, and that does not obscure the domain-specific details of the implementation.

This project has been carried out in collaboration with the Active Safety and Autonomous Driving department at Volvo Cars.

2
Background

2.1 Ray tracing

Ray tracing can be used to simulate the functionality of both the camera and the LIDAR sensor; it gives us information about how the light behaves within a given scene. Figure 2.1 illustrates how the ray tracing pipeline relates to the verification of automotive functions. A ray tracer, given a scene, produces data that can be fed to perception software, that in turn can be used to verify how car systems respond to various situations.

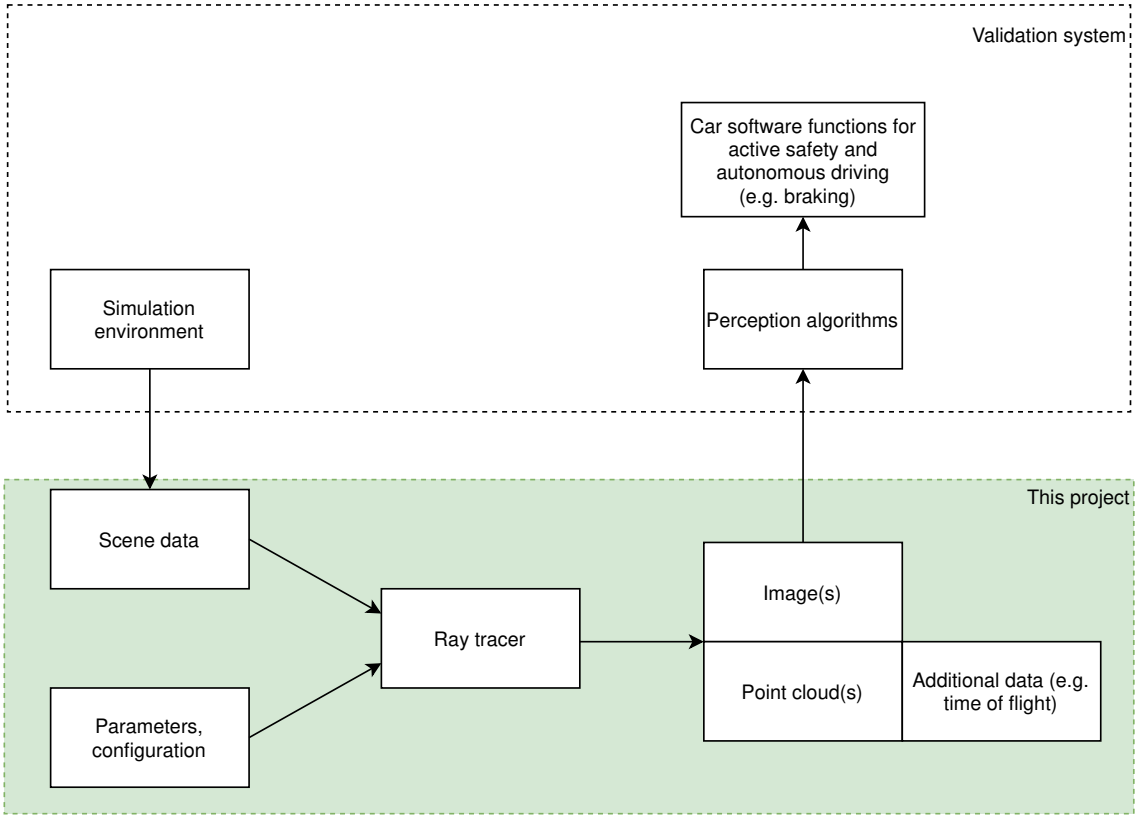


Figure 2.1: The context of the system at a glance

Ray tracing is an overloaded term. It can refer either specifically to *ray casting* – the problem of finding the first object intersected by a ray – or more generally to the whole class of algorithms based on recursively casting rays, that among others includes *Whitted style ray tracing* [Whitted, 1979], *path tracing* [Kajiya, 1986], and

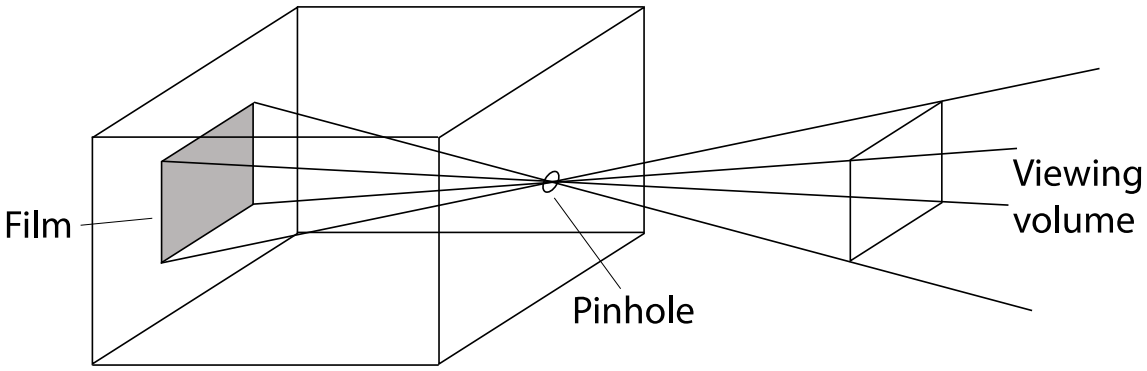


Figure 2.2: The pinhole camera model.
From [Pharr et al., 2016]: CC BY-NC-SA 4.0

bidirectional path tracing [Veach and Guibas, 1995]. Ray tracing can both refer to techniques that are physically based as well as those that are not.

We use the term *ray tracing* when discussing our project, in the hope that it's a less foreign term to readers with primary expertise in other domains than physically based rendering, such as LIDAR technology or sensor simulation. From a computer graphics perspective, our implementation could less ambiguously be called a *path tracer*.

To give an intuitive understanding of how ray tracing – that is, some recursive application of ray casting – can be used to simulate visible light in computer graphics, consider the pinhole camera in Figure 2.2. Our computer screen can be thought of as the film plane in the camera box, mirrored in front of the pinhole. The amount of light that "hits" a pixel on the film is computed by casting a ray from the camera origin – the pinhole – through the mirrored pixel in the screen plane, into the scene. The first intersection point of the ray and the scene is returned, and the amount of outgoing light from that point along the ray is then computed by some recursive method. A very simple illustration of the basic idea can be seen in Figure 2.3, where rays 1 and 3 hit different surfaces and are reflected in various directions depending on the material. Ray 2 hits a glass ball and is refracted through it.

When used to simulate the camera, ray tracing can be thought of as just rendering an image of the virtual scene. LIDAR is similar, but the light behaves differently and the properties of interest are different – we want to find out the time it takes for the light to travel from the transmitter, to reflect/refract at a point of interaction in the scene, and to travel back to the sensor. A point cloud that represents those interactions can then be generated. A LIDAR point cloud, illustrated in Figure 2.4, is similar to a rendered image, but is better for visualising the properties just mentioned, as the perceived colours of the visible light spectrum are no longer of interest. Instead, colours can be used to encode other information such as measured distance.

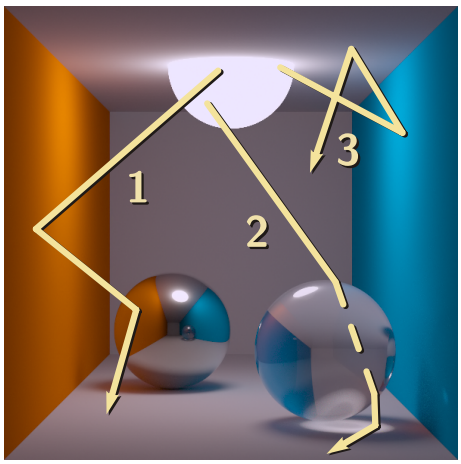


Figure 2.3: An example of light interacting with a scene.

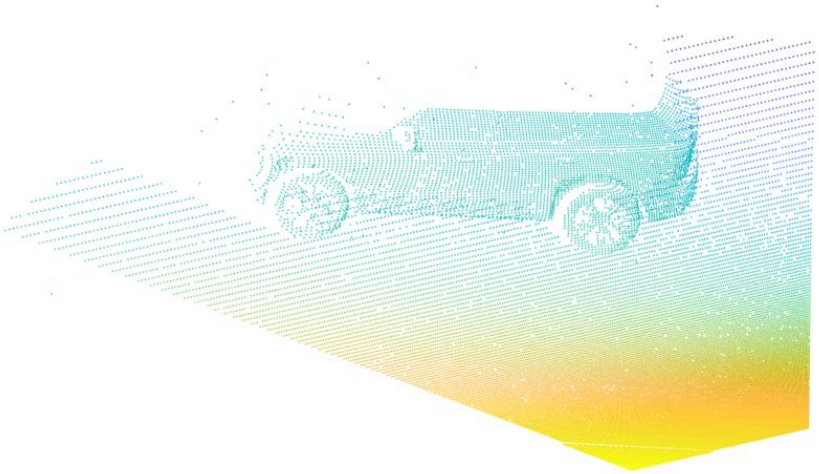


Figure 2.4: A LIDAR point cloud, rendered by our ray tracer. The points are re-projected from the LIDAR view to world space to the current camera view, using depth information captured as time-of-flight data. The depth from the sensor is additionally visualised with colour. The model car is a Volvo XC60.

2.2 Parallel ray tracing

Ray tracing is quite computationally demanding. Doing it on a single-core CPU is possible, but only if your time is worthless. Instead, the massive parallelisation of modern GPUs can be utilised to greatly accelerate the process. While ray tracing does often present a very incoherent workload (which results in degraded performance on GPUs) compared to rasterised graphics, the embarrassingly parallel[1] nature of the problem mostly makes up for this disadvantage. If the GPU architecture is taken into consideration when writing the implementation, it is possible to achieve significantly better performance on GPUs than CPUs [Parker et al., 2010], and the current advances in hardware acceleration of ray tracing on GPUs only further incentivises writing for the GPU first.

Writing general purpose GPU (GPGPU) code that fits the GPU architecture is far from trivial, however. The two main APIs, OpenCL and CUDA, are fairly low-level and can be complicated to work with. This is undesirable when we want to write software that is designed with the domain knowledge of ray tracing in focus, and low-level implementation details would get in the way of understanding the domain.

There are many aspects that combine to make parallel programming hard. When a mutable state is used, a lot of care usually has to be taken to not let different threads interfere with each other in a manner that was not intended. Such unintended interference can be extremely hard to debug when the order of execution is not deterministic. Traditional ways of dealing with this include various abstractions such as locks and semaphores, which add a lot of complexity and cognitive overhead for the programmer.

2.3 Futhark and data-parallel functional programming

Functional programming languages offer an attractive alternative to the traditional methods for dealing with parallelism. Some functional programming languages reject shared mutable state, thereby eliminating many of the pitfalls of imperative programming, especially in the context of multiple threads. Functional languages also allow for composing high-level abstractions that can be used to model many problems in a more expressive manner. These abstractions make them ideal for describing complex domains (such as ray tracing) in a way that is easy to understand by just reading the code, and also makes it much easier to write modular and flexible programs [Hughes, 1989].

Futhark [Henriksen et al., 2017] is a Functional programming language that can be compiled to OpenCL or CUDA code, allowing the programmer to write high-level programs for the GPU without worrying about any of the low-level details. Futhark is part of a programming language design tradition that follows in the footsteps of several seminal works by Guy Blelloch. He showed that many important

[1] A problem is embarrassingly parallel when it requires very little shared state and/or data dependencies between threads

```
function QUICKSORT(S) =
if (#S <= 1) then S
else
  let a   = S[rand(#S)];
      S_1 = {e in S | e < a};
      S_2 = {e in S | e == a};
      S_3 = {e in S | e > a};
      R   = {QUICKSORT(V): v in [S_1, S_3]};
  in R[0] ++ S_2 ++ R[1];
```

Listing 1: QuickSort in NESL [Blelloch, 2000]

algorithms could be expressed in terms of simple array operations such as `map` and `fold`, and that those operations could be parallelised very well [Blelloch, 1990]. He also introduced a concept language called NESL in 1992 [Blelloch, 1992], which could compile those array operations for highly parallel systems.

NESL had a very simple syntax, and a built-in performance model for keeping track of both the total amount of work and the degree of possible parallelism of a program. Listing 1 shows an implementation of QuickSort in NESL. Notice that the list comprehensions have curly brackets – that is all that is needed for achieving parallel computations on lists in NESL. Implementing the same code with the standard tools used for writing parallel code at the time – C and MPI – resulted in 1700 lines of code [Blelloch, 2000].

This way of expressing parallelism is called *data parallelism*, and is best thought of as a programming paradigm of its own. The fundamental idea is to express programs as parallel operations on multiple data. For example, `map` is perhaps the most important such operation. `map` takes a function and applies it to each element of an entire array. When using `map` in a non-parallel context, the array is processed sequentially – but as the order in which the array is processed does not matter, it could just as well be done in parallel. A data parallel language has a compiler that recognises such opportunities for parallelism and generates appropriate parallel code for a target architecture.

Futhark uses *Second Order Array-Combinators* (SOACs) to indicate code that could potentially be executed in parallel. SOACs are higher-order functions like `map` (as well its siblings `map2` through `map5`, which apply a function that takes multiple arguments to multiple arrays), `filter`, `scan`, `reduce`, `partition`, and `scatter`.

Futhark also supports *nested* data parallelism - SOACs can be combined and nested as many times as desired. This may seem obvious, but it's an essential feature of Futhark, and is enabled by a *flattening algorithm* which was introduced by [Blelloch, 1990]. A standard example of this is shown in Listing 2 which shows a function for multiplying matrices in Futhark.

Listing 3 shows the result of a full flattening transformation on the code in Listing 2. It turns out, however, that this is a naïve way of doing things, and the compiler needs to be much smarter than that.

```
map (\xs -> map (\ys -> let zs = map2 (*) xs ys
                        in reduce (+) 0 zs)
                yss)
    xss
```

Listing 2: Matrix multiplication in Futhark [Henriksen, 2019]

```
let ysss = replicate n (transpose yss)
let xsss = map (replicate n) xss
let zsss = map2 (map2 (map2 (*))) xsss ysss
in map (map (reduce (+) 0)) zsss
```

Listing 3: Full flattening of Listing 2 [Henriksen, 2019]

2.3.1 Futhark and the GPU

In the words of [Henriksen, 2017], Futhark was designed to be "a small functional language that has just the features and restrictions to enable the construction of an aggressively optimising compiler capable of generating GPU code. This compiler will worry about all the low-level details, leaving the programmer solely concerned with how best to express the parallelism in their algorithm."

It is still worth knowing where some of its restrictions come from. Targeting APIs that run on GPUs – such as CUDA and OpenCL – imposes many restrictions on the generated code:

- Parallelism has to be *flat*, meaning that a parallel thread may not spawn additional threads
- Memory cannot be allocated during GPU runtime - it has to be allocated before the execution is initiated
- It's not possible to use function pointers
- GPU stacks are too small for recursion

The flattened code in Listing 2 preserves the asymptotic time complexity of the original code, but requires much more space for the intermediate arrays – generally, the amount of space needed may increase polynomially, which is not acceptable on a GPU. Another issue is that full flattening will usually lead to excess parallelism and thereby bad performance. In the example which we have seen, it would lead to all three maps being run in parallel – but it would be better to execute the innermost map sequentially.

Futhark uses a technique called *moderate flattening* in order to ameliorate these issues. Moderate flattening employs various heuristics in order to avoid excess nested parallelism and turn it into efficient sequential code instead. An example of

such a heuristic is that nested `map-reduce` compositions are always sequentialised [Henriksen, 2019].

Moderate flattening relies on static assumptions, however, that may not be valid at runtime, depending on the specific workloads. This is why Futhark also employs a technique called *incremental flattening*, which generates multiple versions of the code for each instance of nested parallelism and picks the most suitable version at runtime. This technique is described in [Henriksen et al., 2019].

These compiler techniques are not of primary interest for the purposes of this report, however, beyond knowing that they are there and do a lot of work for us. One of the main advantages of Futhark is that it is a language that makes sure that programmers do not need to worry too much about such aspects.

3

Implementation

In this chapter, we describe the implementation of a physically based ray tracer capable of rendering both camera images and LIDAR point clouds. The concepts necessary to understand the implementation are explained, and interesting parts of the code are shown.

3.1 Ray casting

At the heart of the ray tracer lies the ray casting algorithm. It is a general algorithm that performs intersection tests between a ray and a set of geometric objects. The algorithm has two entry points: one to find the closest intersection, and one that returns whether *anything* was intersected at all.

A geometric ray is represented by an origin point in the 3D space and a direction vector, and a triangle is represented by its three corner points.

```
type ray = { origin: vec3, dir: vec3 }
type triangle = { a: vec3, b: vec3, c: vec3 }
```

When discussing the mathematics of ray tracing, we will denote the origin with o, the direction with d, and a function $p(t)$ that moves a point along the ray according to the parameter t. The function p is implemented as follows:

```
let point_at_param (r: ray) (t: f32): vec3 =
  r.origin vec3.+ vec3.scale t r.dir
```

To find the points where a ray intersects a geometric primitive, we have to figure out what values of t in $p(t)$ are on the surface of the geometric primitive. For our purposes we will only need triangles, since they can be used to approximate arbitrary geometries. The following presentation of how to test for triangle intersections is based on [Akenine-Möller et al., 2018, Chapter 22.8].

When specifying a point within a triangle, we use *barycentric coordinates*. A point is given by three such coordinates: u, v, and $1 - u - v$, where $u \geq 0, v \geq 0$, and $u + v \leq 1$. These coordinates specify how much each vertex contributes to the location in question. A point on the triangle can then be specified as a function of u and v:

$$f(u, v) = (1 - u - v)p_0 + up_1 + vp_2$$

Finding out where the ray intersects the triangle is then a matter of figuring out where $p(t) = f(u, v)$:

$$\begin{gathered} o + td = (1 - u - v)p_0 + up_1 + vp_2 \\ \Longleftrightarrow \\ \begin{pmatrix} -d & p_1 - p_0 & p_2 - p_0 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = o - p_0 \end{gathered} \tag{3.1}$$

Solving this system of linear equations will give us the value of the parameter t, which is what we are really interested in, as well as the barycentric coordinates (u, v).

If $M = \begin{pmatrix} -d & p_1 - p_0 & p_2 - p_0 \end{pmatrix}, e_1 = p_1 - p_0, e_2 = p_2 - p_0$, and $s = o - p_0$, we can find the solution by multiplying Equation 3.1 with M^{-1} and applying Cramer's rule [Cramer, 1750, pp. 656-659]:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{det(-d, e_1, e_2)} \begin{pmatrix} det(s, e_1, e_2) \\ det(-d, s, e_2) \\ det(-d, e_1, s) \end{pmatrix} \tag{3.2}$$

We know that $det(a, b, c) = |a\ b\ c| = -(a \times c) \cdot b = -(c \times b) \cdot a$, which allows us to rewrite Equation 3.1 as:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(d \times e_2) \cdot e_1} \begin{pmatrix} (s \times e_1) \cdot e_2 \\ (d \times e_2) \cdot s \\ (s \times e_1) \cdot d \end{pmatrix} \tag{3.3}$$

The normal is $e_1 \times e_2$ and thus constant, as is $s \times d$ which we denote m, so Equation 3.1 can in turn can be reformulated in order to reduce the number of computations:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{-(e_1 \times e_2) \cdot d} \begin{pmatrix} (e_1 \times e_2) \cdot s \\ (s \times d) \cdot e_2 \\ -(s \times d) \cdot e_1 \end{pmatrix} = \frac{1}{-n \cdot d} \begin{pmatrix} n \cdot s \\ m \cdot e_2 \\ -m \cdot e_1 \end{pmatrix} \tag{3.4}$$

We can now write a `hit` function for triangles. As should now be apparent, it's pretty straight-forward once the maths is in place.

```
let hit_triangle (tmax: f32) (ra: ray) (tr: triangle)
                 : maybe hit =
  let eps = 0.00001
  let e1 = tr.b vec3.- tr.a
  let e2 = tr.c vec3.- tr.a
  let n = vec3.cross e1 e2
```

```
  let a = -(vec3.dot n ra.dir)
  in maybe.when (!(approx_zero a eps))
     <| let s = ra.origin vec3.- tr.a
        let m = vec3.cross s ra.dir
        let { x = t, y = u, z = v } =
          vec3.scale (1 / a)
                     (mkvec3 (vec3.dot n s)
                             (vec3.dot m e2)
                             (-(vec3.dot m e1)))
        let in_triangle = u >= 0 && v >= 0 && u + v <= 1
        in maybe.guard (in_triangle && in_bounds t tmax)
           <| let pos = point_at_param ra t
              let normal = vec3.normalise n
              in { t, pos, normal }
```

We use bounds on t to make sure we don't intersect things outside the interval of interest. For example, we may restrict the maximum length of the ray in order to detect if there's any geometry in the way causing one point to be occluded relative to another. Also, we only want to detect intersections that are in front of the camera, in view. Bounds will also be relevant when implementing acceleration structures later. They are trivially checked as follows:

```
let in_bounds (tmax: f32) (t: f32): bool = t < tmax && t > 0
```

Also note that what we return is a `maybe hit`, because there's no sensible value of type `hit` that we can return when the ray doesn't actually intersect anything. Futhark has a minimalistic standard library, and as such there is no already defined `maybe` type to use. We found ourselves reaching for it repeatedly however, so we define our own `maybe` module with the type as well as some associated functions.

```
module maybe = {
  type maybe 't = #nothing | #just t

  let is_just 'a (x: maybe a): bool =
    match x case #just _ -> true
            case #nothing -> false

  let map 'a 'b (f: a -> b) (x: maybe a): maybe b =
    match x
    case #just a -> #just (f a)
    case #nothing ->  #nothing

  let when 'a (pred: bool) (x: maybe a): maybe a =
    if pred then x else #nothing

  let guard 'a (pred: bool) (x: a): maybe a =
    if pred then #just x else #nothing
```

```
  let or 'a (x: maybe a) (y: maybe a): maybe a =
    match x
    case #just a -> #just a
    case #nothing -> y

  let unwrap_or 'a (default: a) (x: maybe a): a =
    match x
    case #just a -> a
    case #nothing -> default
}
```

A `hit` contains the value of the parameter t, its position in space, and its normal.

3.1.1 Bounding volume hierarchy

A naïve ray tracer would iterate over the whole list of geometric primitives in the scene for each cast, testing for collision with each primitive and finally selecting the closest hit. This brute-force algorithm for scene traversal would have a linear time complexity, which is completely unacceptable since it is in fact possible to solve the same problem in logarithmic average-case time complexity. By employing a certain kind of data-structure for the purpose of organising and traversing the geometric primitives of a scene, we can greatly accelerate the process. There are many different kinds of such acceleration structures that make different trade-offs with regards to construction time and traversal performance (usually denoted *quality*). If a scene is completely static and a large number of rays are cast, a structure with good quality is desirable and the constant construction time doesn't matter as much. If the scene is instead very dynamic – i.e. objects are moving around over time – and fewer rays are shot per time-unit, it is more important that the structure is fast to reconstruct in every step.

We've chosen to implement the *bounding volume hierarchy (BVH)* acceleration structure from [Karras, 2012]. This specific BVH has the property that while the quality is not very high compared to more expensive methods, it is cheap to construct, making it particularly good for dynamic scenes. This is somewhat interesting to us, as we want the ray tracer to be usable with dynamic scenes in the future. More importantly, however, this BVH is specifically designed to be efficiently implementable with the constraints of a GPU. This will be elaborated on shortly.

A BVH is a tree structure where nodes are bounding volumes (usually *axis-aligned bounding boxes (AABB)*) that fit their children, and the leaves are geometric primitives like triangles and spheres. An example of a two-dimensional BVH is shown in Figure 3.1. A high-quality BVH has nodes with little overlap, such that little backtracking is required to determine whether the hit in one branch is really the closest hit. One heuristic that can be used to construct a BVH very quickly but while still achieving some measure of spatial coherency, and therefore quality, is to sort primitives along the *Z-order curve*.

The Z-order curve is a way of mapping points in $N \geq 2$-dimensional space to a single dimension without losing the locality of the points – points close to each other

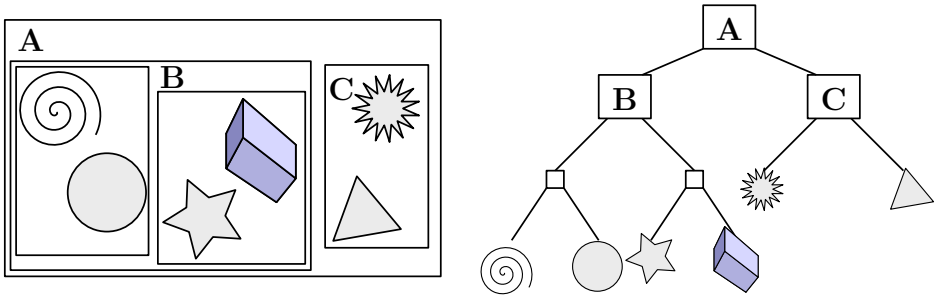


Figure 3.1: A visualisation of a two-dimensional BVH. To the left is the scene with geometry grouped in bounding boxes. To the right is the BVH-tree formed.
By Schreiberx / CC-BY-SA 3.0

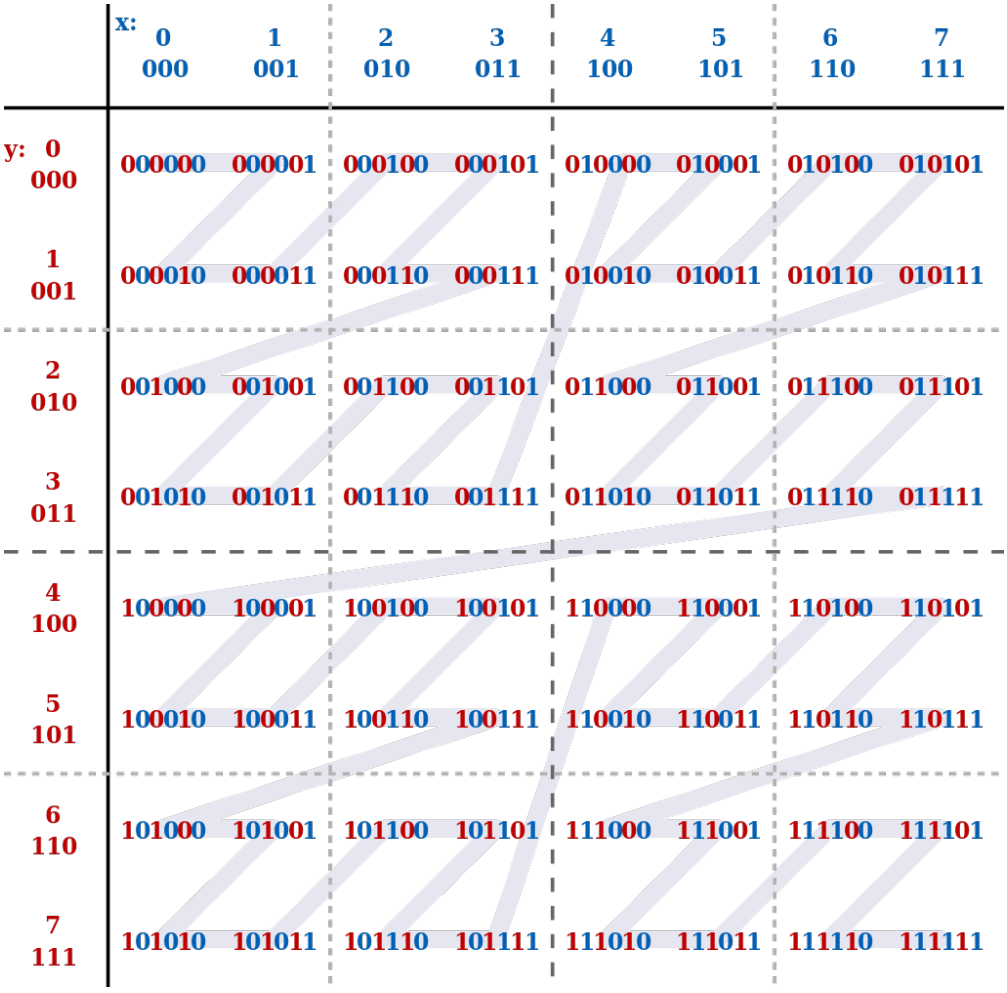


Figure 3.2: The Z plane-filling curve formed by interleaving the binary representations of the coordinates (x,y) and sorting the resulting interleaved binary numbers.
By David Eppstein / Public domain

in N-D space remains close to each other in the Z-order. It produces an ordering of the points that can then be used for standard data structures such as binary search trees. In particular, a 3D point is represented by a *Morton code*, which is produced by interleaving the binary representations of its coordinates, as shown in Figure 3.2. Listing 4 shows how this is implemented in Futhark. This is the main idea behind the *linear bounding volume hierarchy (LBVH)* of [Lauterbach et al., 2009], which is the base that is improved upon in [Karras, 2012].

```
let morton_n_bits: u32 = 30
let morton_component_n_bits: u32 = morton_n_bits / 3
let morton_component_max_val: f32 =
  f32.u32 (2**morton_component_n_bits - 1)

-- Expands a 10-bit integer into 30 bits by inserting 2 zeros after
-- each bit.
let expand_bits (x: u32): u32 =
    let x = (x * 0x00010001) & 0xFF0000FF
    let x = (x * 0x00000101) & 0x0F00F00F
    let x = (x * 0x00000011) & 0xC30C30C3
    let x = (x * 0x00000005) & 0x49249249
    in x

-- Calculates a 30-bit Morton code for the given 3D point located
-- within the unit cube [0,1].
--
-- Morton code: (X0X1X2..., Y0Y1Y2..., Z0Z1Z2...)
--           => X0Y0Z0X1Y1Z1X2Y2Z2...
let morton3D (v: vec3): u32 =
  let { x, y, z } = vmin (vec3.scale (morton_component_max_val + 1) v)
                         (mkvec3_repeat morton_component_max_val)
  let (xx, yy, zz) = ( expand_bits (u32.f32 x)
                     , expand_bits (u32.f32 y)
                     , expand_bits (u32.f32 z) )
  in xx * 4 + yy * 2 + zz
```

Listing 4

What makes the LBVH particularly suitable for our purposes is that it is designed with the constraints of GPU hardware in mind. Normally, it's common to allocate the nodes in a tree structure on the heap, and represent the edges to the children with pointers. In Futhark however, and in GPU programming in general, heap memory can't be dynamically allocated from within a kernel. Karras takes these constraints of the GPU into consideration, and the LBVH is contained in only two flat arrays (i.e., it is stored linearly, hence the name) – one containing the actual elements (the leaves), and one containing the internal nodes of the tree in a specific

layout.

After implementing the tree construction[1], what remains in order to finish the ray-scene intersection algorithm is implementing the *closest_hit* and *any_hit* functions. The implementation of the closest hit function is quite straightforward. The nodes in the tree are simply visited in depth-first order, and whether to keep traversing a branch or not is decided by testing the AABB of the node for intersection with the ray. The linear nature of the structure, and the lack of recursion in Futhark introduces a bit of boilerplate that is needed for keeping track of parents and indices. The `any_hit` function is not shown here but it is implemented similarly, although it is somewhat simpler since only the first hit is returned, regardless of the order.

```
let closest_hit (tmax: f32) (r: ray) (bvh: bvh)
                : maybe (t, hit) =
  let (closest, _, _, _) =
    loop (closest, tmax, current, prev) = (-1, tmax, 0, #internal (-1))
    while current != -1
    do let node = unsafe bvh.nodes[current]
       let rec_child: maybe ptr =
         if prev == node.left
         then #just node.right
         else if prev != node.right && hit_aabb tmax r node.aabb
         then #just node.left
         else #nothing
       in match rec_child
          case #nothing -> (closest, tmax, node.parent, #internal current)
          case #just ptr ->
            match ptr
            case #internal i -> (closest, tmax, i, #internal current)
            case #leaf i ->
              match hit_triangle tmax r (G.get_geom (unsafe bvh.leaves[i]))
              case #just hit -> (i, hit.t, current, ptr)
              case #nothing -> (closest, tmax, current, ptr)
  in maybe.when (closest >= 0)
     <| let a = unsafe bvh.leaves[closest]
        in maybe.map (\h -> (a, h)) (hit_triangle tmax r (G.get_geom a))
```

Finally, we wrap the call to `closest_hit` in a function that associates a hit with the material of the hit triangle. An `.obj` file has already been parsed in order to figure out the material of each triangle. Each triangle thus has an associated material index (`mat_ix`), and the function takes the list of materials where this index can be used to find the correct material. Notice the use of the `unsafe` keyword when indexing the materials array. This is because the length isn't known at compile time, but Futhark requires this information to ensure that index operations are safe. The `unsafe` keyword tells the compiler that we're aware that the indexing could be out

[1]We do not go into any more detail here, as our implementation is rather long, but not that interesting – it is very close to what is described in [Karras, 2012]

of bounds and thus unsafe – it is then very important to make sure that it is really not out of bounds.

```
let closest_interaction
  (tmax: f32) (lr: lightray) (ms: []material) (xs: obj_bvh)
: maybe interaction =
  maybe.map (\(o, h) -> { h
                        , mat = unsafe ms[i32.u32 o.mat_ix]
                        , wavelen = lr.wavelen })
            (obj_bvh.closest_hit tmax lr.r xs)
```

3.1.2 Parallelising over all pixels

The "embarrassingly parallel" nature of ray tracing can mostly be attributed to the fact that each pixel of the image can be rendered independently; no mutable state has to be shared between the threads. To trace all pixels in parallel, we simply use the function `tabulate_2d : (n: i32) -> (m: i32) -> (f: i32 -> i32 -> a) -> *[n][m]a`, which given dimensions and a function from an index to a value, generates values for each index in the dimensions, returning a 2-dimensional array of the values. The Futhark compiler knows that each slot in `tabulate_2d` can be computed in parallel, so it generates GPU code which distributes the workload appropriately. It is not necessary for us as programmers to understand how the compiler does this – we get efficient parallel code without having to think about it, and this is the main advantages of the approach to parallel programming which Futhark takes.

```
let sample_pixels (rng: rnge)
                  (w: u32, h: u32)
                  (scene: accel_scene)
                  (cam: camera)
                  (ambience: spectrum)
                : (rnge, [][](ray, [path_len]pixel_sample)) =
  let rngs = rnge.split_rng (i32.u32 (w * h)) rng
  let sample' rng ji =
    sample_pixel scene cam ambience (f32.u32 w, f32.u32 h) ji rng
  let img = tabulate_2d (i32.u32 h) (i32.u32 w) <| \i j ->
    let ix = i * i32.u32 w + j
    let rng = rngs[ix]
    in sample' rng (u32.i32 j, u32.i32 i)
  in (advance_rng rng, img)
```

3.2 Path tracing

Some applications of ray tracing don't strive to be physically based and accurate, instead preferring performance as long as the perceived visual quality remains "good enough". For our purposes however, properly simulating how light behaves in real

life is very important, so we make use of a ray tracing method from the area of physically based rendering called path tracing.

Path tracing is a recursive ray tracing algorithm that provides a numerical solution to the *rendering equation* by employing *Monte Carlo integration*. The result is a method that does not introduce any statistical bias and can produce images indistinguishable from photographs, when used with high quality 3D models and material descriptions.

3.2.1 Monte Carlo integration

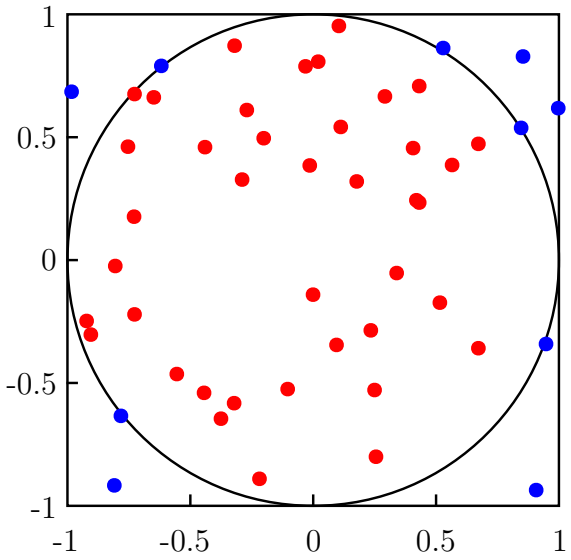


Figure 3.3: Computing the area of a circle with Monte Carlo integration. Image taken from Wikipedia, where it was distributed under a CC0 license.

At a high level, Monte Carlo methods are simply algorithms that apply repeated random sampling to numerically solve integrals, sample probability distributions, etc. In other words, it turns a continuous integral

$$I = \int_a^b f(x)dx$$

into a discrete, finite, and most importantly, easily computable sum

$$I \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i)$$

For example, the area of the unit circle can be represented as a 2D integral, which can be estimated by sampling random points (x, y) in $[-1, 1]^2$ and having the integrand return 1 if $x^2 + y^2 < 1$, and 0 otherwise. A visualisation of this can be seen in Figure 3.3, where a number of points have been randomly sampled according to a uniform random distribution. The points are red where the integrand is 1, and blue where it's 0. There are 50 points in total, and for 40 of them, $f(x, y) = 1$. The area can then be estimated by

$$
\begin{aligned}
I &= \int_{y_a}^{y_b} \int_{x_a}^{x_b} f(x,y)dxdy \\
&\approx \frac{(y_b - y_a)(x_b - x_a)}{N} \sum_{i=1}^{N} f(x_i, y_i) \\
&= \frac{(1-(-1))(1-(-1))40}{50} \\
&= 3.2
\end{aligned}
$$

The area is estimated to be 3.2 units by the Monte Carlo integration, which is indeed approximately $r^2\pi = 1^2\pi = \pi$, as expected. Increasing the number of samples N improves the estimate, and as N approaches infinity the result converges towards the correct value of the integral.

Russian roulette

Russian roulette is an important technique for improving the efficiency of Monte Carlo estimations [Pharr et al., 2016, chapter 13.7]. The idea of Russian roulette in sampling is very intuitive: select samples which are likely to contribute more to the final result more frequently.

3.2.2 The rendering equation

The rendering equation was first introduced by [Kajiya, 1986], and simultaneously by [Immel et al., 1986]. Kajiya describes it as a new formulation – more suitable for computer graphics – of an equation that has already been studied in the literature on radiative heat transfer. Realistic rendering techniques attempt to solve this equation.

$$L_o(\mathbf{x}, \omega_o, \lambda) = L_e(\mathbf{x}, \omega_o, \lambda) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda) L_i(\mathbf{x}, \omega_i, \lambda)(\omega_i \cdot \mathbf{n}) d\omega_i$$

The equation is based on the law of conservation of energy – that is, the outgoing radiance from a point $\mathbf{x}$ in the exitant direction ω_o at wavelength λ is equal to the radiance emitted in that direction plus the fraction of all incident radiance that is reflected in that direction.

The rendering equation consists of an integral that can't be solved analytically – but by using random sampling the value of an integral can be estimated. Kajiya shows how to use a Monte Carlo method to achieve this, and introduces the term *path tracing* to denote this new form of physically based ray tracing. The name comes from the fact that the sequence of sampled directions comprise the light path that a single photon (or a packet of photons) can take, illustrated in Figure 3.4. We don't implement the algorithm in the exact form shown above, but rather make use of a common modification that takes direct lighting into special consideration, as described in Section 3.4.1.

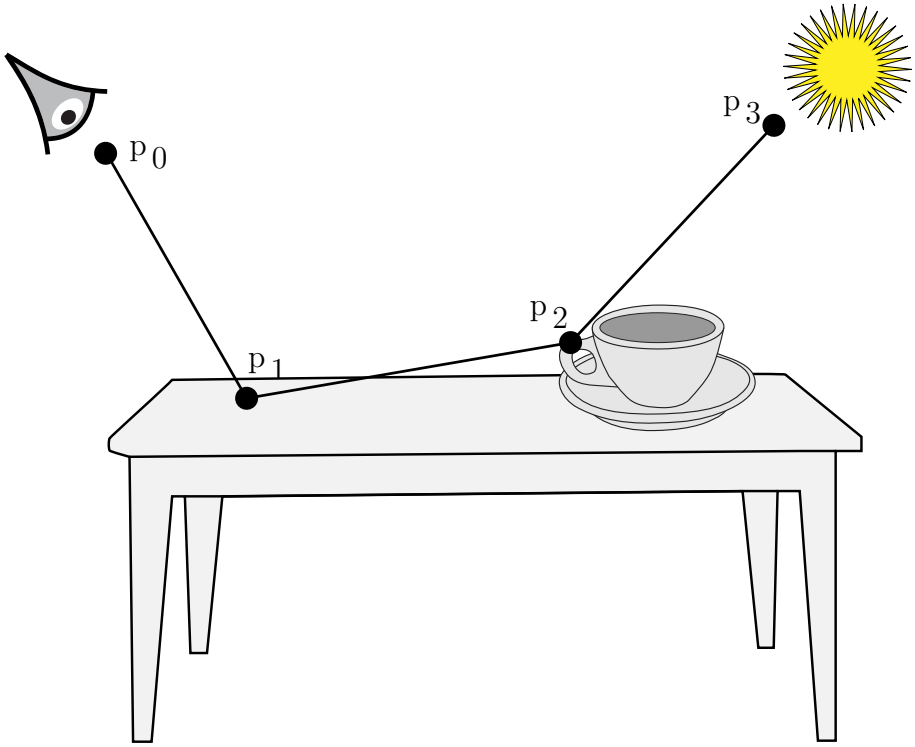


Figure 3.4: One sampled path of light created by path tracing.
From [Pharr et al., 2016].

3.2.3 Implementation

The full listing of our implementation of the path tracing integrand can be seen in Listing 5. Following is a description of the function step by step.

As we need the integration to be applicable to LIDAR and not only vision, we cannot simply accumulate the radiance. Instead, we return the full path of hits for the caller to manipulate as needed for the purpose. The result will be stored in a matrix, so the length of the arrays returned from the function must be constant. Empirically, we find a path length of 16 points strikes a good balance between being accurate and not using too much GPU memory. Note that using a finite path length means that our integrator is technically no longer statistically unbiased, but in practice this should not make much of a difference.

```
let path_len: i32 = 16

type path = [path_len]{ distance: f32, radiance: f32 }
```

In `path_trace`, the recursive definition in the rendering equation first has to be converted to use a sequential loop.

```
loop (path, i, distance, should_continue, lr, rng) =
     (dark_path, 0, 0, true, lr, rng)
while should_continue && i < path_len
do match closest_interaction tmax lr scene.mats scene.objs
```

```
    case #just inter ->
        ...
```

For each point along the path, the direct radiance (see Section 3.4.1) incident to the point is computed first. This value together with the current distance comprise one point of the `path`.

```
let (rng, direct_radiance) =
  direct_radiance rng wo inter scene
let radiance = direct_radiance
  + if i == 0
    then spectrum_lookup lr.wavelen inter.mat.emission
    else 0
let distance = distance + inter.h.t
let path = path with [i] = { distance, radiance }
```

Sampling the next points along the path corresponds to computing the indirect lighting term of the integral (see Section 3.3). The scattering direction which will give us the next point along the path is given by *sampling the bidirectional scattering distribution function (BSDF)* (see Section 3.3.3).

```
let (rng, { wi, bsdf, pdf }) = sample_dir wo inter rng
let pdf = match pdf
          case #impossible -> 0
          case #delta -> 1
          case #nonzero x -> x
let cosFalloff = f32.abs (vec3.dot inter.h.normal wi)
```

Finally, we use a Russian roulette method to terminate with a probability based on the light throughput for the surface BSDF sample. This would have ensured an unbiased result, had we not limited the path length. It is still useful, as it ensures we sample paths where less light is absorbed more often than those where no light can pass through.

```
let p_terminate = 1 - bsdf * cosFalloff / pdf
let (rng, q) = random_unit_exclusive rng
in if pdf == 0 || q < p_terminate
   then finish path
   else let lr = lr with r = mkray_adjust_acne inter.h wi
        in continue path (i + 1) distance lr rng
```

To combine the samples of `path_trace` into the estimated integral, we simply compute the moving average of the image returned by sampling all pixels. The higher the number of frames in the accumulation, the more the result converges towards the ground truth over time.

```
let sample_frame_accum [m] [n] (s: state): (rnge, [m][n]vec3) =
  let (rng, img_new) = sample_frame s
  let nf = f32.u32 s.n_frames
  let merge_all merge_one =
    (rng, map2 (map2 merge_one) s.img (img_new :> [m][n]vec3))
  in merge_all (\acc c -> (vec3.+) (vec3.scale ((nf - 1) / nf) acc)
                                   (vec3.scale (1 / nf) c))
```

3.3 Indirect lighting

To create one path of the Monte Carlo integration, we need to be able to sample a reflection or refraction direction to find the next point along the path. Starting in the eye, the exitant direction is the direction from the first surface intersected in the scene, through the virtual pixel, to the eye. Next, an incident direction is sampled from the second point to the first, then an incident direction is sampled from the third point to the second, and so on.

`dir_sample` is a structure representing a sampled incident direction, and contains as well the BSDF-value (see Section 3.3.2) of the composed junction and the probability that the particular direction was sampled with (see Section 3.3.3).

```
type dir_sample = { wi: vec3, bsdf: f32, pdf: sample_pdf }
```

The top-level direction sampling function and most of the rest are of the form `vec3 -> interaction -> rnge -> (rnge, dir_sample)` – given an exitant direction, a surface interaction, and a random number generator, return the new state of the random number generator and the incident direction sample. All the direction sampling functions except the top-level further assumes that the normal is $(0, 0, 1)$ in order to simplify calculations. As such, `sample_dir` has to transform the vector into the local space before calling into the hierarchy, and inversely transform the result back into world space afterwards.

```
let sample_dir (wo: vec3) (i: interaction) (rng: rnge): (rnge, dir_sample) =
  let onb = mk_orthonormal_basis i.h.normal
  let wo' = world_to_local onb wo
  let (rng, s) = uber_sample_dir wo' (material_at_wavelen i.mat i.wavelen) rng
  in (rng, s with wi = local_to_world onb s.wi)
```

3.3.1 The material hierarchy

Light will behave very differently depending on the properties of the surface that it hits. Fundamentally, three things can happen when light hits a surface: it can be reflected, refracted and transmitted, or absorbed. Usually, it is a combination of the three. The following are the principal material types, which we compose in a hierarchy. This hierarchy is a convenient approximation of material blends, inspired by the *UberMaterial* of [Pharr et al., 2016, Chapter 9.2.5], which is described as "[a] "kitchen sink" material representing the union of [the following materials]."

```
let path_len: i32 = 16

type path = [path_len]{ distance: f32, radiance: f32 }

let path_trace (lr: lightray)
               (scene: accel_scene)
               (ambience: spectrum)
               (rng: rnge)
             : path =
  let dark_path = replicate path_len { distance = f32.inf, radiance = 0 }
  in (.0) <|
     loop (path, i, distance, should_continue, lr, rng) =
          (dark_path, 0, 0, true, lr, rng)
     while should_continue && i < path_len
     do match closest_interaction f32.highest lr scene.mats scene.objs
        case #just inter ->
          let rng = advance_rng rng
          let wo = vec3_neg lr.r.dir
          let (rng, direct_radiance) =
            direct_radiance rng wo inter scene
          let radiance = direct_radiance
            + if i == 0
              then spectrum_lookup lr.wavelen inter.mat.emission
              else 0
          let distance = distance + inter.h.t
          let path = path with [i] = { distance, radiance }
          let (rng, { wi, bsdf, pdf }) = sample_dir wo inter rng
          let pdf = match pdf
                    case #impossible -> 0
                    case #delta -> 1
                    case #nonzero x -> x
          let cosFalloff = f32.abs (vec3.dot inter.h.normal wi)
          let p_terminate = 1 - bsdf * cosFalloff / pdf
          let (rng, q) = random_unit_exclusive rng
          in if pdf == 0 || q < p_terminate
             then finish path
             else let lr = lr with r = mkray_adjust_acne inter.h wi
                  in continue path (i + 1) distance lr rng
        case #nothing ->
          let ambience = spectrum_lookup lr.wavelen ambience
          in finish (path with [i] = { distance = f32.inf
                                     , radiance = ambience })
```

Listing 5: The full implementation of the path tracing integrand

Diffuse materials *Diffuse* materials scatter the light that hits them in all directions. You can think of diffuse materials as matte surfaces, like matte paint, rough wood, or cloth. They are implemented with the *Lambertian* model which is very simple and a good approximation for many real-life matte surfaces [Pharr et al., 2016, Chapter 8.3].

Dielectrics *Dielectrics* are materials that don't conduct electricity. They reflect some ratio of light that hits them, and refract the rest. The reflectance distribution and reflection direction depend on various factors such as roughness and refractive index. The refractive index, while a complex number in general, doesn't have any imaginary part for dielectrics in practice. The implication of this is that reflections are not coloured by the material, but instead reproduce the same colours, like mirrors. The refraction direction is simply calculated with Snell's law [Pharr et al., 2016, Chapter 8.2]. Glass, water, and plastic are examples of common dielectrics.

The model used for the implementation is an early *microfacet* model introduced by [Torrance and Sparrow, 1967]. As a microfacet model, the *Torrance-Sparrow* model statistically models the scattering of light from a distribution of perfectly smooth, mirrored microscopic surfaces. This model works well for most materials due to the fact that so many microfacets would fit in a pixel that the statistical behaviour looks correct, but it breaks down when the a microfacet would in reality cover a whole pixel or more by itself. Glittery car paint is an example of a material that can behave like this.

Conductors *Conductors* (that is, metals) behave much like dielectrics and share the same reflectance distribution model in our implementation. Conductors differ from dielectrics in that their refractive indices do have an imaginary part, which in practice means that not all wavelengths are reflected to the same extent, so reflections are coloured. Further, refracted light is quickly absorbed rather than transmitted.

The kitchen sink

```
type material =
  { color: spectrum
  , roughness: f32
  , metalness: f32
  , ref_ix: f32
  , opacity: f32
  , emission: spectrum }
```

Collectively, all the material parameters are compiled in a single type `material`, which is interpreted in a hierarchical fashion, using a simple Russian roulette method at each level when sampling to select one of two different "branches". For example, when sampling the material BSDF for a scattering direction, we start at the root in `uber_sample_dir`. There, we apply Russian roulette to either treat the surface

as a conductor or a dielectric, with the `metalness` parameter as the probability to select the conductor branch.

```
let uber_sample_dir (wo: vec3) (m: material') (rng: rnge)
                    : (rnge, dir_sample) =
  let (rng, p) = random_unit_exclusive rng
  in if p < m.metalness
     then metal_sample_dir wo m rng
     else dielectric_sample_dir wo m rng
```

3.3.2 BSDF

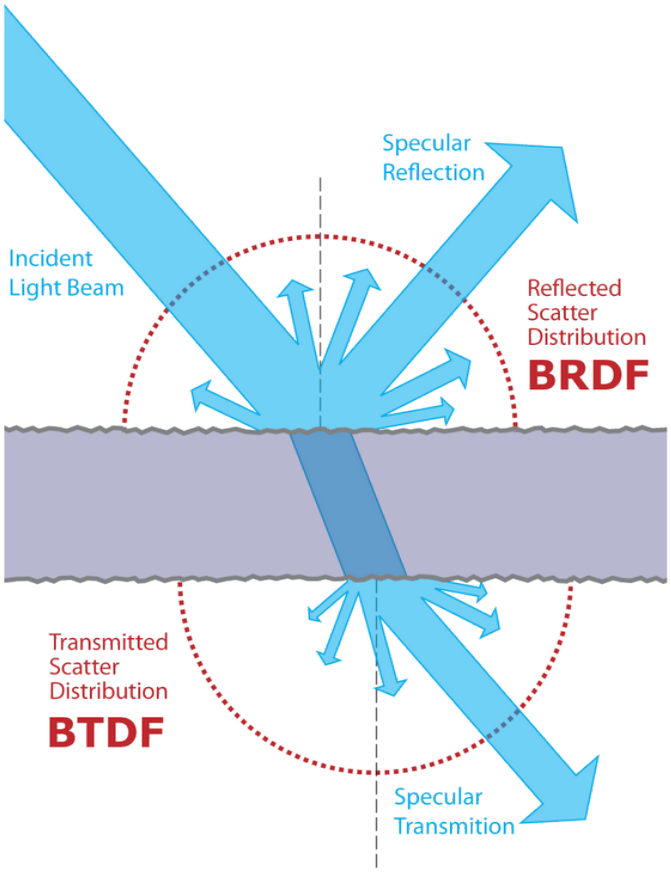


Figure 3.5: Illustration of a bidirectional scattering distribution for some arbitrary surface. By Wikipedia user Jurohi. CC BY-SA 3.0.

The *bidirectional scattering distribution function (BSDF)* describes how light that hits a surface is scattered. Given an incident direction vector, an exitant direction vector, the surface normal, and the surface material, the BSDF tells us what fraction of light passes through that junction without being absorbed. This idea is illustrated in Figure 3.5.

The type signature for a BSDF function is `(wo: vec3) -> (wi: vec3) -> (m: material') -> f32`. Note that there is no explicit parameter for the normal

– like with the direction sampling functions, the normal is always $(0, 0, 1)$ in order to simplify calculations, and the top-level BSDF function handles transforming rays into and out of the local space. As an example, the following is the BSDF function for the conductor/metal material. The implementation reflects the fact that metallic reflections are much like dielectric reflections, but coloured.

```
let metal_bsdf (wo: vec3) (wi: vec3) (m: material'): f32 =
  m.color * dielectric_reflection_bsdf wo wi m
```

Now, when scattering an exitant ray at a surface, we could randomly pick a vector in the sphere and simply apply the BSDF function to simulate rays being absorbed more at certain angles than others. The more samples we take, the better our estimation of the irradiance at the point, and the radiance along the exitant vector. This is the essence of the Monte Carlo approach to ray tracing.

3.3.3 Importance sampling

Not all incident directions are equally useful to sample, though. Just as the BSDF tells us, there will be more light travelling in some directions, and less in others. Naïvely sampling directions randomly in a sphere would result in most samples having a low BSDF value and therefore contributing very little to the result, while a few would have a very high BSDF value and produce small and sparse bright spots – that is, the result would be noisy. We could greatly reduce the noise if we could somehow sample scattering directions with a distribution similar or equal to the BSDF – that is, sample directions where the amount of scattered light is lower less often, and completely disregard angles where we know that there can be no light at all.

Noise in our ray traced images is equivalent to statistical variance in the Monte Carlo integration. *Importance sampling* is a generalisation of the idea described in the previous paragraph, and is a powerful technique for reducing statistical variance in Monte Carlo integrations by generating samples according to a distribution $p(x)$ that is similar to the function in the integrand $f(x)$ [Pharr et al., 2016, Chapter 13.10]. In our case, $f(x)$ is the BSDF function.

Importance sampling is particularly good for more specular materials, as the distribution we'd like to sample is very narrow. For perfectly specular mirrors, importance sampling (if it can even be called that when we always pick the single reflection direction) is a must, as the distribution is a unit impulse.

As an example, `dielectric_reflection_sample_wh` is a direction sampling function for the so-called *half-way vector* of a dielectric reflection. It uses importance sampling of the *Beckmann* distribution [Beckmann and Spizzichino, 1987] to approximately sample the lobe-shaped distribution of a glossy reflection given by the Torrance-Sparrow microfacet BSDF we use. An illustration of a Beckmann microfacet distribution can be seen in Figure 3.6.

```
let dielectric_reflection_sample_wh (wo: vec3) (m: material') (rng: rnge)
                                    : (rnge, vec3, f32) =
  let (rng, (u0, u1)) = random_in_unit_square rng
```

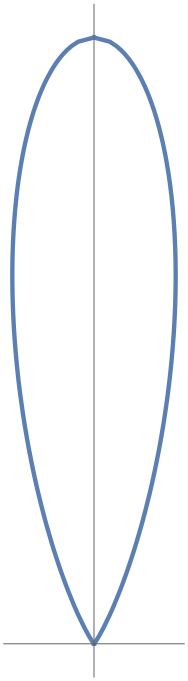


Figure 3.6: Full Beckmann microfacet distribution $D(\omega_h)$ for a roughness of $\alpha = 0.3$.
From [Pharr et al., 2016]. CC BY-NC-SA 4.0

```
let log_sample = f32.log (1 - u0)
in if f32.isinf log_sample then (rng, mkvec3 0 0 0, 0) else
let alpha = beckmann_alpha m.roughness
let tan2_theta = -alpha * alpha * log_sample
let phi = u1 * 2 * f32.pi
let cos_theta = 1 / f32.sqrt (1 + tan2_theta)
let sin_theta = f32.sqrt (f32.max 0 (1 - cos_theta * cos_theta))
let wh = spherical_direction sin_theta cos_theta phi
let wh = if same_hemisphere wo wh then wh else (vec3_neg wh)
let pdf_wh = microfacet_distribution alpha wh * f32.abs cos_theta
in (rng, wh, pdf_wh)
```

3.4 Ray tracing for LIDAR simulation

While a LIDAR sensor is similar to a vision camera in many ways, it differs in some key aspects. In order for the implementation to be able to simulate a LIDAR sensor, certain features must be added. *Direct lighting integration* is needed to improve the efficiency of the Monte Carlo integrator and bring it up to usable speeds. *Spectral path tracing* is strictly necessary in order for the result to be physically accurate.

3.4.1 Direct lighting

When a surface is very rough and has a very wide BSDF or when the light source is very small and far away from an intersection point (i.e., the solid angle is small), the chance of hitting the light with a recursive ray is impractically small and the scene will take a very long time to converge. We render this effect with our ray tracer, the result of which can be observed in Figure 3.7. Here we can see that the reflections converge more efficiently in the top-left, and become noisier as we approach the bottom-right. The transmitter of a LIDAR transceiver is generally very small, so unless this problem is addressed, the performance of a LIDAR simulation will be very poor.

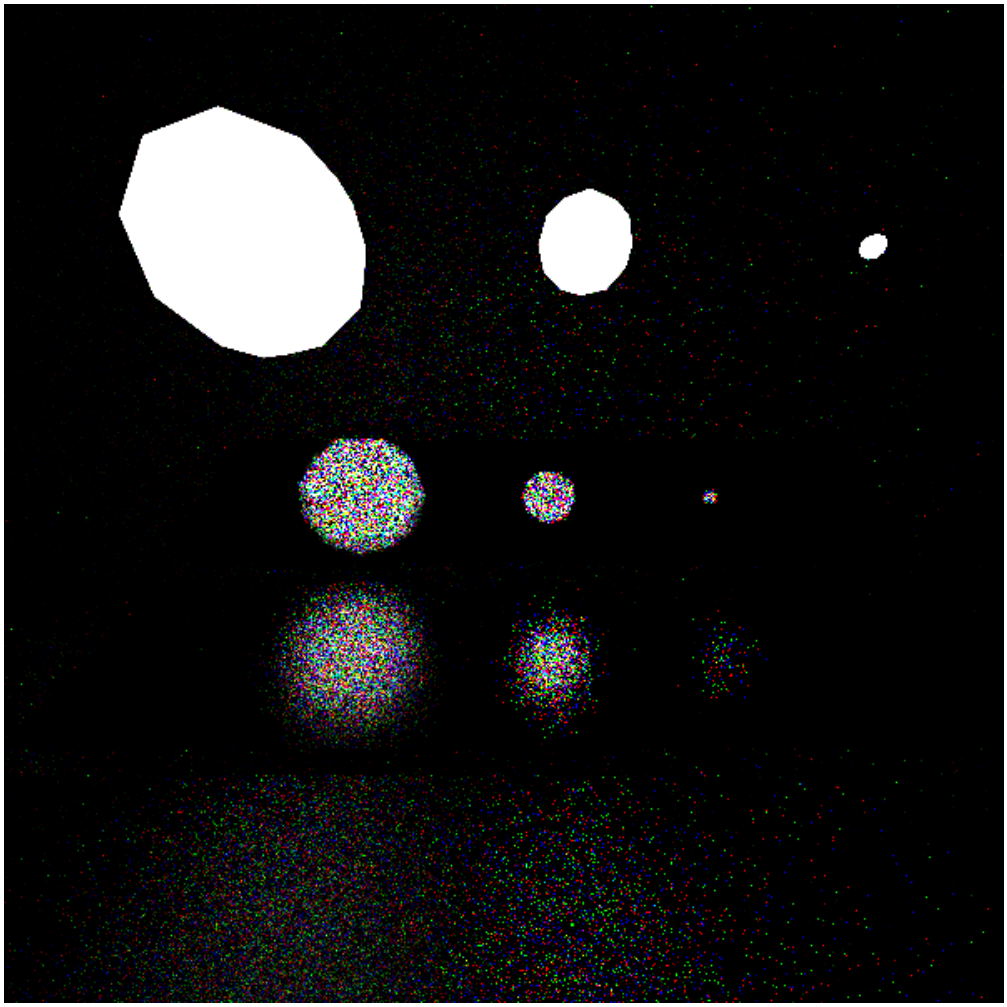


Figure 3.7: Sampling only the BSDF. The light source decreases in size from left to right, and the row of surfaces increase in roughness from top to bottom. Rendered with our ray tracer.

Just like we use importance sampling to only sample the incident directions for which the BSDF-value is non-zero when we sample an indirect ray, we would also like to somehow take the properties of the light source into special consideration. By directly using information about the light sources, the efficiency of the ray tracer can be improved for scenes where light sources with high emission can be seen with

a small solid angle. Informally, we can consider it as breaking the incident radiance factor in the Monte Carlo integral into a direct and an indirect term. More formally, we're *partitioning the integrand* [Pharr et al., 2016, Chapter 14.4.6].

Finding the direct radiance for a surface interaction and an exitant radiance is implemented by randomly picking one light source, estimating the radiance with regards to that light source, and then weighting the result by the total number of lights. Alternatively we could have exhaustively estimated the radiance for every single light. This would clearly reduce the variance, but would not scale well for scenes with many lights.

```
let direct_radiance (rng: rnge)
                    (wo: vec3)
                    (i: interaction)
                    (scene: accel_scene)
                  : (rnge, f32) =
  if null scene.lights
  then (rng, 0)
  else let (rng, l) = random_select rng scene.lights
       let (rng, radiance) = estimate_direct rng wo i l scene.objs (1, 1)
       let light_pdf = 1 / f32.i32 (length scene.lights)
       in (rng, radiance / light_pdf)
```

Figure 3.8 shows another render made with our ray tracer in the same amount of time, now with light source sampling instead of BSDF sampling. The efficiency is much better for more diffuse surfaces and when the light sources are smaller, but it has become worse in the top-left.

3.4.2 Multiple importance sampling

One problem that appears when we only sample the light source for direct lighting is that shiny surfaces now take longer to converge, as can be seen in the upper-left half of Figure 3.8. For diffuse materials, the probability is high that the BSDF will be high for a light sample since the scattering distribution is a uniform hemisphere, but for shiny materials, the distribution is quite narrow, and the probability that a ray to a random point on the light source is in that distribution is not so high. One might then be tempted to simply sample both the BSDF and the light source for direct radiance and average the two contributions, but this would not help, as variance/noise is additive. But there does exist a solution to sample two distinct probability distributions and get a third distribution that is the "best of both worlds". This is done with *multiple importance sampling (MIS)*, and the result can be observed in Figure 3.9.

MIS was introduced by Eric Veach in his landmark thesis *Robust Monte Carlo methods for light transport simulation* [Veach, 1997, Chapter 9]. Veach himself provides a good summary of the idea: "It is based on the idea of using more than one sampling technique to evaluate a given integral, and combining the sample values in a provably good way". Our implementation is based on this strategy, specifically the version presented in [Pharr et al., 2016, Chapter 14.3.1].

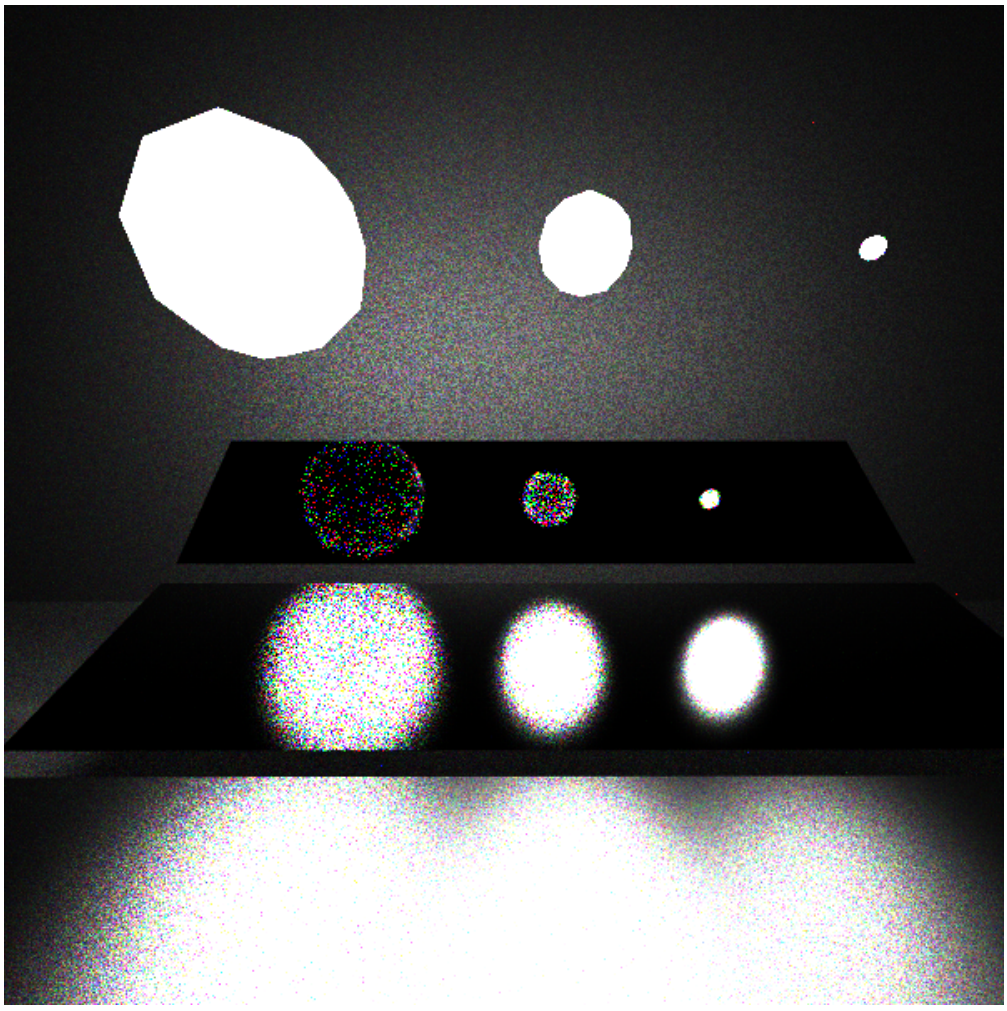


Figure 3.8: Sampling only the light. Same render duration as Figure 3.7.

3.4.3 Spectral path tracing

A photon has a particular wavelength. Light waves of different wavelengths will interact with surfaces in different ways, depending on material properties. Traditional path tracers are only used for rendering images, which means they only operate in the visible light spectrum. Further, they don't treat light as a continuous or sampled spectrum, but rather as a single unit with a red, green, and blue value [Peercy et al., 1995]. Doing it the traditional way is significantly faster, and still looks good and realistic in most scenarios – wavelength-dependent effects are often quite subtle when it comes to vision. However, in order to be able to simulate LIDAR and to properly simulate phenomena like dispersion, it is necessary to use *spectral ray tracing*, taking a whole spectrum of light into account.

We represent the colour and emission of a material with a `spectrum` type which describes a sampled spectrum with a certain finite number of points (more specifically, we used 6 non-uniform points for our prototype). When looking up an arbitrary frequency value in the `spectrum`, we use linear interpolation between the closest points to get the resampled intensity for the associated particular wavelength. See Figure 3.10.

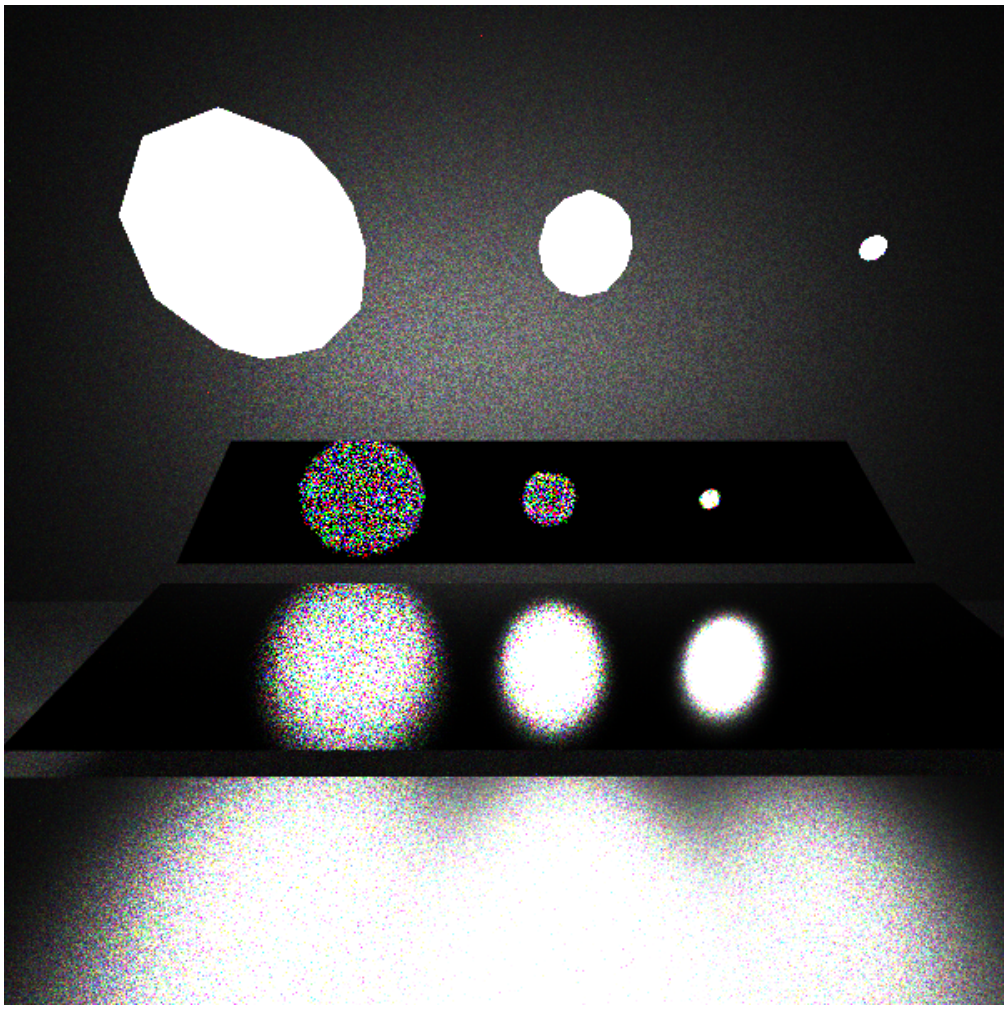


Figure 3.9: Using MIS to combine both BSDF sampling and light sampling. Rendered with our ray tracer for the same duration as Figure 3.7 and Figure 3.8.

To make the path tracing spectral, we introduce a new type which consists of a geometrical `ray` and an associated wavelength.

```
type lightray = { r: ray, wavelen: f32 }
```

The wavelength is sampled from the sensitivity distributions of the modelled camera or LIDAR sensor. The sensitivity distributions are represented as an association list of the sensitivity distribution of the particular "receptor" and approximated with a normal distribution, and a function that converts an intensity registered for this receptor to an RGB value for visualisation purposes.

For example, consider the sensor of a Canon 400D camera. A sampled spectrum is shown in Figure 3.11. We approximate a normal distribution for each receptor: blue, green, and red.

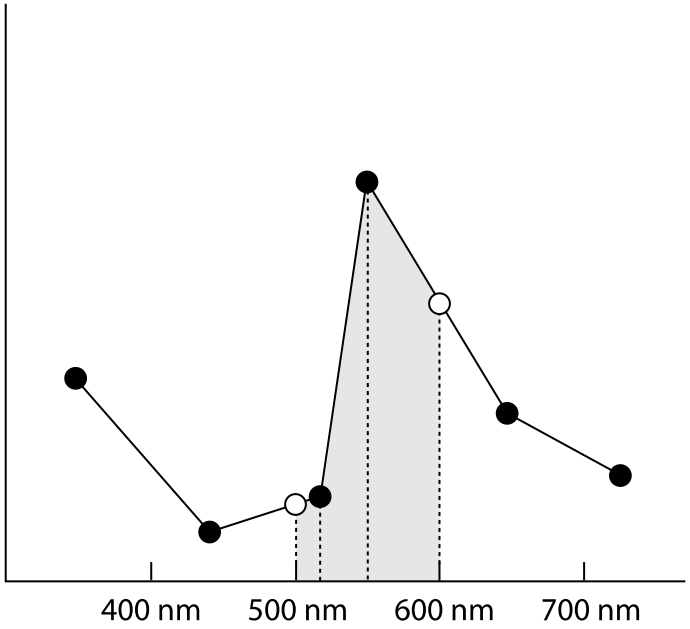


Figure 3.10: The black points constitute a non-uniform sampled spectrum with a resolution of 6. The white points are lookups into the spectrum, which implies resampling with linear interpolation between the closest points to the left and right.
From [Pharr et al., 2016].

```
let sensor =
  [ ({ mu = 455, sigma = 22 }, mkvec3 0 0 1)
  , ({ mu = 535, sigma = 32 }, mkvec3 0 1 0)
  , ({ mu = 610, sigma = 26 }, mkvec3 1 0 0) ]
```

In order to sample a wavelength from a normal distribution, the `statistics` Futhark library is used, which provides a function that samples a value from a normal distribution with *inverse transform sampling*.

```
let (rng, (wavelen_distr, wavelen_radiance_to_rgb)) =
  random_select rng sensor
let wavelen_radiance_to_rgb =
  vec3.scale (f32.i32 (length sensor)) wavelen_radiance_to_rgb
let (rng, p) = random_unit_exclusive rng
let wavelen = stat.sample (stat.mk_normal wavelen_distr) p
```

That light rays are of particular wavelengths has some implications to the BSDF-related computations of materials described in Section 3.3. Beyond enabling the description of colours outside the visible spectrum, it affects how light is refracted with Snell's law (see Section 3.3.1). While often ignored for convenience, the refractive index of a material is actually a function of wavelength. Another value affected by the refractive index is the Fresnel reflectance. Together, this allows rendering phenomena like dispersion. This can be be seen in Figure 3.12, which was rendered with our ray tracer.

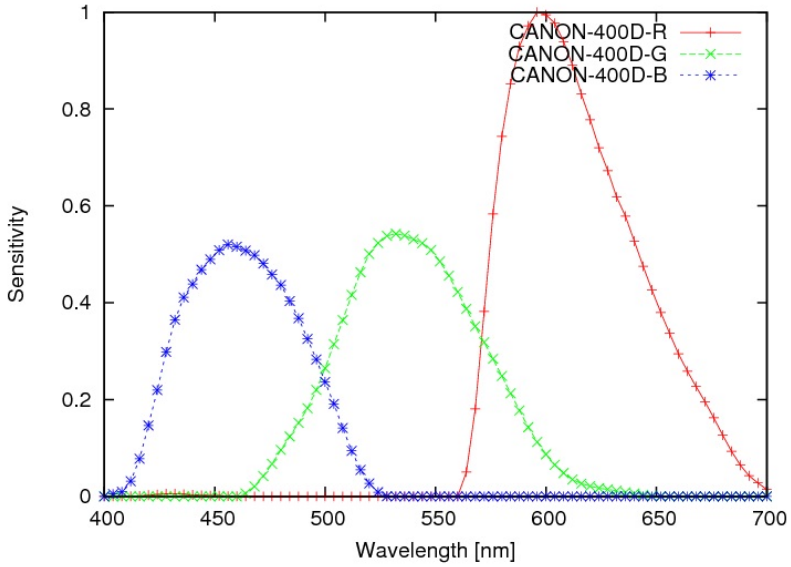


Figure 3.11: The sampled spectrums of the sensitivity distributions of a Canon 400D camera. Image from the database [Kawakami et al., 2009], associated with the paper [Kawakami et al., 2013]. CC BY license.

3.4.4 Transmitter properties

Beyond a receiver, what we call a LIDAR sensor also contains a transmitter. The transmitter is what generates the laser that interacts with the scene and then arrives back at the receiver, is absorbed, or escapes the scene. A vision camera can similarly have a transmitter as well, which is what we call a flash. We represent this transmitter with a type `transmitter`:

```
type transmitter
  = #flash { radius: f32, emission: spectrum }
  | #scanning { radius: f32, theta: angle, emission: spectrum }
  | #none
```

The `#flash` variant works like a camera flash – illuminating the entire field of view, and is essentially the same for each sampled pixel when rendering. On the other hand, a `#scanning` transmitter is displaced together with the directional receiver, meaning that a unique, narrow laser is fired for each pixel in the scan. For LIDAR, both scanning transmitters and flash transmitters are commonly used.

During the actual ray tracing, physical light sources are generated from the transmitter description, leveraging our already existing code for lights.

```
let gen_transmitter (c: camera) (r: ray): []light =
  let n_sectors = 8 in
  map (\l -> #arealight l)
  <| match c.conf.transmitter
     case #flash { radius, emission } ->
       map (\t -> #diffuselight { geom = t, emission })
```

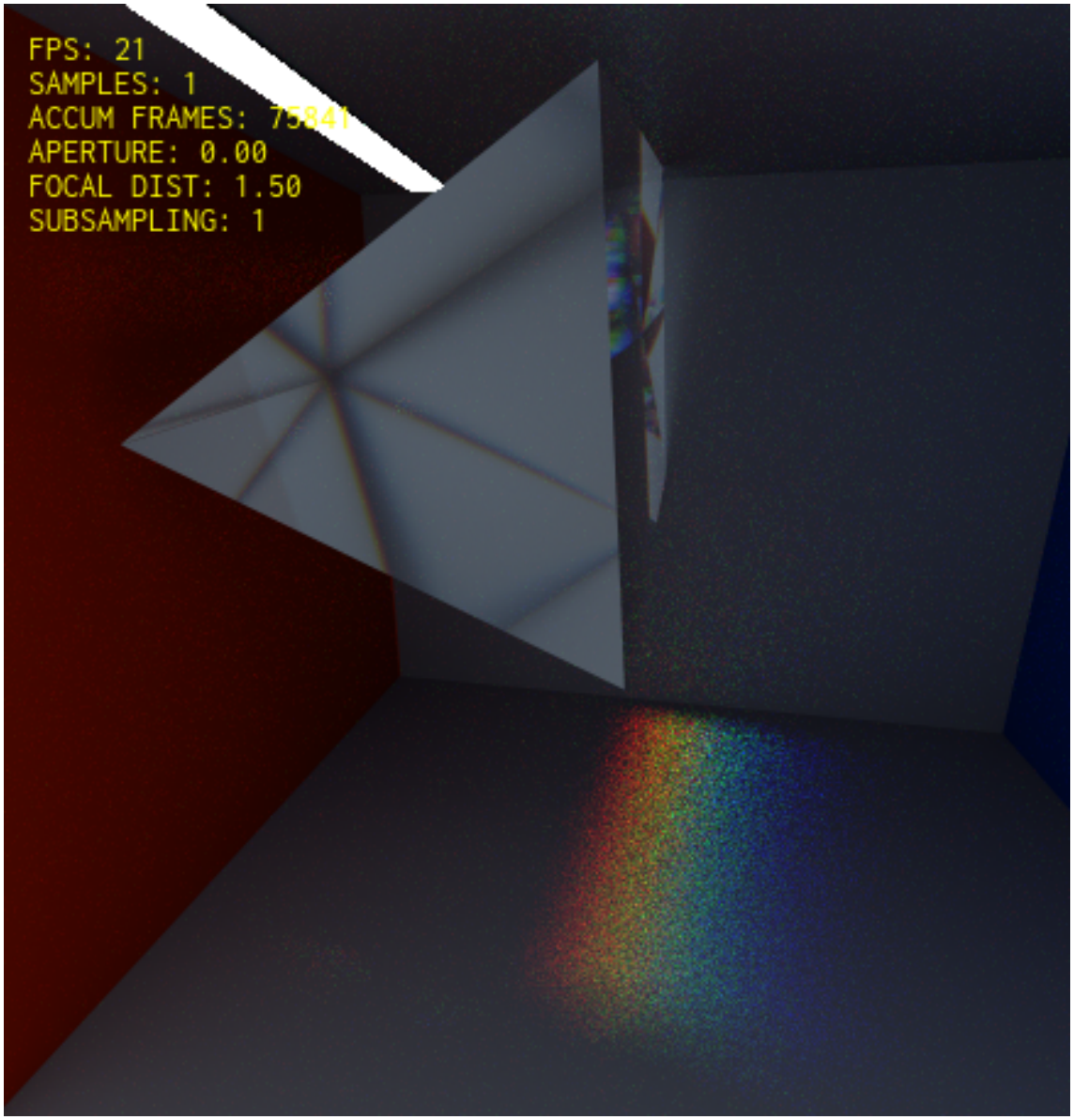


Figure 3.12: A render illustrating how our ray tracer can simulate spectral phenomena like dispersion through a prism. For effect, we've exaggerated the slope of the refractive index function. Normally the phenomenon wouldn't be as pronounced in this particular scene.

```
        (disk c.origin (cam_dir c) radius n_sectors)
    case #scanning { radius, theta, emission } ->
      map (\t -> #frustumlight { geom = t, theta, emission })
        (disk c.origin r.dir radius n_sectors)
    case #none -> []
```

3.4.5 Time of flight and distance

Perhaps the most important difference between LIDAR depth sensors and regular cameras is that LIDAR sensors measure the distance that light travels, rather than the colour of a particular pixel. The distance is trivially calculated from the *time of flight* – the time which transmitted light takes to return to the sensor. Together with the information about the direction in which the laser was fired, a 3D point in space can be derived. The generated spatial coordinates compose what is called a *point cloud*, a visualisation of which can be seen in Figure 2.4.

It should be noted that even when the simulation is physically accurate, the produced point cloud may not represent an accurate depth map of the scene. This is also the case with real LIDAR sensors in the physical world. This is because the time of flight can not be directly translated to spatial distance. It's possible for a laser to return to the sensor not by directly reflecting back on the first interaction in the scene, but via indirect reflections. The calculated distance in such cases will not match the depth to the scene.

The time of flight is recorded as part of the path tracing integrand in `path_trace`, cf. Section 3.2.3.

3.5 Declaring the devices

Sensor models can be specified quite concisely in a declarative style in our implementation. A value of type `device_config` has to be created, and then passed to the path tracing integrator.

```
-- A scanning LIDAR sensor
let lidar_conf: device_config =
  { offset_radius = 0.01
  , sensor = [ ({ mu = 1550, sigma = 10 }, mkvec3 1 0 0) ]
  , transmitter = #scanning { radius = 0.01
                            , theta = from_deg 3
                            , emission = uniform_spectrum 1500 } }

-- A conventional vision camera
let visual_conf: device_config =
  { offset_radius = 1
  , sensor = [ ({ mu = 455, sigma = 22 }, mkvec3 0 0 1)
             , ({ mu = 535, sigma = 32 }, mkvec3 0 1 0)
             , ({ mu = 610, sigma = 26 }, mkvec3 1 0 0) ]
  , transmitter = #none }
```

We could also create a camera with a flash by modifying a copy of `visual_conf`.

```
let visual_flash_conf: device_config = visual_conf with transmitter =
  #flash { radius = 0.05
         , emission = map_intensities (* 1000)
                                      (blackbody_normalized 5500) }
```

4

Results

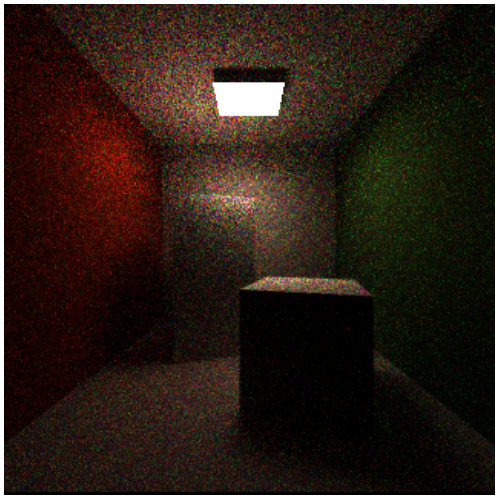
The finished implementation is a spectral path tracer which includes some features to improve the speed of convergence and make LIDAR simulation practically possible. Sensor devices can be defined with a declarative approach, and the ray tracer can simulate both a visual camera and LIDAR. The full source code is available under an open source license and can be found online at https://gitlab.com/JoJoZ/futhark-tracer. The repository includes instructions on how to build and run the ray tracer.

4.1 Performance

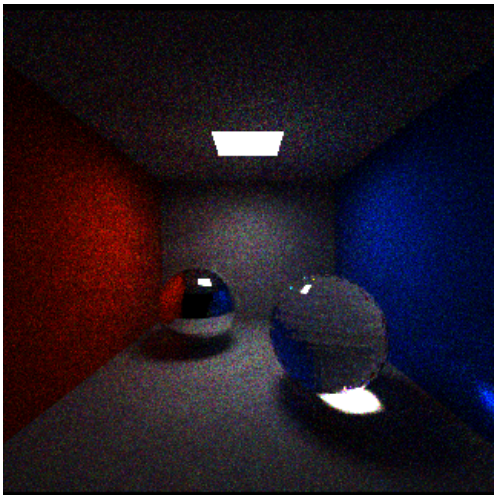
Instances of the ray tracer were compiled for four of the backends supported by Futhark: sequential C, multicore C, OpenCL, and CUDA. Both C backends execute on the CPU, while both the OpenCL and CUDA backend execute on the GPU. All backends except the sequential C backend utilize the parallelization features of Futhark.

Each instance was used to render two different scenes. The first scene was a traditional Cornell box with 44 triangles. The second was a box containing one metal sphere and one glass sphere, with a total of 2188 triangles. The images were rendered with 4096 samples per pixel. The measurements were made on a computer equipped with an Intel® Core™ i5-7600 CPU that has four cores running at 3.5GHz, and a Geforce GTX 1060 GPU with 6GB video RAM.

The results in Table 4.1 show the sequential C backend is unsurprisingly the slowest, while both of the GPU backends are more than an order of magnitude faster, apparently benefiting greatly from the parallelizing compilation of Futhark. Even the CPU multicore backend appears to have gained significantly from the parallelization by the compiler, being more than twice as fast as the sequential CPU backend.



(a) Cornell box



(b) Sphere box

Figure 4.1: Renders produced by the OpenCL backend during the benchmarks.

Backend	Time, Cornell	Time, Spheres
C, sequential	805s	1079s
C, multicore	366s	473s
OpenCL	28s	81s
CUDA	21s	49s

Table 4.1: Execution times for four different Futhark backends and two different scenes.

4.2 Validation

From a visual inspection of rendered images and point clouds, comparing them to images and point clouds captured by real devices on the real scene, it it is clear that the renders are fairly similar to the target scenes and that our ray tracer is at least in the right ballpark – that is, approximated material properties cause the expected sort of effects in the renders. What we present here are some somewhat more rigorous, albeit strictly relative metrics for evaluating improvements to the ray tracer.

It is not clear what the criteria for measuring absolute correctness actually are, so demonstrating a high degree of correctness is therefore outside our scope. Instead, we provide a framework that uses relative measures, which can be used to systematically improve the accuracy of the implementation. They also enable comparisons to other implementations.

Another reason for why the scope of validation has to be limited is that our methods for validation requires that real-world scenes be reproduced as virtual scenes as faithfully as possible. This requires substantial amounts of both time and 3D modelling knowledge.

4.2.1 Validation scene

A real scene representing the ground truth was constructed, containing several objects in a relatively light-isolated room. The room is shown in Figure 4.2 and contains a table upon which a plastic object is placed, with a magenta-coloured piece of paper placed between the plastic object's two panes. Geometric measurements were taken of the room and objects for the purpose of 3D-modelling the scene.

We then attempted to reproduce the scene in Blender, a software used for 3D-modelling. Figure 4.3 shows how the scene looked in Blender after we finished modelling it. Unfortunately, both the measurements of the scene and our expertise in the realm of 3D modelling were very far from perfect. For the material properties, we had no way to physically measure them, so we had to rely on intuition and visual inspection when defining them.

4.2.2 Validation of simulated LIDAR data

Information captured by a LIDAR sensor is represented by point cloud data. A point cloud is a list of plane coordinates along with a distance value which in practice means it's just a list of room coordinates. Fortunately, point cloud comparison is a well-established procedure and the tooling for doing so is quite mature. We chose to use the open-source CloudCompare [Girardeau-Montaut, 2016] software for our comparison.

The validation scene was captured using a LIDAR sensor, and the resulting point cloud from a single frame is shown in Figure 4.4. We then used our ray tracer to produce a point cloud from the virtual validation scene, as shown in Figure 4.5.

The two point clouds were then aligned in CloudCompare. The result is shown in Figure 4.6. The floors and walls aligned fairly well, but the table and its objects

Figure 4.2: The validation scene

did not. We attribute this to a combination of errors in the measurements of the room and mistakes in the process of modelling the scene in Blender.

The root mean square (RMS) distance was then calculated by CloudCompare, and it turned out to be 0.074. This number does not tell us anything about the quality of the simulation without more context, but the RMS distance can be used as a metric for measuring improvements to both the faithfulness of the virtual 3D environment as well as – more importantly – improvements to the ray tracer.

4.2.3 Validation of camera simulation

The result of rendering the validation scene with 1000 samples per pixel can be seen in Figure 4.7.

A procedure that is similar to the one used for LIDAR point cloud comparisons can be used for comparisons between camera images and images rendered with our ray tracer. One well-established metric for comparing the quality of images with the same or similar content is the structural similarity index (SSIM). Like the RMS distance, the SSIM is a fairly arbitrary number, unbounded above zero. It does not say anything on its own, but can be used to measure improvements to the accuracy of both the 3D scene and the ray tracer. It can also be used to compare which of two different ray tracers produces a more similar result.[1]

[1]We used an implementation of SSIM called *dssim* [Lesiński, 2020] to compute the SSIM between the render and and the photo, and it was 0.728033. This number is meaningless without further context, however.

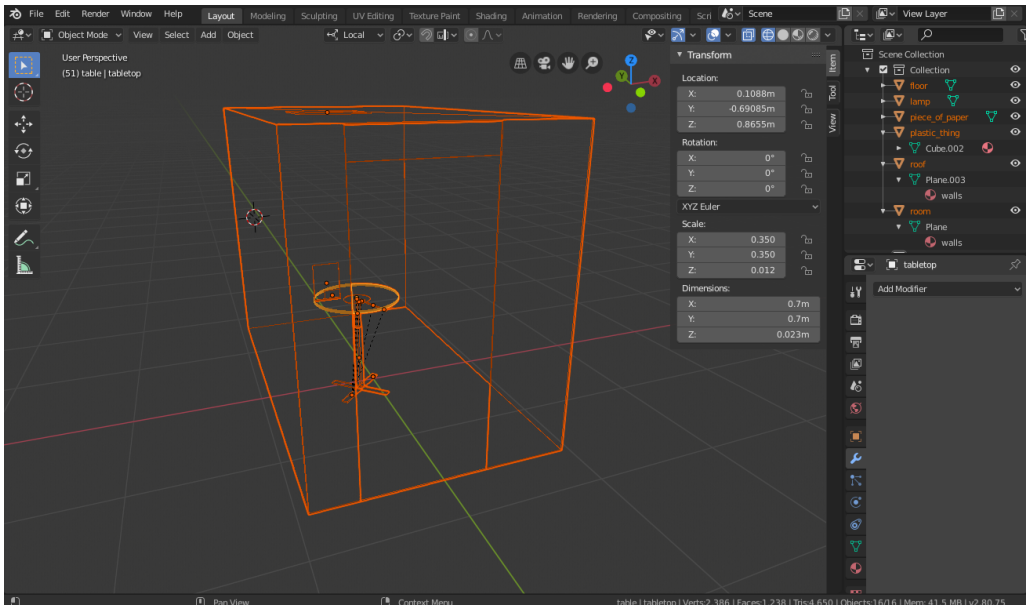


Figure 4.3: 3D modelling in Blender

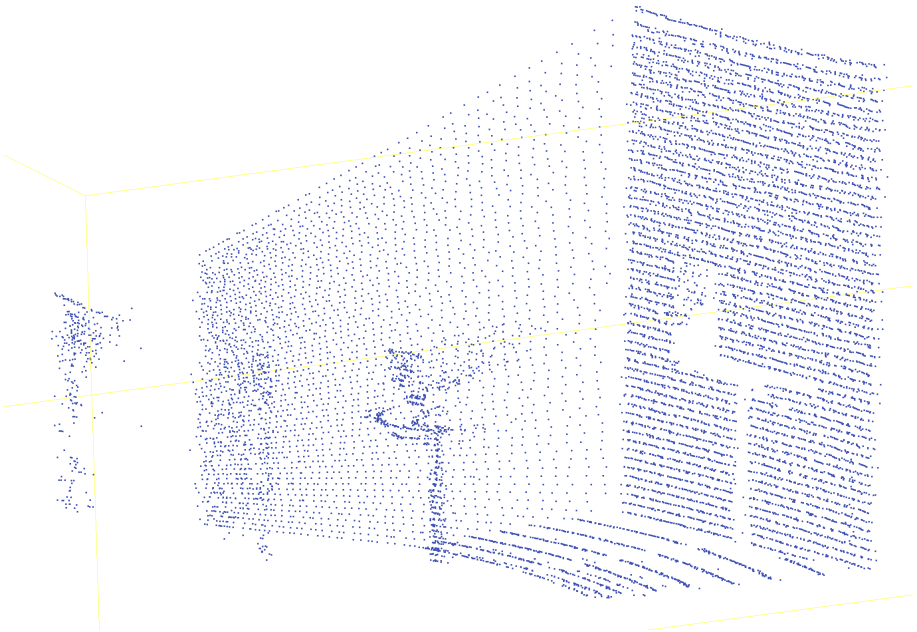


Figure 4.4: Point cloud captured by a real LIDAR sensor and visualised in CloudCompare

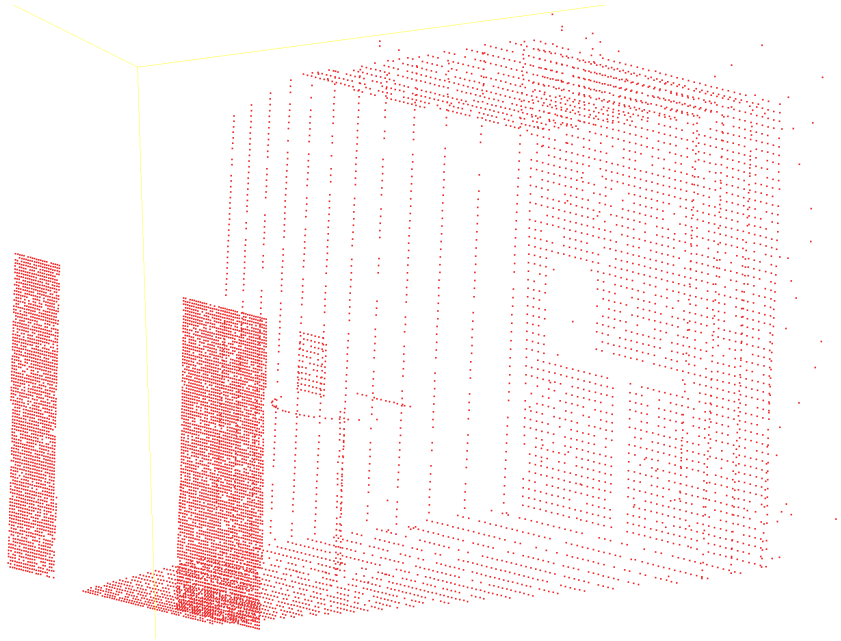


Figure 4.5: Point cloud produced by our ray tracer in a virtual scene and visualised in CloudCompare. Each point is the closest of 6 samples per pixel.

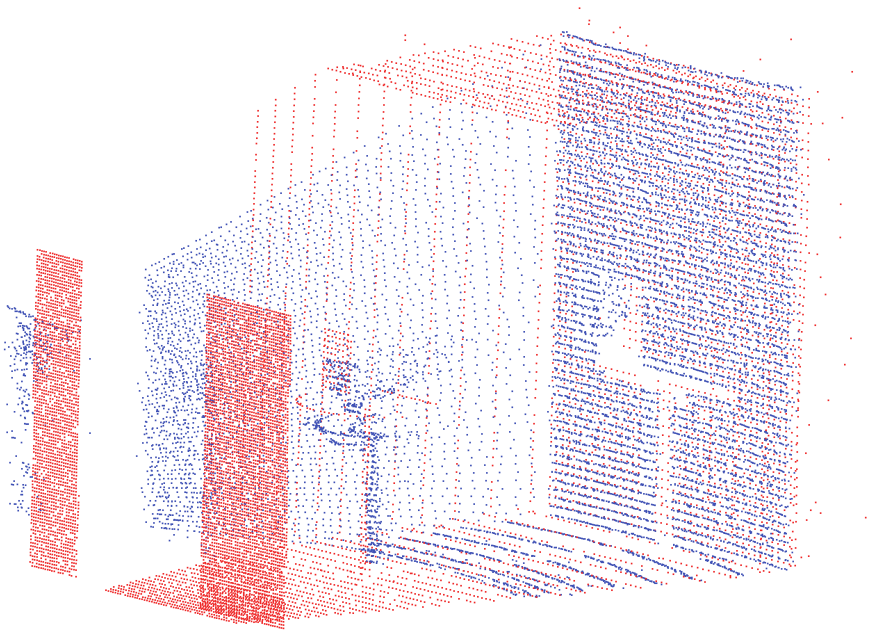


Figure 4.6: The alignment of Figure 4.4 and Figure 4.5 in CloudCompare

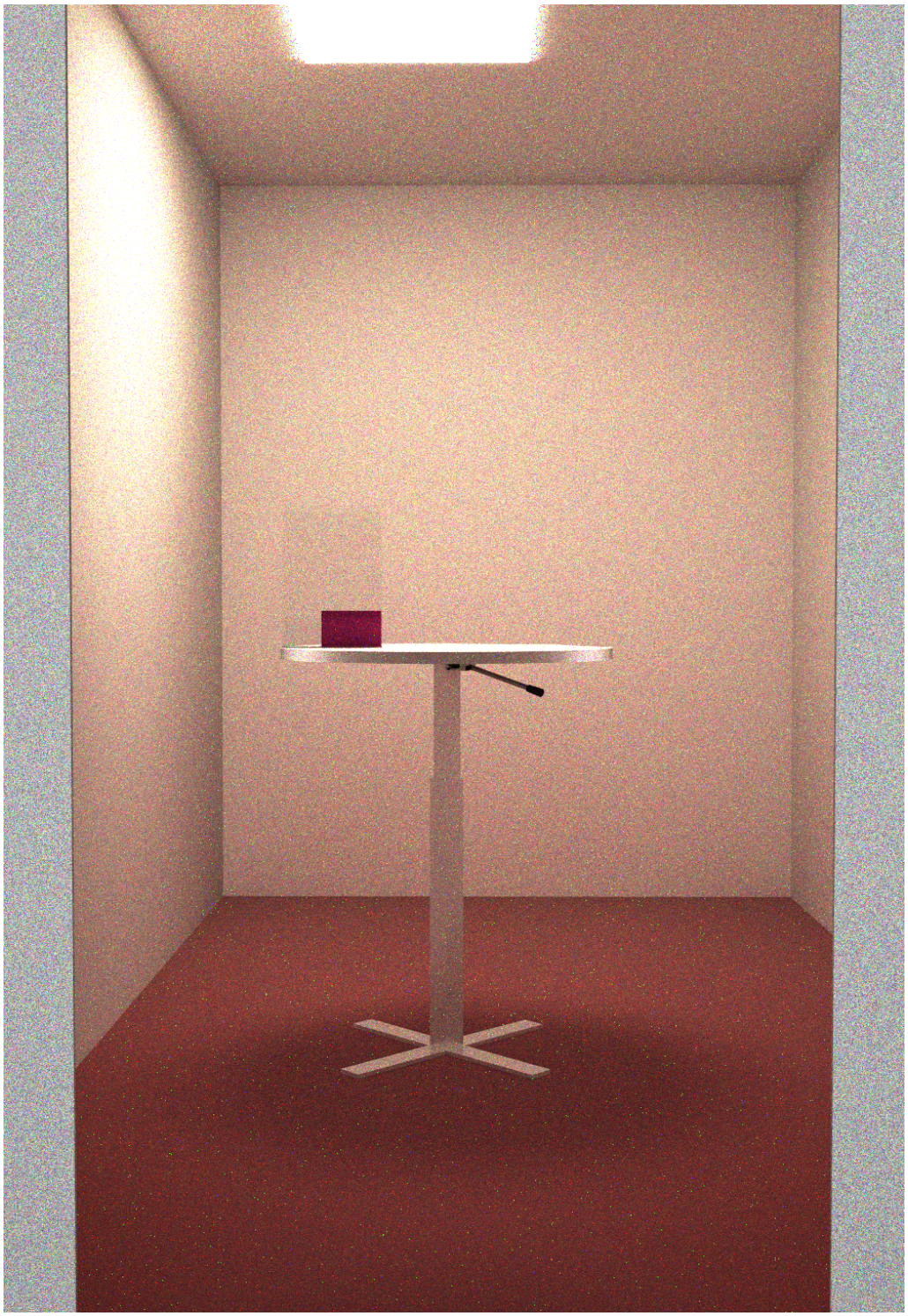


Figure 4.7: Image rendered by our ray tracer in the virtual scene with 1000 samples per pixel.

5
Discussion

In this thesis we have demonstrated the viability of using a high-level functional programming language to implement a non-trivial and physically-based ray tracer, that seems promising for simulating both LIDAR and vision sensors. By only using the simple SOAC functions built into the language, a parallel implementation running on the GPU was achieved.

5.1 Using Futhark

Subjectively, Futhark has been very nice to use in contrast to other programming languages that can target the GPU like CUDA C. One does not have to worry about uninitialised variables, or be aware of exactly how memory is allocated and parallelism introduced. As it's put on the Futhark website, "With Futhark, you don't have to be smart to make your code run fast, you just need to be able to afford an expensive GPU." [Henriksen, 2020].

Futhark is a research language still in development and, as such, stability has not been guaranteed. We even encountered a couple of compiler bugs during the course of this project, but Troels Henriksen – the language's author – was able to identify and fix them very quickly.

We also experienced breaking changes in the type system and syntax with the update to Futhark version 15.1, which introduced sized types. This only posed a minor inconvenience, however, and according to Henriksen the language is close to being completed, so breaking changes are likely to become less frequent.[1]

For this project, the only backends that were used were the OpenCL, CUDA, and C backends. These are not the only backends, however, and several different alternatives are being developed.

5.1.1 Multicore backend

The most interesting alternative backend is perhaps the `multicore` backend, which is currently under development. It is similar to the OpenCL and CUDA backends, but can generate code that will run in parallel on multiple cores. As with the other backends, the key is that the programmer does not need to think much about the parallelism – the compiler should find it on its own. The multicore backend should also be able to extract irregular nested parallelism – something which the GPU backends cannot do. In the context of ray tracing it is unlikely that the performance

[1] Said by Henriksen in a meeting where we spoke about the thesis project.

of CPUs will even be close to that of similarly priced GPUs, but it is could still be useful to compile the exact same code to parallel CPU code in contexts where GPUs are not readily available or difficult to integrate with an existing workflow or hardware configuration. In Section 4.1, we show some initial results of generating code using an early version of the multicore backend. If the performance of the generated code continues to improve in the future, it will likely become a useful alternative backend.

5.1.2 Low-level controls

Futhark is very high-level, and thus does not come with many cranks and levers for controlling more low-level aspects of the generated GPU code. There are some options which can be changed, such as the default size of thread blocks and number of thread blocks. There is also no way to access hardware-specific features such as the NVIDIA RTX ray tracing extensions. This is an inherent limitation of using a language that is designed to be generic – that is, which targets multiple hardware backends.

5.2 Alternatives to Futhark

While we are fairly confident that Futhark was the best choice for our project, we could have used other languages. The strongest candidate that we considered in the early stages of the project was Accelerate, a domain-specific language embedded within Haskell [Chakravarty et al., 2011]. We decided, however, that while Accelerate might be more expressive in a Functional Programming sense, Futhark would likely be both more performant and easier to use.

It is also not entirely clear that Functional Programming was necessarily the best approach for achieving our goals. The C++ code used in [Pharr et al., 2016] is also fairly high-level and easy to read, and achieves multicore parallelism (but not GPU parallelism). It should be noted, however, that it is often much more difficult to write correct code – and especially correct parallel code – in non-functional languages. The clear advantage of Futhark specifically over other languages that we are aware of is that Futhark allows us to generate code for multiple parallel backends, with a very low amount of effort.

5.3 Validation

The aim of the validation that was done for this project has been to provide a *methodology* for validating the functionality of a ray tracer. As such, the validation itself is far from complete. This is mainly the result of prioritisation – performing a more complete validation would likely require as much time as implementing a ray tracer. Instead, we want to briefly show some approaches that can be used to validate the outputs of a ray tracer, and some rudimentary results. The main challenge with validating a ray tracer is that replicating a real scene as a virtual

scene is difficult and time-consuming, and unless the quality of the replicated virtual scene is very high, it is not likely to be very useful for validation purposes.

The results that we obtained do not, unfortunately, say much by themselves. They could still be used to measure improvements to the ray tracer, if virtual scenes were replicated more accurately in the future.

5.3.1 Comparing to other ray tracers

No comparisons were made between our ray tracer and any other existing ray tracer – neither in terms of quality, nor performance. The main reason for why this was outside our scope was that it would require a lot of effort to unify our scene and material description formats with those used by other ray tracers. Also, the fact that we have not measured the actual material properties of the real scene would make it difficult to definitely know whether our ray tracer or the ones tested against are more physically correct.

5.4 Future work

5.4.1 Bidirectional ray tracing method

In this implementation, a relatively simple method of path tracing is used. This method is unidirectional, meaning rays are traced only from the eye into the scene. This works well in some scenes, but in others it results in a lot of variance; for example when the light source is obstructed from contributing any direct light to the scene, but still contributes much via indirect lighting from reflecting off the obstruction. A bidirectional method such as *bidirectional path tracing* [Veach and Guibas, 1995, Chapter 10] or *Metropolis light transport* [Veach and Guibas, 1997, Chapter 11] handles cases like this by tracing not just a ray path from the eye into the scene, but one from the light source as well. By tracing from both directions and then connecting both paths, speed of convergence is often greatly improved, far outweighing the extra cost of tracing an additional path of rays.

Extending our ray tracer with bidirectional path tracing seems like it would be quite straightforward. According to Veach, the algorithm is "relatively simple to implement" [Veach and Guibas, 1995, Chapter 10]. Metropolis light transport, while yielding even better results, seems significantly more difficult to implement. Apparently, "mistakes in any part of the system can cause subtle convergence artifacts that are notoriously difficult to debug" [Pharr et al., 2016, Chapter 16.4].

5.4.2 Importance sampling many lights

Currently when estimating the direct lighting in the path tracing integrator, a single light out of all lights is picked at random, and the result is weighed by the number of light sources (see Section 3.4.1). To reduce variance without impacting performance too much, some technique utilizing acceleration structures and heuristics could be used. Conty Estevez and Kulla present one such technique which uses a BVH which

aggregates energy, spatial, and orientation information from the lights of the scene to enable accurate estimation with good performance [Conty Estevez and Kulla, 2018].

5.4.3 Simulating lenses

In this project, only a rudimentary lens – essentially a pinhole camera – was simulated. The usual approach to simulating lens distortion effects in the industry is to apply distortion effects in a post-processing step. Such an approach is not entirely satisfactory, however, as the transformation is inherently lossy and does not produce fully accurate results. A physically based ray tracer – such as the one presented in this thesis — could potentially produce much more accurate results. But in order to do so, it would be necessary to have a model of the entire lens – which is something that is not usually readily available.

5.4.4 Other sensors

Cameras and LIDAR sensors are not the only sensors that modern vehicles are equipped with. Ultrasound sensors as well as radar sensors are also used in similar ways to the camera and LIDAR. Radar sensors are also used to detect passengers in the car. These sensors operate according to physical principles which are similar to those of the camera and LIDAR – essentially, they are just different ways of using electromagnetic radiation to create a representation of the surroundings. Ray tracing should therefore also be a good fit for simulating these other sensors, and extending our current implementation to support additional sensors is quite feasible – the high-level nature of the Futhark language should hopefully make it fairly painless. The challenges are likely to be more related to understanding what physical properties differ and how.

5.4.5 Scenes and dynamic scenes

Our ray tracer currently only supports the Wavefront .obj format for defining 3D geometry. For more advanced use cases, it would be useful to have a system for combining multiple objects as scenes. This would also be a prerequisite for rendering dynamic scenes within our ray tracer – and that is obviously an important feature in the context of testing virtual cars.

5.4.6 Improving validation

As has already been discussed in Section 5.3, the validation presented in this report has been quite rudimentary. There are many ways in which it could be improved. The most urgent improvement would be to improve the 3D modelling. Taking accurate measurements of the real world objects and scenes to be modelled would also be quite important. In order to properly validate how the ray tracer handles physical properties, it would also be necessary to properly measure the physical properties of the materials used.

Validating using perception algorithms

A more promising approach for validating ray tracers in the context of simulating virtual cars is to use perception algorithms. Such algorithms are already used for safety features in cars, and are the core of what makes autonomous driving possible. Validation of a ray tracer could then be achieved by comparing what objects the perception algorithms detect in a real scene with what they detect in a scene rendered by the ray tracer. That is, one would use the functions that are used to validate actual car functionality to also validate the ray tracer. This approach is likely to be more forgiving of faults in the ray tracer, but has the advantage of being much closer to actual car function validation.

Bibliography

[Akenine-Möller et al., 2018] Akenine-Möller, T., Haines, E., Hoffman, N., Angelo, P., Michal, I., and Sebastien, H. (2018). *Real-time rendering*. AK Peters/CRC Press.

[Beckmann and Spizzichino, 1987] Beckmann, P. and Spizzichino, A. (1987). The scattering of electromagnetic waves from rough surfaces. *Norwood, MA, Artech House, Inc., 1987, 511 p.*

[Belbachir et al., 2012] Belbachir, A., Smal, J.-C., Blosseville, J.-M., and Gruyer, D. (2012). Simulation-driven validation of advanced driving-assistance systems. *Procedia-Social and Behavioral Sciences*, 48:1205–1214.

[Blelloch, 2000] Blelloch, G. (2000). Making parallel programming easy and portable. https://www.cs.cmu.edu/~scandal/nesl/info.html.

[Blelloch, 1990] Blelloch, G. E. (1990). *Vector models for data-parallel computing*, volume 356. MIT press Cambridge.

[Blelloch, 1992] Blelloch, G. E. (1992). *NESL: a nested data parallel language*. Carnegie Mellon University.

[Chakravarty et al., 2011] Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., and Grover, V. (2011). Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM.

[Conty Estevez and Kulla, 2018] Conty Estevez, A. and Kulla, C. (2018). Importance sampling of many lights with adaptive tree splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):1–17.

[Cramer, 1750] Cramer, G. (1750). *Introduction a l'analyse des lignes courbes algebriques*. Chez les freres Cramer & Cl. Philibert.

[Girardeau-Montaut, 2016] Girardeau-Montaut, D. (2016). Cloudcompare. http://pcp2019.ifp.uni-stuttgart.de/presentations/04-CloudCompare_PCP_2019_public.pdf.

[Hecht, 2018] Hecht, J. (2018). Lidar for self-driving cars. *Optics and Photonics News*, 29(1):26–33.

[Henriksen, 2020] Henriksen, T. (2016-2020). Futhark developer blog. https://futhark-lang.org/blog.html.

[Henriksen, 2017] Henriksen, T. (2017). Streaming combinators and extracting flat parallelism. https://futhark-lang.org/blog/2017-06-25-futhark-at-pldi.html.

[Henriksen, 2019] Henriksen, T. (2019). Incremental flattening for nested data parallelism on the GPU. https://futhark-lang.org/blog/2019-02-18-futhark-at-ppopp.html.

[Henriksen et al., 2017] Henriksen, T., Serup, N. G., Elsman, M., Henglein, F., and Oancea, C. E. (2017). Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *ACM SIGPLAN Notices*, volume 52, pages 556–571. ACM.

[Henriksen et al., 2019] Henriksen, T., Thorøe, F., Elsman, M., and Oancea, C. (2019). Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 53–67. ACM.

[Hughes, 1989] Hughes, J. (1989). Why functional programming matters. *The computer journal*, 32(2):98–107.

[Immel et al., 1986] Immel, D. S., Cohen, M. F., and Greenberg, D. P. (1986). A radiosity method for non-diffuse environments. *ACM Siggraph Computer Graphics*, 20(4):133–142.

[Kajiya, 1986] Kajiya, J. T. (1986). The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150. ACM.

[Karras, 2012] Karras, T. (2012). Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37. ACM.

[Kawakami et al., 2009] Kawakami, R., Zhao, H., Tan, R. T., and Ikeuchi, K. (2009). Spectral sensitivity database. https://nae-lab.org/~rei/research/cs/zhao/database.html.

[Kawakami et al., 2013] Kawakami, R., Zhao, H., Tan, R. T., and Ikeuchi, K. (2013). Camera spectral sensitivity and white balance estimation from sky images. *International Journal of Computer Vision*, 105(3):187–204.

[Lauterbach et al., 2009] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. (2009). Fast BVH construction on GPUs. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library.

[Lesiński, 2020] Lesiński, K. (2017-2020). dssim: RGBA Structural (Dis)similarity. https://kornel.ski/dssim.

[Majek and Bedkowski, 2016] Majek, K. and Bedkowski, J. (2016). Range sensors simulation using GPU ray tracing. In *Proceedings of the 9th International Conference on Computer Recognition Systems CORES 2015*, pages 831–840. Springer.

[Parker et al., 2010] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., et al. (2010). Optix: a general purpose ray tracing engine. *ACM transactions on graphics (tog)*, 29(4):1–13.

[Peercy et al., 1995] Peercy, M. S., Zhu, B. M., and Baum, D. R. (1995). Interactive full spectral rendering. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 67–ff. ACM.

[Pharr et al., 2016] Pharr, M., Jakob, W., and Humphreys, G. (2016). *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

[Torrance and Sparrow, 1967] Torrance, K. E. and Sparrow, E. M. (1967). Theory for off-specular reflection from roughened surfaces. *Josa*, 57(9):1105–1114.

[Veach, 1997] Veach, E. (1997). *Robust Monte Carlo methods for light transport simulation*. PhD thesis, Stanford University, Department of Computer Science.

[Veach and Guibas, 1995] Veach, E. and Guibas, L. (1995). Bidirectional estimators for light transport. In *Photorealistic Rendering Techniques*, pages 145–167. Springer.

[Veach and Guibas, 1997] Veach, E. and Guibas, L. J. (1997). Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76. ACM.

[Whitted, 1979] Whitted, T. (1979). An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, page 14. ACM.