MASTER'S THESIS | LUND UNIVERSITY 2018

Filtered Path Tracing using Halide

Fredric Berg

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-42

Filtered Path Tracing using Halide

Fredric Berg
`tfy11fbe@student.lu.se`

December 5, 2018

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Michael Doggett, `Michael.Doggett@cs.lth.se`

Examiner: Flavius Gruian, `flavius.gruian@cs.lth.se`

Abstract

Halide is an embedded language in C++ used for writing high-performance image and array proccessing code. This thesis explores the possibilites and limitations of Halide by using it for an anisotropic reconstruction of images rendered with path tracing. By taking a Monte Carlo rendered image as an input and using the information from the path tracing process and surrounding pixels a new value can be calculated for the current pixel. The algorithm is first implemented in C++ and later using Halide. The results are compared through runtime and similarity to the ground truth. The conclusion is that Halide is not suitable for these types of imaging pipelines. The Halide implementation takes 8% longer to produce the final images.

Keywords: MSc, halide, monte carlo, path tracing, image filtering

Acknowledgements

I’d like to thank my supervisor Michael Doggett.

Contents

Chapter 1
Introduction

1.1 Image processing

Image processing is an important application in todays world. It is used in many different fields such as computer graphics, medicine and computer vision. Most commonly the limiting factor for pushing it even further is time and hardware constraints. That makes it important to have efficient and well optimized algorithms. Optimization is especially necessary in many modern devices, such as cell phones where the hardware is more limited. Two common methods for rendering realistic images are ray tracing and path tracing. A common theme for image rendering, and especially for methods such as ray tracing and path tracing, is that the more time you can spend on rendering an image the better the result will be. By optimizing the rendering pipeline a higher quality can be achieved in a similar time frame. One of the companies that work with this is Disney. They use a renderer called Hyperion, which is a path traced based renderer, for rendering scenes in their animated movies. A single frame of their productions can take 68 minutes to render. [1] Another application is real-time ray tracing. Traditionally ray tracing and path tracing has been pre-rendered, which means that it is not rendered in real time. That is why trailers for games often look a lot better compared to playing the game itself. Modern advancements has made it possible to produce ray traced images in real-time. The newest Battlefield game, Battlefield V, has support for real-time ray tracing. [2] This is currently only available on Nvidias newest RTX 2000 series, which are top of the line graphics cards.

1.2 Halide

Halide is a language that is designed to make it easier to write high-performance image and array processing code. [3] Halide is not a standalone language but is instead embedded in C++. Halide provides tools to make it easier to take advantage of various optimization aspects, such as memory locality and vectorized computations. The primary benefit of Halide is that it separates the implementation of the algorithm from the optimization of the algorithm. This means that you can implement the algorithm in one step, and afterwards modify the way that data is processed without changing how the algorithm functions. To illustrate the differences consider the examples below. Listing 1.1 is an example of how Halide separates the algorithm from the order of computation.

Listing 1.1: Algorithm and schedule

```
// Define algorithm - what is computed
Func a, b;
Var x, y;
b(x, y) = x+y;
a(x, y) = (b(x,y-1) + b(x, y) + b(x, y+1))/3;

// Define schedule - where and when it is computed

b.compute_at(a, x);

// compute_at computes the values of b as soon
// as they are needed for the computation of a
```

The algorithm is separated from deciding how the data is processed. Our schedule will only define the order that data is processed, and will not change the functionality of the algorithm. Listing 1.2 is the same algorithm and order of processing but in C++.

Listing 1.2: C++

```
// Define Algorithm and Schedule
int a[height][width];
for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
          int b[3];
          b[0] = x+y-1;
          b[1] = x+y;
          b[2] = x+y+1;
          a[y][x] = (b[0]+b[1]+b[2])/3;
     }
}
}
```

The modifications to the order in which data is processed is made "inside" the algorithm. This has the drawbacks that the code is harder to read and understand, but making mistakes can also lead to compilation errors or computing the wrong values and change the meaning of the algorithm. Comparing the two Halide clearly states what is being computed and then defines how it is done. In C++ just understanding what is being computed and the order it is done is harder, because it is hidden in the middle of the code even for a simple example. Optimizations are more easily implemented with the Halide framework because of the scheduling aspect. [4] You can try different schedules without making modifications to the actual algorithm. Halide also provides tools that makes it easier to take advantage of parallelism, through things such as Single Instruction Multiple Data (SIMD). Changing the order that data is processed can have a significant benefit to the runtime of an algorithm. As a comparison an optimized version of a 3x3 box filter, where each pixel in the output has a value equal to the average value of the surrounding pixels in the input, can be 11x faster than a naive implementation in C++. [4]

1.3 Purpose

This thesis will explore the Halide language and the opportunities for efficient programming by applying it to an algorithm for de-noising images rendered using a path tracing algorithm. De-noising takes advantage of information generated during the path tracing process and combines it with neighbouring pixels to compute a new value for the current pixel. The focus lies in investigating the Halide language and attempting to speed up the de-noising algorithm in terms of run time by using Halide.

1.4 Research Questions

The report answers the following questions.

- How efficient is Halide for optimizing advanced image filters?
- Are there any challenges using Halide for advanced image filters?

1.5 Contributions

The end result is two implementations, one running in C++ and one running partly in C++ and partly in Halide. The run time for the hybrid version is 8% slower than the C++ version. The main take away of this thesis is the pros and cons of Halide. In order to make it easier to implement high performance algorithms some limitations are also present in the language. These limitations make Halide unsuitable for ray tracing and path tracing based algorithms due to the heavy reliance on structures such as trees and lists, which is a data structure that Halide can not represent. Finding alternatives to these types of data structures is a hard problem to solve. [5]

1.6 Related Work

There are existing research projects for de noising Monte Carlo renderers.

Lehtinen, Aila, Laine, and Durand describes a technique for de noising stochastically rendered images, such as a path traced one. By storing information from the rendering process and using the values of neighbouring pixels to compute a new value for the current pixel the effective sampling rate can be increased. By applying this technique to a sparsely sampled (low quality) input image an output image of higher quality can be generated. [6]

There is no related work to my knowledge of combining this type of image filters with Halide, but it has been used for other applications.

Ragan-Kelley presents an in depth discussion of Halide from one of the designers behind Halide. It discusses the strengths and weaknesses and compares Halide implementations of imaging algorithms such as a blur, bilateral grid and a camera pipeline to hand-tuned C and CUDA implementations. The results shows that the Halide implementation can be up to four times as fast. [5]

Kelly, Adams, Paris, Levoy, and Amarasinghe discusses the strengths of separating the algorithm from the scheduling, as Halide does. A comparison between a clean version of C++, an optimized version of C++ and a Halide version of a blur algorithm is made. While the optimized version of C++ is equally fast, it is far more complex and the readability is much worse. The clean version of C++ is ten times slower than the Halide implementation. [4]

Li, Gharbi, Adams, Durand, and Ragan-Kelley uses Halide for gradient-based optimization. Gradient-based optimization is a common technique in image processing and has been used for applications such as image restoration. By extending the Halide language a user can automatically derive and optimize gradient-based code for an image processing pipeline. The halide implementation of the forward and gradient computations of the bilateral slicing layer [8] is compared to implementations in PyTorch and CUDA. The results show that the Halide implementation is twenty and six times faster respectively. [7]

Mullapudi, Adams, Sharlet, Ragan-Kelley, and Fatahalian discusses the auto-scheduler. Halide provides the tools for optimizing the algorithms execution, but it is still a hard task defining a good schedule. The auto-scheduler can be used to automatically optimize the execution of an algorithm. It is applied to a variety of applications and is competitive with manual schedules defined by experts in certain cases. It is still a work in progress and can not handle advanced reasoning about schedules. [9]

Chapter 2

Background

2.1 Ray tracing

Ray tracing is a method used in order to simulate how lighting works in the real world. It tries to recreate how light from a light source such as the sun bounces around and eventually makes it to the retina which is what we perceive as vision. A ray tracing algorithm sends rays out of a virtual camera into the scene. Whenever it intersects with an object it accounts for different properties of that object. If it is a reflective surface the ray is bounced at an angle and if it is a transparent surface a part of the light travels through the object to simulate refraction. At every intersection point a new ray, in addition to the reflected/-transmitted rays, is traced towards each light source. If the ray can not find a way to a light source without intersecting with an object, it is discarded. This type of ray tracing is referred to as Whitted style ray tracing, after the creator Turner Whitted [10].

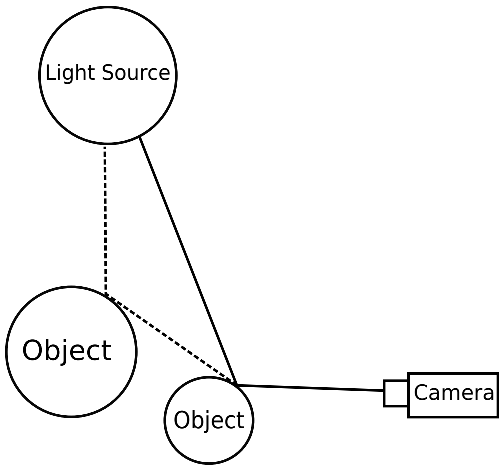


Figure 2.1: Whitted style ray tracing

Figure 2.1 depicts a ray cast from the camera. When the ray intersects with an object the ray is split. One is traced towards the light and another is reflected and travels further on in the scene.

2.2 Path tracing

Path tracing is a method that is based on ray tracing. In a similar fashion rays are sent from a virtual camera. Whenever the ray intersects with an object a random number is used to decide whether the ray should be traced back to the light source or cast in a random direction. By repeating this process for every intersection a path is generated for the ray, which is why this method is called path tracing.

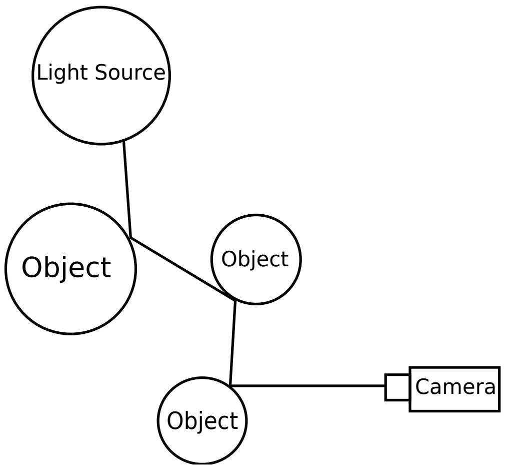


Figure 2.2: Path tracing

Figure 2.2 depicts a ray being cast from a camera. The ray is bounced around the scene in random directions generating a ray path. Everything along this path will contribute to the final pixel value. Because the ray is not directly traced back to the light source, as in ray tracing, this method is stronger at rendering realistic shadows. The ray path can include parts of the scene that are in shadow. Due to the reliance on randomizers this method is prone to noise. In order to get an image that accurately reflects the scene a large number of ray paths must be traced, typically in the thousands per pixel in the final image. The number of ray paths per pixel is denoted by samples per pixel (spp). This algorithm was presented by Kajiya [11], along with a model for computing the pixel values.

$$L(\mathbf{p} \rightarrow \mathbf{c}) = \frac{1}{\pi} \int L_{in}(p \leftarrow \omega) f_r(\mathbf{p}, w \rightarrow w_c) cos\theta d\omega \tag{2.1}$$

where

- $\mathbf{p}$ = Hit point of a ray
- $\mathbf{c}$ = The camera
- L_{in} = The light arriving from the given direction
- L = The light for the current point
- ω = The direction of incoming light
- f_r = Reflectance function
- ω_c = Direction to camera
- θ = the angle between the incoming light and the surface normal

Figure 2.3 and 2.4 are examples of how the quality of a path traced image might wary depending on the number of ray paths computed. The images were rendered using Physically Based Rendering Tool (PBRT). [12] The scene depicts a model of a monkey, called Suzanne, with differing amounts of gloss. It is a test model available in blender.

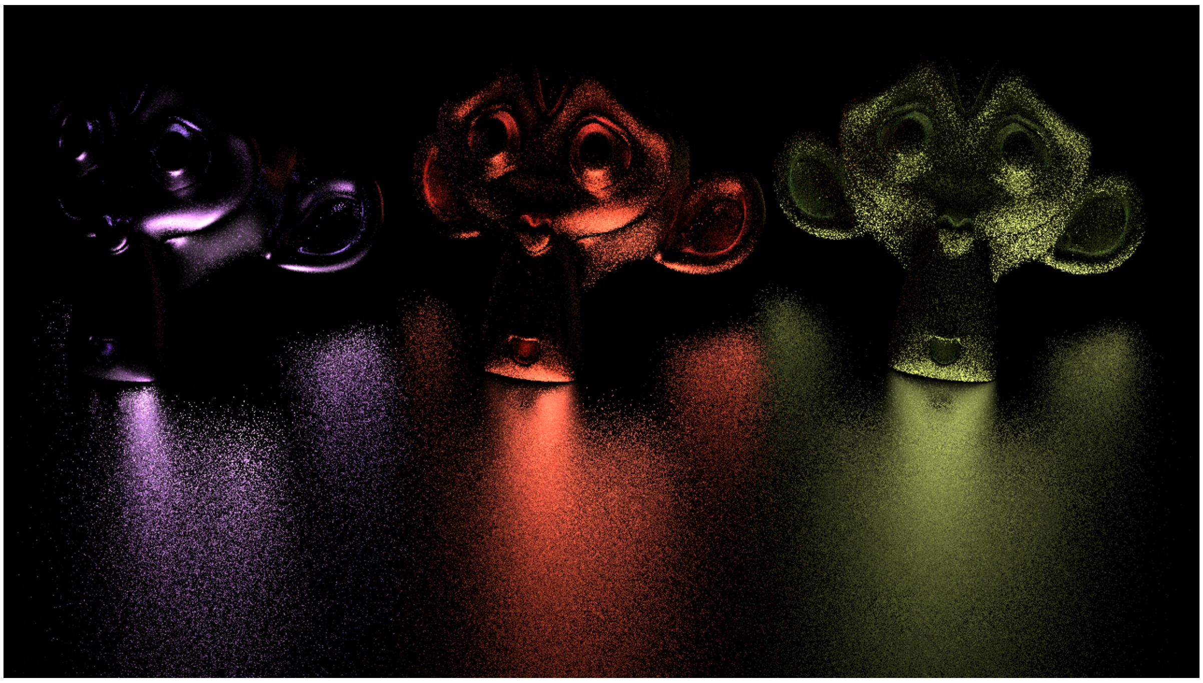


Figure 2.3: Path traced image with 8spp

Figure 2.3 depicts a path traced image with 8spp. Because it was rendered with ”too few” ray paths there are a lot of visual artifacts.

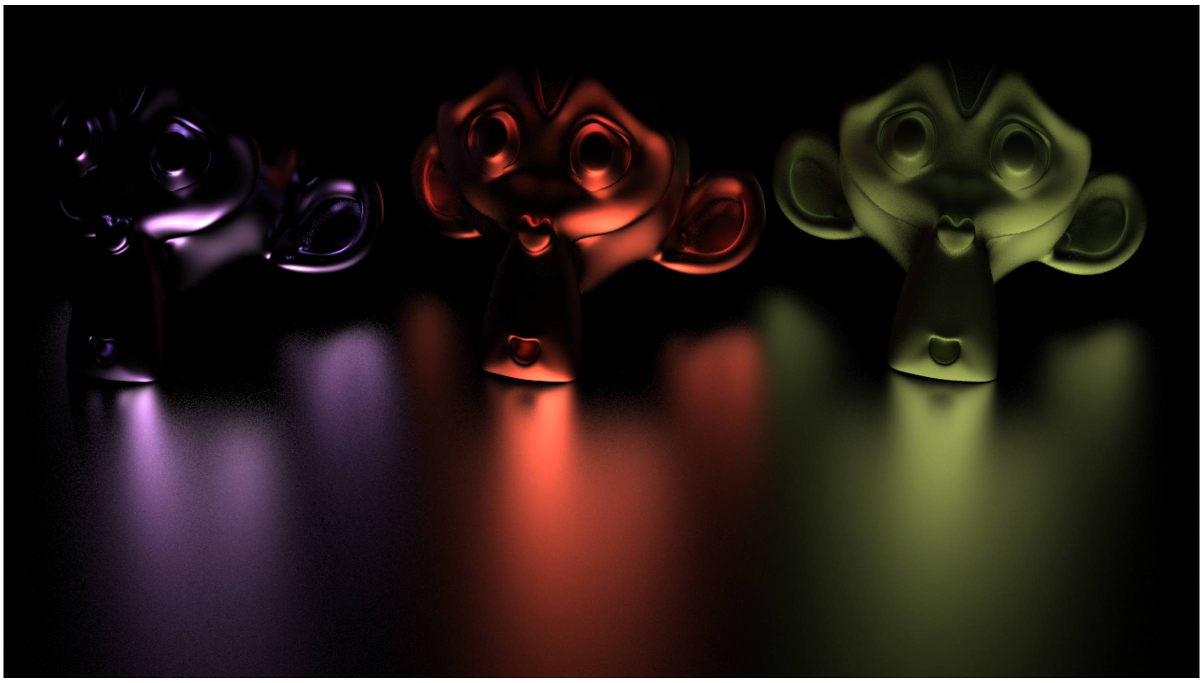


Figure 2.4: Path traced image with 512spp

Figure 2.4 depicts a path traced image with 512spp. There are still some visual artifacts, most easily seen in the ground in front of the purple monkey. Overall though, the image quality is very high and gives an accurate representation of shadows, caustics and indirect lightning.

2.2.1 K-nearest neighbours

K-nearest neighbours (kNN) is an algorithm used for classifying data. The type of an object is decided by having the k closest neighbours vote. Figure 2.5 depicts an object, marked with a question mark, about to be classified. If k = 3, the it would be classified as a square, because the three closest neighbours are two squares and one circle. If k = 7 it would be classified as a circle.

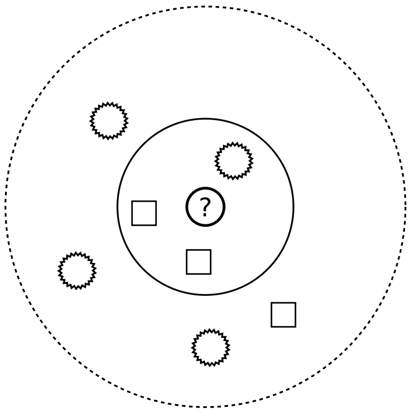


Figure 2.5: Visualization of a kNN algorithm

2.3 De-noising Algorithm

The following section describes an algorithm for de-noising a path traced image with a low number of spp, such as Figure 2.3, with the goal of reaching a similar quality of a path traced image with a high amount of spp, such as Figure 2.4, but with a shorter run time.

2.3.1 Overview

The de-noising algorithm is divided into two stages. In the first stage a path tracer is used to render a low spp image. During this process information, presented in section 2.2, is stored. This is the input data. The intent is to reuse the data to improve the effective sampling rate. This is done by calculating a spatial radius of influence for every sample, which is a measurement of how far that sample influences its surroundings. This is described in section 2.3.2.

In the second pass the image is reconstructed. Using the information stored in the first pass reconstruction rays, described in section 2.3.3, can be sent out in similar fashion to a path tracer, except this time the spatial radius of influence is used to upsample the image. This means that information from surrounding samples contributes to the final pixel values. By reconstructing the image the noise levels can be reduced. The output is the r,g,b values for the upsampled image, which is significantly less noisy.

2.3.2 First pass

Path tracing

In order to reconstruct the image we need additional information from the path tracer in order to accomplish this. The path tracer used for this is pbrt. Since it is normally used simply for path tracing and the additional information is not needed, a parser is used to extract and present the information in a useful way. Below are some the values which are stored from the path tracing process.

- x, y values of the samples pixel coordinates
- r, g, b values for the samples radiance
- 3d camera-space normal for the primary hit
- albedo value for primary hit
- origin of the secondary ray
- hit point of the secondary ray
- normal for the secondary hit

Because the image is 8 SPP there will be 8 sets of data for every data point above. An assumption here is that the input data accurately samples the scene. Due to the random nature of a path tracer the ray paths may not fully describe the indirect lightning for the sample. Information that is not present in the input can not be recovered in the output. Before the reconstruction can take place information needs to be extrapolated from the input data. The following sections will describe how the samples are processed.

Computing densities

The de-noising algorithm assumes that the value of a sample in general is related to the surrounding samples. By treating the input samples as circular disks (splats) the size of a splat can be determined by analyzing the surroundings. If there are many nearby samples the splat is enlarged. Some care must be taken however, because a spatially close sample does not neccesarily belong to the same splat. Figure 2.6 depicts four samples lying in two surfaces. Because the samples are on different surfaces, we do not want to group them together.

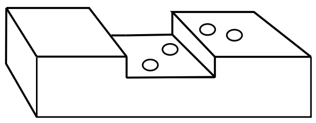


Figure 2.6: Samples in different surfaces

In order to prevent this from happening a modified version of the a kNN algorithm, presented in section 2.2.1, is used. When searching for nearby samples movement in the direction of the normal is discouraged using the following penalty function [13].

$$\|\mathbf{p} - \mathbf{q} + 2((\mathbf{p} - \mathbf{q}) \cdot \mathbf{n})\mathbf{n}\| \tag{2.2}$$

where **p** and **q** are the locations of the samples and **n** is the normal. The vertical bars means that we are taking the euclidean length of the expression. This encourages the kNN algorithm to prioritize spatially close samples in locally flat regions. Figure 2.7 depicts the intended outcome of the classification of the samples from Figure 2.6. The samples are grouped into two hit splats, one lying in each surface.

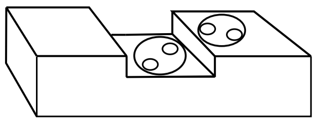


Figure 2.7: Intended outcome of sample grouping

Shrinking the hit splats

Even though measures were taken to prevent the kNN algorithm from enlarging hit splats it should not, there are still some remaining artifacts. Imagine a small feature of the scene, such as a leaf. The leaf will not have as many samples as a larger geometry in the path traced image, but they might be close to other samples. This means that some of the hit splats for these features will be enlarged by the previous step. During the path tracing stage information was stored of a rays starting and ending point. By retracing these rays there might now be a collision that was not there previously. Figure 2.8 depicts how this might be a problem. Because the leaf is close to the wall the hit splat might erroneously be enlarged. A ray that was supposed to hit the wall now intersects with the hit splat of the leaf.

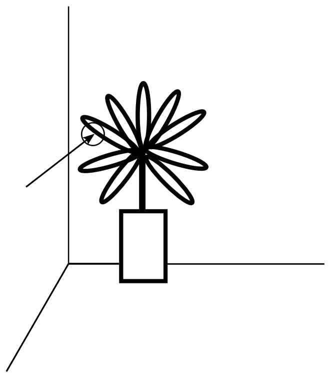


Figure 2.8: Flower in scene with too large hitsplat

Figure 2.9 depicts how shrinking the hit splats can solve this problem. A ray is traveling from point A to B. After the hit splats have been determined the ray no longer reaches point B, because it is blocked by the splat. It is therefore shrunk in order to be consistent with the original input.

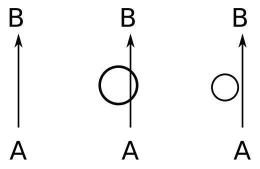


Figure 2.9: Reconstruction rays

2.3.3 The second pass

During the second pass the radiance for the samples is reconstructed.

The Shading Equation

In section 2.2 the shading equation, 2.1, was presented. It was used by the path tracer to compute and store values of L_{in}, which represents the radiance for the samples. In this stage the following equation will be used [6].

$$L_{out}(\mathbf{p} \rightarrow \mathbf{c}) \approx \frac{1}{\pi}\frac{1}{N}\sum_{j=1}^{N} Reconstruct(\mathbf{p}, \omega_j) f_r(\mathbf{p}, \omega_j \rightarrow) \omega_c cos\theta \tag{2.3}$$

The reconstruct function is calculated by sending out reconstruction rays which is a way of upsamling the incident light field. In section 2.3.2 samples were grouped into splats, which represents a surface. Figure 2.10 depicts this process. The reconstruction rays hits a sample that lies within a hit splat. By taking a weighted average of every sample that lies in that splat the value for the current sample can be computed.

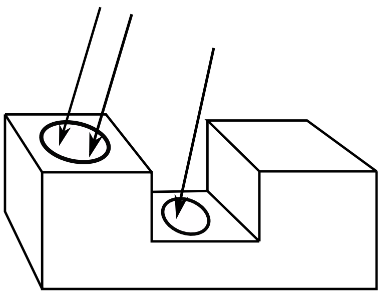


Figure 2.10: Reconstruction ray

Octree

In this section a data structure called an octree will be presented. Because there is a large amount of samples distributed across the scene there needs to be an efficient way of doing different operations. The problem will first be outlined and shown in a binary tree (2D). The differences to an octree are rather straightforward. As an example of why this structure is useful, consider the process of finding the closest samples to our current sample. A naive way of doing it could be to simply check the distance between the current sample and every other sample. Figure 2.11 depicts a scene with samples in it. It is an illustration of a naive way of computing the distance between sample one and every other sample.

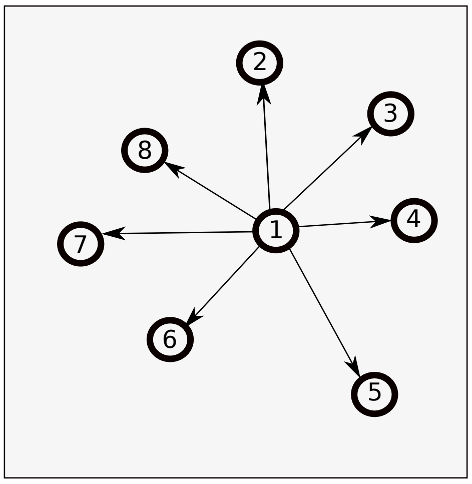


Figure 2.11: Scene view of distances to current sample

Figure 2.12 is a list view of Figure 2.11

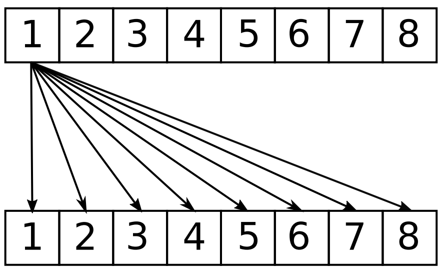


Figure 2.12: List view of distances to current sample

In a small scale scenario a solution such as this might not be such a big problem, but reconstruction of a 400x400 image with 8 SPP has 1280000 samples. Finding the closest samples for every sample using the naive solution is simply going to take too long. The purpose of the tree is to divide the space into different sections. Figure 2.13 depicts how the scene can be divided into segments in order to speed up the computations. Figure 2.14 depicts a list view of the partition.

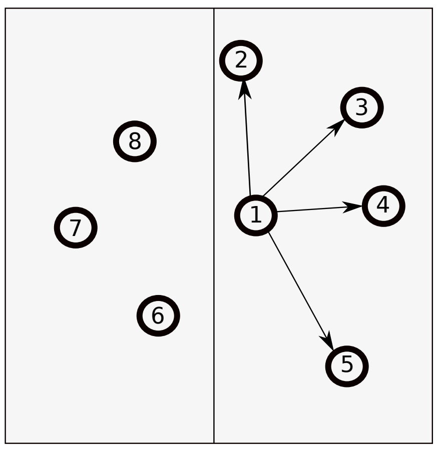


Figure 2.13: First partition

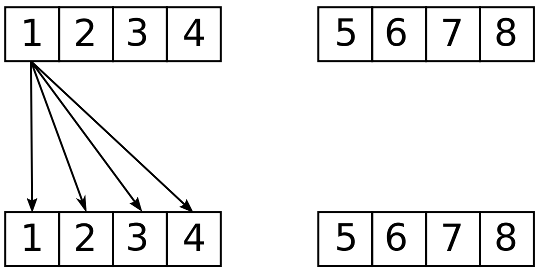


Figure 2.14: List view of first partition

After the space has been divided into two, the solution is to only compare the sample to other samples in the same space as itself. Every time the space is divided the amount of comparisons that has to be made is greatly reduced. Figure 2.15 depicts how the scene might look like when the space has been partitioned several times.

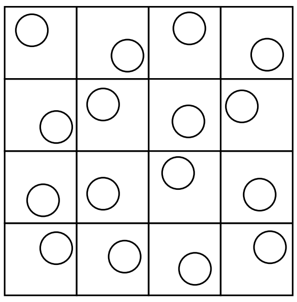


Figure 2.15: Partitioned space

Similar reasoning and methods can be applied to an octree, but for partitioning a 3D space. [14] This is done by recursively dividing the scene into segments of eight. Figure 2.16 depicts a similar partitioning, but this time in a 3D space.

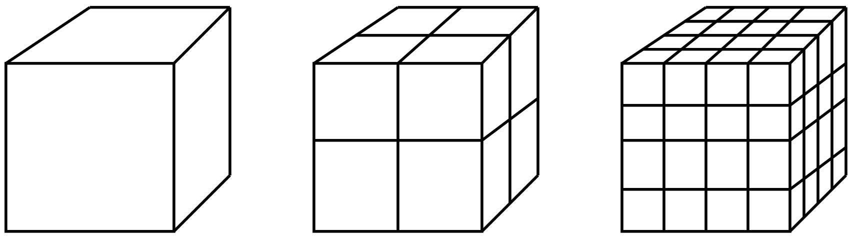


Figure 2.16: Illustration of an octree recursively dividing into octants, starting with the root node

The octree is used throughout the algorithm, but primarily for the computing densities stage, described in section 2.3.2 and the shrink stage, described in section 2.3.2.

2.4 Halide

This section will discuss optimization aspects that Halide takes advantage of as well as a introduction to the language.

2.4.1 Halide Language

In order to better understand the Halide code, the primary primitives and functions will be introduced.

Func Funcs are objects that represents a pipeline stage. It defines what value each pixel should have.

Expr Expressions are used to define a func at any integer coordinate.

Var Vars are used as variables, but they have no meaning by themselves. They are used in a more symbolic manner in the definition of an expression.

Realization Since Halide is just-in-time compiled (JIT) nothing happens until you "realize" it. This also means that you must specify the domain which you want to evaluate your func over.

Example Listing 2.1 defines a function, vars and an expression. The func is then defined by the expression and when it is realized it JIT compiles and evaluates the function over the entire domain. In this case our buffer would be represent an image where every pixel is the sum of the x and y coordinate.

Listing 2.1: Halide example

```
// Create undefined function
Halide::Func f;

// Create variables. They are used symbolically
// x = 0 f.ex. is undefined
Halide::Var x,y;

// Create an expression
Halide::Expr e = x + y;

// Define function at every x and y
f(x,y) = e;
// The size of f is not yet determined,
// so f(5,3) f.ex. is not valid

// The function needs to be realized over a domain
// Evaluate f over the specified domain (800,600)
// and store result in buffer
Halide::Buffer<int32_t> output = f.realize(800,600);
// At every (x,y) f is evaluated using
// the function definition, e = x+y
// f is realized and stored in the buffer
// as an 800x600 image where every (x,y) = x+y
```

2.4.2 Scheduling

One of the reasons for using Halide is the scheduling aspect. The Halide framework is built with this in mind. Scheduling means that you can change the order in which data is processed after the algorithm has been defined. The paragraphs below showcases some of the ways that the order of processing can be rearranged in Halide.

Split Halide functionality for splitting a loop into two nested loops. Listing 2.2 shows how the order of processing data can be changed using the split command.

Listing 2.2: Splitting in Halide

```
Halide::Func f;
Halide::Var x, y, x_outer, x_inner;

// Define Algorithm
f(x, y) = x + y

// Define schedule
f.split(x, x_outer, x_inner, 2);
// Divide x into an inner and outer part.
// The 2 specifies that it should be divided into two parts.

// When f is realized (evaluated)
```

```
    // the splitting is taken into account.
    Buffer<int> output = f.realize(4, 4);
```

Listing 2.3 is the C++ equivalent of listing 2.2.

Listing 2.3: Splitting in C++

```
int g[4][4];
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 2; x_outer++) {
        for (int x_inner = 0; x_inner < 2; x_inner++) {
            int x = x_outer * 2 + x_inner;
                g[x][y] = x+y;
        }
    }
}
```

Reorder Can be used to reorder the way data is processed. You could use it to do a column major traversal instead of a row-major traversal. Listing 2.4 showcases how you can change which variable is processed first. By reordering in this manner we will traverse the data by column-major.

Listing 2.4: Reordering in Halide

```
Func f;
Halide:Var x,y;
f(x, y) = x + y;

// When f is evaluated, loop over y variable first
f.reorder(y,x);

// Evaluate f
Buffer<int> output = f.realize(4,4);
```

Vectorize Vectorize is a functionality designed to split the input data into vectors. It can be used to process data in vectors of 3 f.ex. Listing 2.5 defines a 6x4 image and processes it by vectors of three.

Listing 2.5: Vectorizing in Halide

```
    Halide::Var x, y;
    Halide::Func grid;
    grid(x,y) = x+y;
    grid.vectorize(x, 3)
    grid.realize(6,4);
```

Figure 2.17 shows how the data is processed, in blocks of three.

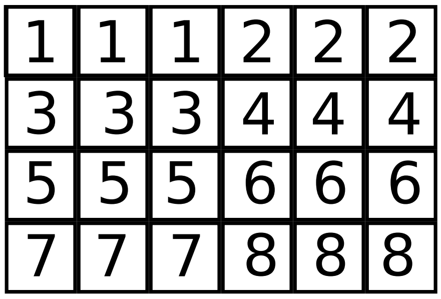


Figure 2.17: Vectorized data

Equivalent C++ code for processing data in the same way would be as follows

Listing 2.6: Vectorizing in C++

```
int grid[6][4];
for (int y = 0; y < 4; y++) {
        for (int x = 0; x < 2; x++) {
        grid[x*3][y] = x*3+y;
        grid[x*3+1][y] = x*3+1+y;
        grid[x*3+2][y] = x*3+2+y;
    }
}
```

Fuse Fusing is when you merge two loops into a single loop. If you have a loop over x and y you can fuse it to loop over a single variable. Listing 2.7 shows how you can fuse variables.

Listing 2.7: Fusing variables in Halide

```
Halide::Func f;
Halide::Var fused, x, y;
f(x, y) = x + y;

// Merge x and y into one variable
f.fuse(x, y, fused);

Buffer<int> output = gradient.realize(4, 4);
```

The C++ equivalent is shown in listing 2.8

Listing 2.8: Fusing variables in C++

```
int f[4][4]
for (int fused = 0; fused < 4*4; fused++) {
        int y = fused / 4;
        int x = fused % 4;
        f[x][y] = x+y;
}
```

Tile Tile can be used to divide a dataset into blocks of various sizes. This function is a combination of using split and reorder together. Listing 2.12 shows example code for how a dataset can be divided into tiles.

Parallel Used for computing in parallel using different threads.

Listing 2.9: Computing in parallel

```
Halide::Func f;
Halide::Var x, y;
f(x, y) = x + y;

// Compute values in parallel over the y coordinate.
// Each row will be calculated in parallel.
f.parallell(y);

Buffer<int> output = gradient.realize(4, 4);
```

Examples As an example consider listing 2.10, which is C++ code for setting the value to x+y for every coordinate in a grid.

Listing 2.10: Grid in C++

```
Grid g[8][8];
for(int x; x < 8; x++) {
    for(int y; y < 8; y++) {
        g[x][y] = x+y;
    }
}
```

This could be implemented in Halide as in listing 2.11 and there would be no difference in processing between the two.

Listing 2.11: Grid in Halide

```
Halide::Var x,y;
Halide::Func grid;
Halide::Expr e = x+y;
grid(x, y) = e;
Buffer<int> output = grid.realize(8, 8);
```

In halide changing the way the data is processed is easier. Listing 2.12 shows how split and reorder can be used to tile the dataset.

Listing 2.12: Tiling

```
Halide::Var x,y;
Halide::Func grid;
Halide::Expr e = x+y;
```

```
grid(x, y) = x + y;

// Create variables for splitting
Halide::Var x_outer, x_inner, y_outer, y_inner;

grid.split(x, x_outer, x_inner, 4);
grid.split(y, y_outer, y_inner, 4);
// Split the x and y variable into an outer and an inner
   part.
// The 4 is called the factor
// It determines how many iterations the inner dimension
   has
// The x_outer and y_outer variables iterate over the tiles
// x_inner and y_inner iterates over the points in each
   tile
// By splitting x and y by 4 the tiles will have a size of
   4x4


grid.reorder(x_inner, y_inner, x_outer, y_outer);
// Specifies the ordering. The dataset will be traversed in
    row-major
// order for each tile. By switching x_inner and
// y_inner we could traverse it in column-major.

// Tile is a shorthand for defining a split
// and reordering in this manner
// We could replace the calls to split and reorder
// with the following code
// and have the same result
// f.tile(x, y, x_outer, y_outer, x_inner, y_inner, 4, 4);

Buffer<int> output = grid.realize(8, 8);
```

Here the loop is split into 4 parts and reordered into what is called a tiled traversal, or tiling. In c++ this would require a quad nested loop. Figure 2.18 shows how the data set is processed for C++ and Halide respectively.

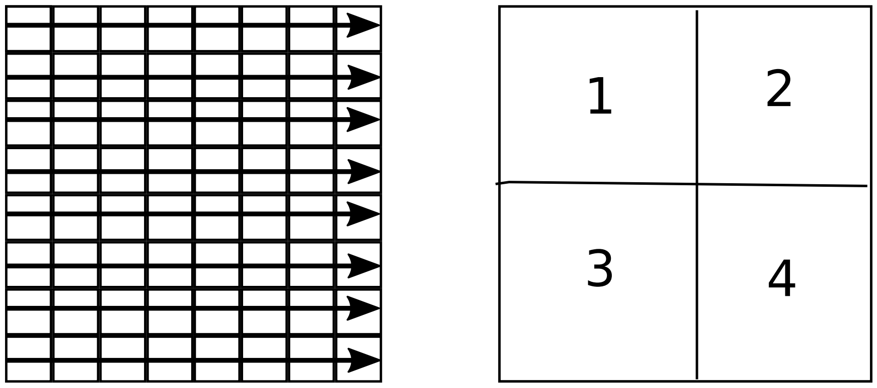


Figure 2.18: C++ and reordered halide

Listing 2.13 C++ code for tiling the data set into tiles of four. It requires a quad nested for loop.

Listing 2.13: Tiling

```
for (int y_outer = 0; y_outer < 2; y_outer++) {
   for (int x_outer = 0; x_outer < 2; x_outer++) {
      for (int y_inner = 0; y_inner < 4; y_inner++) {
          for (int x_inner = 0; x_inner < 4; x_inner++) {
             int x = x_outer * 4 + x_inner;
             int y = y_outer * 4 + y_inner;
          }
      }
   }
}
```

It is a lot more complex defining these kind of schedules in C++. Listing 2.14 shows how tile, fuse and parallel can be combined to create a schedule.

Listing 2.14: Combining tile, fuse and parallel

```
Halide::Func grid;
Halide::Var x, y, xo, yo, xi, yi, fuse;
Halide::Expr e = x+y;
grid = x+y;

grid
    .tile(x, y, xo, yo, xo, yo, 8, 8)
    .fuse(xo,yo, fuse)
    .parallel(fuse);
// Tile the dataset into 8x8 tiles
// Fuse the outer variables (tiles indices) into a single
// variable
// Compute in parallel over the tile indices
```

```
grid.realize(16,16);
```

This schedule would tile the dataset into blocks of eight by eight. Each tile is computed in row-major order. The parallelization means that each tile is computed simultaneously.

2.5 Optimization

This section gives some background as to how Halide can be used to optimize code.

2.5.1 Just-in-time compilation

Just-in-time compilation (JIT), is a way of compiling that Halide uses. Rather than compiling the code prior to running it, it compiles during the execution of the program. The benefit of this is that the code can be analyzed during runtime. This can be used to identify when a speedup can be gained from recompiling it. In Halides case this is useful because it can rearrange executed code to optimize the usage of the cache memory. The downside is that this type of compiling comes with overhead. If time is invested on compilation but it is only used a few times the overhead of compiling it at runtime might outweigh the benefits.

2.5.2 Single instruction, multiple data

Single instruction, multiple data (SIMD) is a way of using processing elements to perform similar operations on multiple data points at the same time. This is often useful in imaging pipelines because modifications are often made in a uniform manner over a data set. One example of this could be when adjusting the brightness of an image. Instead of computing one pixel at a time multiple pixels can be computed simultaneously. Halide provides the tools to make it easy to take advantage of parallelism, such as the parallel function from section 2.4.2.

2.5.3 Memory

There are different types of memory in a CPU. It is divided into a hierarchy which means that data accesses take varying times. The further down you go in this hierarchy the longer it will take in order to access the data. The list below describes the memories and their access speeds. The values correspond to an Intel Xeon E5. [15]

- Internal register - 0.4 ns
- L1 Cache - 0.9 ns
- L2 Cache - 2.8 ns
- L3 Cache - 28 ns
- RAM - 100 ns

- Disk (SSD) - 50-150 μs

The register is at the top of the hierarchy, but it also has the smallest amount of storage. The L1 cache is faster than L2 cache, but has smaller storage the same follows for the other types. Since the access times vary depending on which type of memory data is stored in it is important to make good use of it. If we can make greater use of the cache, meaning that data that is read into the cache is used, the run time of an algorithm can be decreased significantly. The scheduling functions presented in 2.4.2 are helpful tools in order to reduce organize the processing of data in order to reduce the number of cache misses, where data that is needed is not in the cache. Section 5.3 has an example of how this can be used.

Chapter 3
Approach

3.1 Methodology

The process during this thesis was divided into different steps. The first stage was to gain an understanding of a de-noising algorithm aswell as an introduction to Halide. After that a C++ version was implemented, partly to get familiar with the algorithm in a language I am more comfortable with, and partly to have something to compare the Halide implementation to. Lastly an implementation was made using Halide and the result was evaluated by comparing run times and image quality.

Understanding In the initial phase the objective was to gain an understanding of the de-noising algorithm, as well as an introduction to the halide language. This was achieved by studying the de-noising of Monte Carlo renderers, such as [6] and [16]. In order to get introduced to Halide I used their tutorial [17].

C++ To further the understanding of the de-noising algorithm and also to have something to compare the Halide implementation to I implemented it in C++. C++ was selected because Halide is embedded in this language. By comparing the implementations a realistic measure of how much could be gained from switching to Halide can be achieved.

Halide Originally the plan was to have the code running entirely in Halide. Due to limitations with the Halide language, further outlined in section 5.2, this was not the case. Instead a hybrid version was implemented where certain parts are running in C++ and other parts in Halide. The goal of this phase was to achieve a faster run time than the strict C++ implementation.

Evaluation In order to compare the output image quality a ground truth image is needed. A ground truth is used to measure the accuracy of an algorithm. In an object classification algorithm, where the computer labels objects in an image, a ground truth could be a human labelling everything in the image. If the algorithm deviates from the humans labels it is considered as an error. The term is a bit confusing for computer rendered images, since even our ground truth image has noise or errors in it. Regardless a way of measuring the image quality is needed. In this project the ground truth image is an image rendered using path tracing with 2048 spp, depicted in Figure 4.2. This image was rendered using PBRT, which is an open source software used for rendering high quality images. [12] A scene can be created through tools such as Blender or Maya and ran through PBRT to render the images. Originally I planned to do a comparison of the image quality using two methods called peak signal-to-noise ratio (PSNR) and structural similartity (SSIM). Due to compatibility issues explained in section 4.1 the ground truth has different lighting than the result, which made those comparisons meaningless. The run times of the Halide and C++ de-noising implementations will be compared to the path tracer. The test platform was a Intel Core i5-4670k @ 3.40GHz NVIDIA GeForce GTX 1070.

Chapter 4
Results

In this chapter the results will be presented. First the input and output images are presented, along with the ground truth image. The ground truth is then compared to the output images and the run times are presented.

4.1 Resulting Images

Figure 4.1 depicts the rgb values of the input. This is the starting point before the de-noising algorithm is applied. It was path traced using PBRT and with 8 SPP. It contains visual artifacts and is noticeably noisy.

Figure 4.2 is an image of the result after the de-noising algorithm. The quality is much higher. The noise is severely reduced and there are not many, if any, visual artifacts remaining.

Figure 4.3 depicts the ground truth image. It is an image that was rendered using PBRT with an SPP of 2048. This image has different lightning than the result in Figure 4.2. The differences are not due to the algorithm, but because it is of a different scene with different lightning. In order to extract and sort the output from the path tracer a parser was used, mentioned in 2.2. Due to compatibility issues it could not be used to generate a sample buffer from any PBRT file. Therefore I had to use a sample buffer that was already parsed. The rgb values of this buffer is depiced in Figure 4.2. This is an older version of the San Miguel scene which has one type of lightning. The path traced image rendered with pbrt, Figure 4.2, is the same scene, but an updated version with a different type of lightning. I could not find a pbrt file of the older version and therefore this was the best image to compare it with. That is also the reason that the de-noising algorithm is only applied to one image.

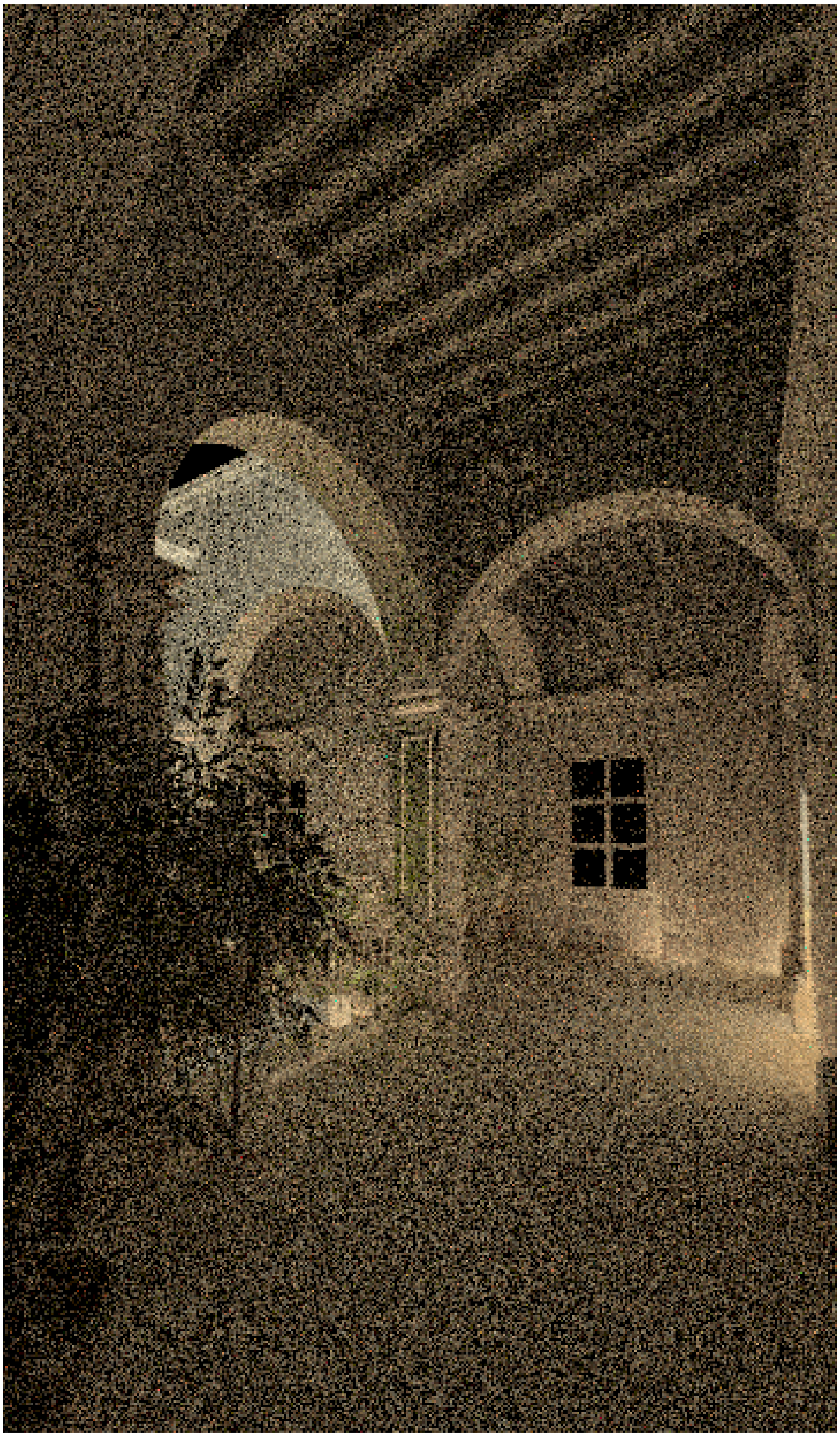


Figure 4.1: Path traced image with 8 SPP

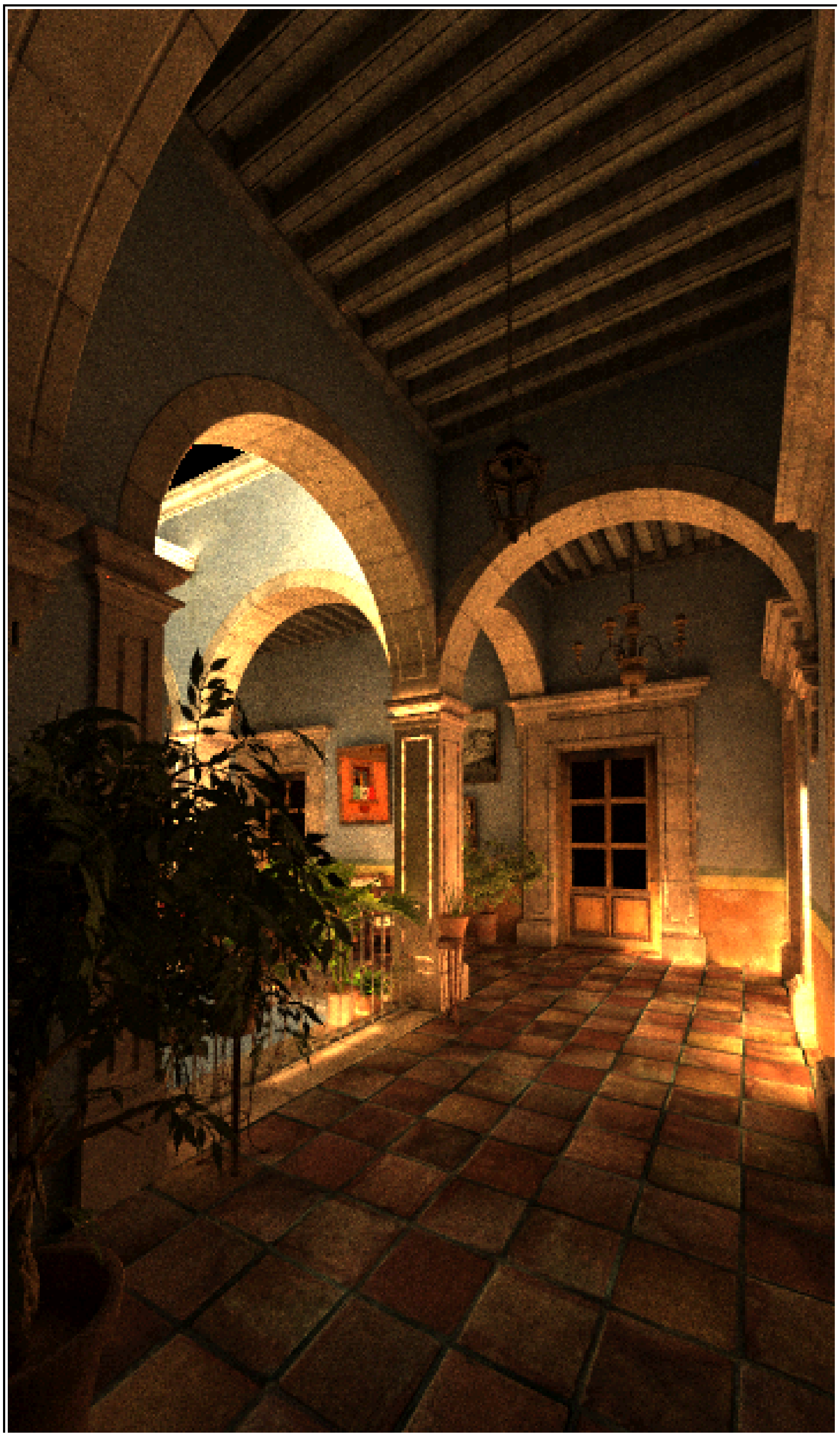


Figure 4.2: Reconstructed image

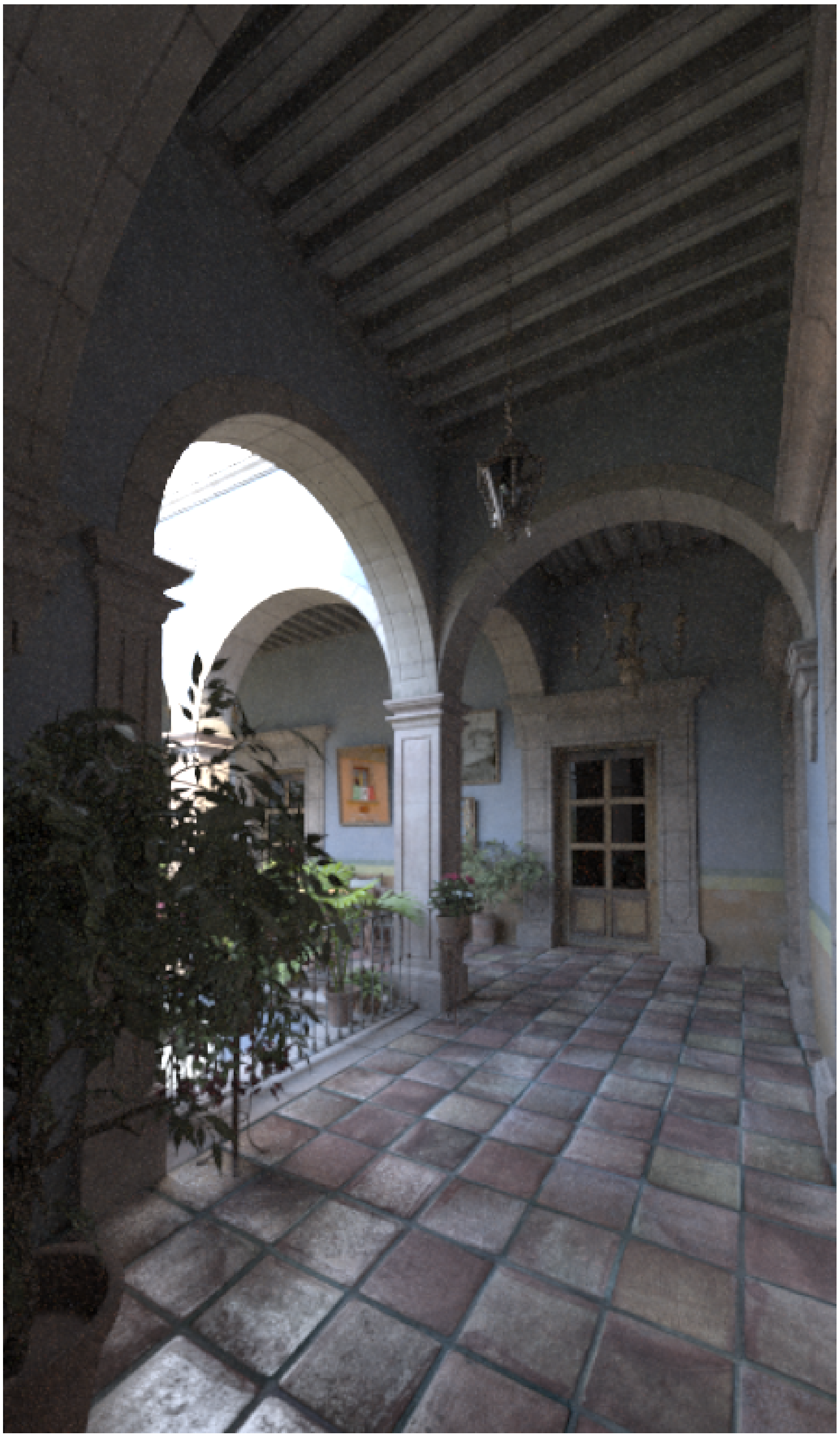
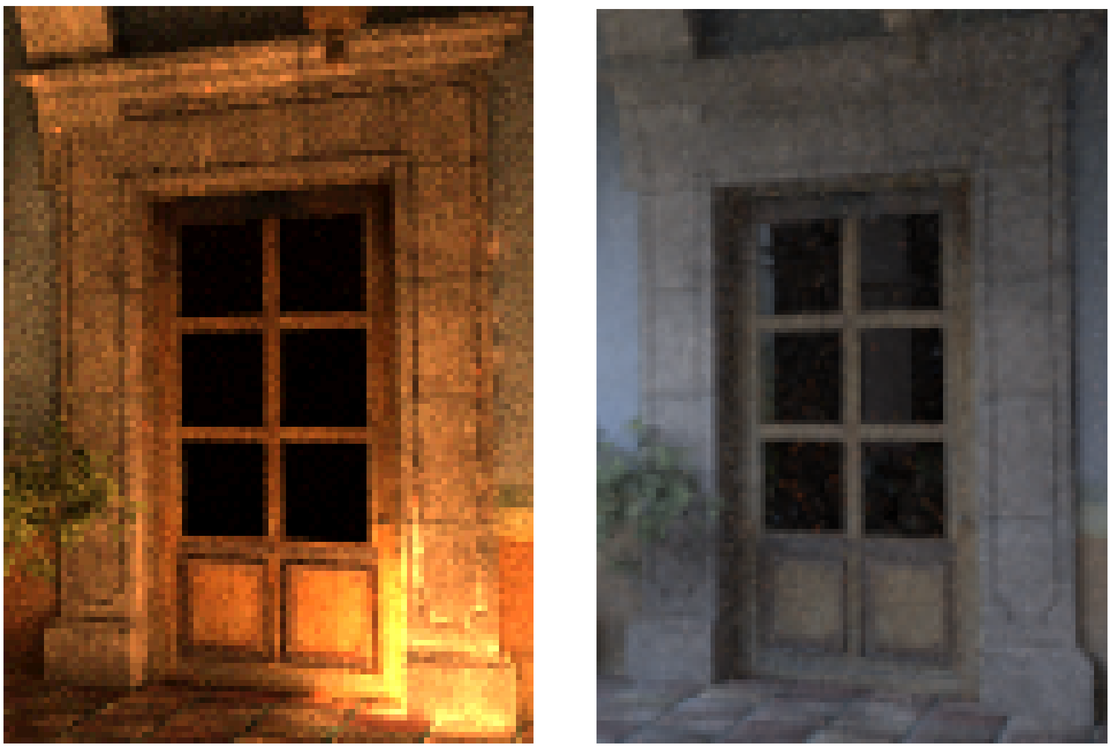


Figure 4.3: Path traced image with 2048spp

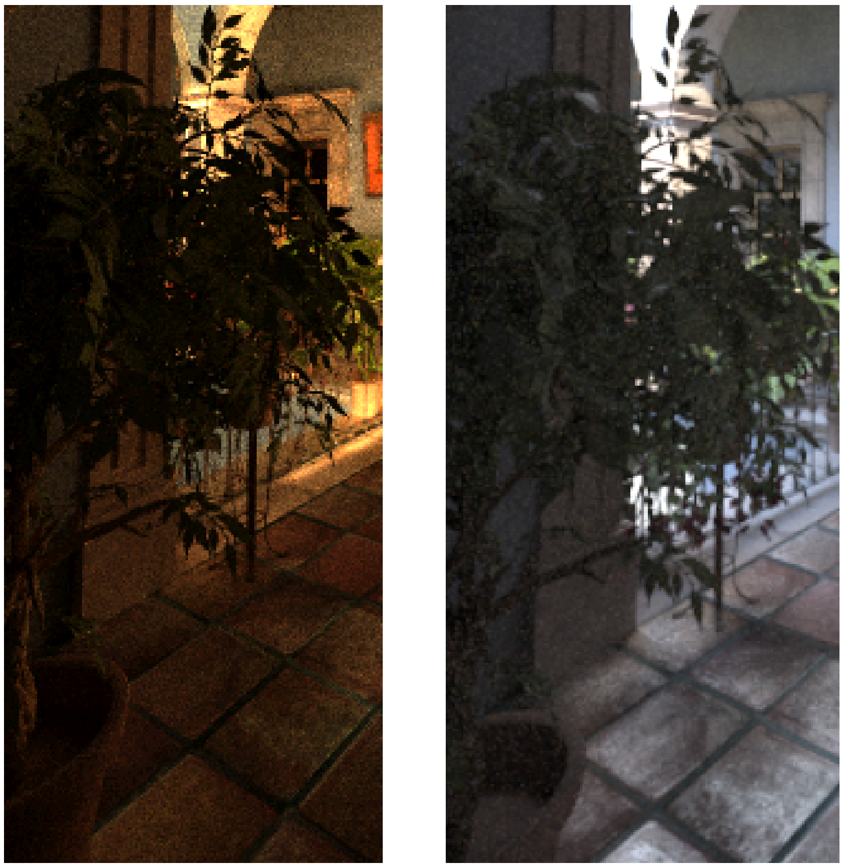
Figure 4.4 are side by side comparisons of a door in the scene. I think it is fair to say that the de-noising does a good job of resembling the ground truth. There is still some remaining noise.



(a) Reconstructed **(b)** Ground truth

Figure 4.4: Comparison between reconstructed and ground truth

Figure 4.5 depicts a comparison of smaller geometry of the scene, leaves. Because a leaf is smaller, than say a brick, there are less samples in the input that relates to the same surface. That means that the reconstruction has less information from related samples to work with. This leads to smaller geometries not being reconstructed as well as large ones.



(a) Re-con-structed

(b) Ground truth

Figure 4.5: Comparison between reconstructed and ground truth

4.2 Run times

Table 4.1 presents the run times for the C++ implementation, Halide implementation and path-traced images. The run times for C++ and Halide are for the de noising algorithm, with input and output in Figure 4.1 and Figure 4.2. The run time for the path traced image corresponds to Figure 4.3, which was rendered using PBRT with 2048 spp.

Type	Run time
C++	2h 32m 12s
Halide	2h 43m 53s
Path-traced (2048 spp)	3h 9m 8s

Table 4.1: Run times

Chapter 5
Discussion

5.1 Strengths and limitations of Halide

The functions presented in 2.4 are high level abstractions. Whenever we use f.ex. tile on a function there is code running under the hood to enable it. This enables the possibility of simpler syntax for scheduling a different order of processing the data. It also enables us to separate the algorithm from the scheduling. These are the strong points of Halide. Optimizations can more easily be explored and defining them is a lot simpler. These abstractions does come at a cost however, in order to enable them Halide has some restrictions in terms of what can be done within the language. It only handles feed forward pipelines. Figure 5.1 represents a feed forward pipeline.

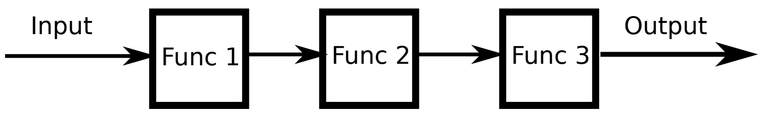


Figure 5.1: Feed forward pipeline

Figure 5.2 represents a feedback pipeline.

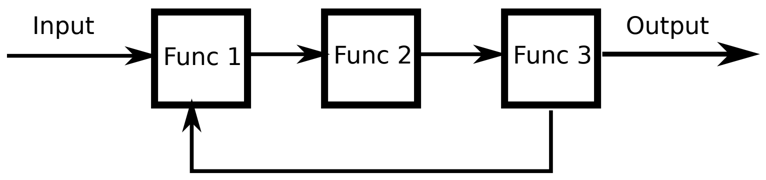


Figure 5.2: Feedback pipeline

Halide also only operates on image data. Data structures such as trees, lists or hash tables can not be expressed. [5] Halide does not have support for dynamic recursion, that is a recursive function that uses data computed in another recursive step for the current one [5]. That is because it is not possible to express higher-order functions in Halide, or a function that calls another function in other words. [5] This is a serious limitation for the de-noising algorithm, because it is very reliant on the octree, which is a tree type data structure generated through dynamic recursion. The octree is used in nearly every step of the algorithm.

5.2 Hybrid

Originally the intent was to have it fully running in Halide. However since the algorithm is so reliant on the octree, which can not be represented in Halide this was not a reasonable goal. One of the creators of Halide had the following to say: "It is much harder to imagine how the existing Halide language can generalize to encapsulate trees and other data structures in a unified and efficient way". [5] Therefore I decided to move forward with a hybrid version instead. Certain parts of the code, specifically ones that relies on the octree, is implemented in C++. Other parts, mainly in the second stage in section 2.3.3, are implemented in Halide.

5.3 The ideal Halide pipeline

An ideal Halide pipeline would make use of its strengths. The main optimization benefits comes from things such as vectorized computations and scheduling in order to improve the memory locality. Therefore we would like a pipeline where an image is read from file and uniform modifications are made to the entire structure. Memory locality cant be taken advantage of unless your reusing the data. An example of a situation where taking advantage of locality could be as follows. Imagine you want to averages 2x2 boxes of pixels. The resulting boxes are then averaged again in 2x1 boxes to produce the output. Figure 5.3 depicts this process. In the first step 2x2 boxes in A are averaged into 1x1 boxes in B. In the second step 2x1 boxes in C are averaged to 1x1 boxes in D, which is the result.

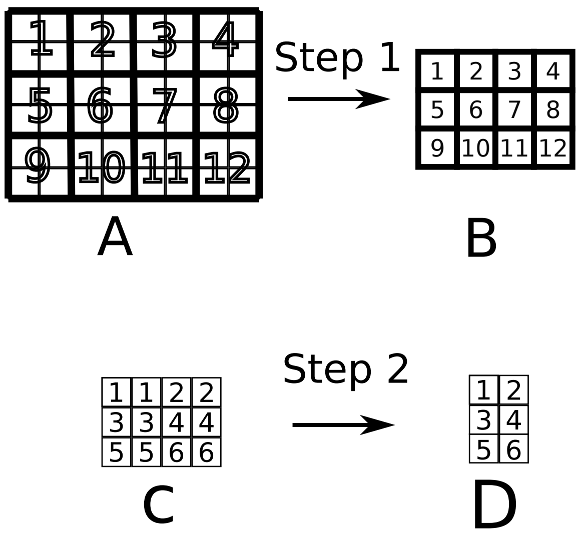


Figure 5.3: Visualization of algorithm

A typical C++ algorithm would compute the averages for every box in A, then compute every box in B to produce the output. This is not optimal because the result of the calculations is re-used in the stage B. This means that the result is no longer stored in the cache, and would need to slowly be read back into memory. A much smarter solution can be obtained by interleaving the stages, as in Figure 5.4. The averages are computed in step one, and immediately re-used in step two to produce the one resulting pixel in D.

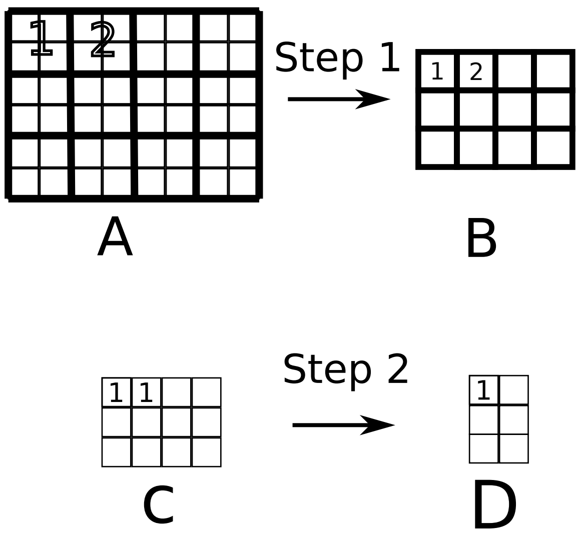


Figure 5.4: Interleaved stages

5.4 Pipeline in this project

The pipeline in this project does not have many computations that can be reorganized as in Figure 5.3. Data is fetched from a buffer and immediately put to use. Modifications are also not made to an input image. Instead we are utilizing the data from the path tracing process presented in section 2.3.2. Many types of operations are also not performed uniformly on the input data, but on a subset, which is problematic. An example of this could be during the reconstruction process. Before sending a reconstruction ray we must make sure that the hit point and origin are valid, meaning that they are actually in the scene. Sometimes the ray can "miss" the scene and go off in the distance and those samples simply do not contribute. While it is possible to compute conditionally in Halide, this leads to redundant work. Halide always works on the entire domain. Using conditional statements simply alters the output [5].

There are certain sections of code that can be made more efficient with Halide. As an example every pixel has 8 entries of the normal for the secondary ray, which is a vector of length three. In the filtering process these values need to be normalized. In this case we can take advantage of parallelism. Figure 5.5 depicts a single pixel and the 8 corresponding samples for the normal values of the secondary hit.

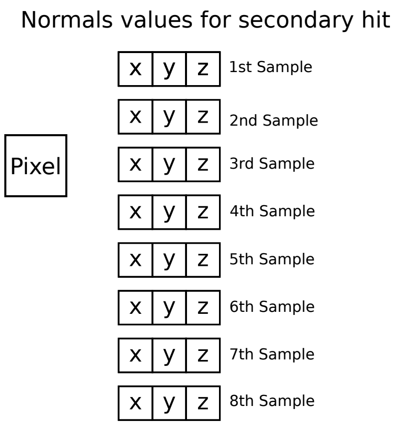


Figure 5.5: One pixel and its input for the normal

By storing all of the samples in 8 separate Func buffers, we can calculate the normals with vectorized computations. Figure 5.6 represents four pixels and their normal values for the secondary hit. The first sample for all pixels is stored in the first Func, second sample for all pixels in the second Func and so on.

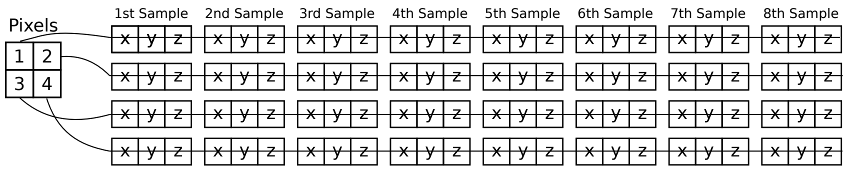


Figure 5.6: Normal values for a pixel in 8 different buffers

Code for calculating the normal for every sample in the func input is shown in listing 5.1

Listing 5.1: Normalizing in Halide

```
Halide::Func halide_normalize(Halide::Func f) {
    Halide::Func normalize;
    Halide::Var x;
    normalize(x) = f(x) / sqrt(f(0)*f(0)
    + f(1)*f(1) + f(2)*f(2));
    return normalize;
}
```

In order to highlight some of the problems with using Halide for some of the calculations consider the the problem of calculating an orthogonal basis for every input normal. Listing 5.2 contains code for this computation is as follows

Listing 5.2: Computing orthogonal basises in C++

```
Mat3f orthogonalBasis(const Vec3f& v)
{
        Mat3f m;
        Vec3f mx = v;
        float temp; // for swapping
        if( std::fabs(mx.x) > std::fabs(mx.y) && std::fabs(mx.x)
            > std::fabs(mx.z) )
        {
                temp = mx.x;
                mx.x = mx.y;
                mx.y = temp;
                mx.x = -mx.x;
        }
        else if(std::fabs(mx.y) > std::fabs(mx.x) && std::fabs(
           mx.y) > std::fabs(mx.z))
        {
                temp = mx.y;
                mx.y = mx.z;
                mx.z = temp;
                mx.y = -mx.y;
        }
        else
        {
                temp = mx.z;
23                  mx.z = mx.x;
24                  mx.x = temp;
25                  mx.z = -mx.z;
26          }
27          m.setCol( 1, cross(v, mx).normalized() );
28          m.setCol( 0, cross(m.getCol(1), v).normalized() );
29          m.setCol( 2, v );
30          return m;
31  }
```

This seems at first glance like something that could be taken advantage of by Halide. Every time this function is called a vector, mx, is computed in lines six to twenty six. The vector is then used to compute a matrix, m, in lines twenty seven to twenty nine. One might think that we could structure this in a similar way as in Figure 5.4, by interleaving the stages. In order to make it an effective Halide pipeline we would like to input every value for the normals, and get every matrix as an output. The problem lies with the temporary values and the switching of values between variables, in line eight, ten, fifteen, seventeen, twenty two and twenty four in listing 5.2. Because Halide is a feed-forward pipeline once a Func has been used to define another Func, the Func cannot be modified. In order to switch the values we must first store the temporary value in a Func, and then use the stored value to define another Func. This can be circumvented by sending in a single input normal and calculating the orthogonal basis for that normal like in listing 5.3.

Listing 5.3: Computing orthogonal basises in C++

```
Halide::Func orthogonalBasis(Halide::Func inpVec)
{
    Halide::Func vec,mat, col0, col1, col2;
    Halide::Var x,y;
    vec(x) = inpVec(x);
    if(Halide::abs(vec(0)).get() > Halide::abs(vec(1)).get() &&
        Halide::abs(vec(0)).get() > Halide::abs(vec(2)).get())
    {
        vec(3) = vec(0);
        vec(0) = vec(1);
        vec(1) = vec(3);
        vec(0) = -vec(0);
    }
    else if(Halide::abs(vec(1)).get() > Halide::abs(vec(0)).get
        () && Halide::abs(vec(1)).get() > Halide::abs(vec(2)).get
        ())
    {
        vec(3) = vec(1);
        vec(1) = vec(2);
        vec(2) = vec(3);
        vec(1) = -vec(1);
    }
    else
    {
        vec(3) = vec(2);
23          vec(2) = vec(0);
24          vec(0) = vec(3);
25          vec(2) = -vec(2);
26      }
27      col1 = halide_cross(inpVec, vec);
28
29      col1 = halide_normalize(col1);
30
31      col0 = halide_cross(col1, inpVec);
32
33      col0 = halide_normalize(col0);
34      col2(x) = inpVec(x);
35
36      mat(x,y) = 0.0f;
37      mat(1,y) = col1(y);
38      mat(0,y) = col0(y);
39      mat(2,y) = col2(y);
40      return mat;
41  }
```

This works when the input Func contains singular normals, but not for multiple normals. In listing 5.3 vec(x) can store four values. This is one more value than the (x,y,z) of the input normal. By using the fourth slot as a way of storing the temporary value this works fine for a singular normal. If we send in a Func that contains more than one normal and thus executes the code more than once, the fourth slot has already been used in order to define col1 in line twenty seven. We are not allowed to change the definition of a Func, in this case vec(x), that has been used to define another Func, namely col1. The problem with these types of solutions is that we might aswell be using C++. We can not take advantage of anything the Halide framework has to offer such as parallelism or any form of scheduling.

5.4.1 Run times

The run times were not improved at all by the Halide modifications I made. There were not that many opportunities to use Halide in an efficient way. The octree is present in many parts of the algorithm. In order to work around it I had to keep double buffers. This is a big problem, because for every image we are storing eight samples for every pixels. There are thirteen data points for every sample. For an image that is 360x620 there are over twenty million values stored in the buffers. There is also the problem of redudant computations described in section 5.4.

Chapter 6
Conclusion

This project set out to answer the following questions

How efficient is Halide for optimizing advanced image filters?

Specifically for this project there were no improvements in run time. I do not think it is fair to say that Halide is inefficient for all types of image filtering because of that. Halide has a lot of great features and I believe it can be used to produce highly efficient imaging code as long as the algorithm is suitable with the language. There are applications where Halide has been used effectively. Jonathan Ragan-Kelley, one of the designers behind Halide, presents a camera pipeline which is 3.4x faster than an hand-tuned implementation in C. [5] A camera pipeline is used to transform data from an image sensor into a usable image. Google is also using Halide in their Pixel Visual core, an image processing unit used in the Pixel 2 smartphone. [18]

Are there any challenges using Halide for these situations?

There were a lot of challenges in using Halide for the de-noising algorithm. In retrospect it is easy to say that it was a poor choice for testing Halide, specifically the octree is the biggest obstacle to overcome. Making it run fully in Halide was not a reasonable goal for one person in the given timeframe. Replacing the octree with a similar structure is a hard problem to solve. [5] In my opinion Halide is simply not suitable for pipelines that depend on such structures at the current time. Halide is quite different to other popular languages such as java, c++ or other. Implementations of an algorithm in java can rather easily be converted to c++ and vice versa. This is not the same for Halide. While optimization is made easier, there is also the downside that it is not as versatile in the data structures it can represent. Implementations that relies heavily on data structures such as trees or lists needs large structural changes to be implemented in Halide.

Future projects

There are some things I would change in order to make future research on this topic more conclusive. In future projects I would recommend building an algorithm in Halide from the ground up, where the focus and design of the algorithm is entirely on an Halide implementation. Because Halide is more limited than C++ there is a larger hurdle writing a Halide equivalent of C++ code than it is to write a C++ equivalent of Halide. That would make the run time comparisons more fair.

Bibliography

[1] C. Eisenacher, G. Nichols, A. Selle, and B. Burley. Sorted deferred shading for production path tracing. *Computer Graphics Forum, 32*, 2013. doi: 10.1111. URL `https://disney-animation.s3.amazonaws.com/uploads/production/publication_asset/70/asset/Sorted_Deferred_Shading_For_Production_Path_Tracing.pdf`.

[2] Battlefield v rtx ray tracing. URL `https://www.nvidia.com/en-us/geforce/news/battlefield-v-rtx-ray-tracing-reflections/`.

[3] J. Ragan-Kelly and A. Adams. Halide. URL `http://halide-lang.org/`.

[4] J. Ragan Kelly, A. Adams, S. Paris, M. Levoy, and S. Amarasinghe. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *SIGGRAPH*, 2012. URL `http://people.csail.mit.edu/jrk/halide12`.

[5] J. Ragan-Kelley. Decoupling algorithms from the organization of computation for high performance image processing. *ACM Trans. Graph.*, May 2014. URL `http://people.csail.mit.edu/jrk/jrkthesis.pdf`.

[6] J. Lehtinen, T. Aila, S. Laine, and F. Durand. Reconstructing the indirect light field for global illumination. *ACM Trans. Graph.*, 31(4):51:1–51:10, July 2012. ISSN 0730-0301. doi: 10.1145/2185520.2185547. URL `http://doi.acm.org/10.1145/2185520.2185547`.

[7] T.M Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 37(4):139:1–139:13, 2018.

[8] M. Gharbi, J. Chen, J.T Barron, S.W. Hasinoff, and F. Durand. Deep bilateral learning for real-time image enhancement. *ACM Transactions on Graphics (TOG)*, 36(4): 118, 2017.

[9] R.T Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4): 83:1–83:11, July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925952. URL `http://doi.acm.org/10.1145/2897824.2925952`.

[10] T. Whitted. An improved illumination model for shaded display. *SIGGRAPH Comput. Graph.*, 13(2):14–, August 1979. ISSN 0097-8930. doi: 10.1145/965103.807419. URL `http://doi.acm.org/10.1145/965103.807419`.

[11] J.T Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, page 143, Aug 1986.

[12] P. Matt, J. Wenzel, and G. Humphreys. Physically based rendering tool. URL `https://github.com/mmp/pbrt-v3`.

[13] R.D Maesschalck, D. Jouan-Rimbaud, and D.L. Massart. The mahalanobis distance. 50, January 2000. URL `https://www.sciencedirect.com/science/article/abs/pii/S0169743999000477`.

[14] D. Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. 10 1980.

[15] Computer latency. URL `https://www.prowesscorp.com/computer-latency-at-a-human-scale/#_edn1`.

[16] N. Kalantari, F. Rouselle, P. Sen, S.E. Yoon, and M. Zwicker. Denoising your monte carlo renders: Recent advances in image-space adaptive sampling and reconstruction. *SIGGRAPH*, 2015. URL `http://cgg.unibe.ch/publications/denoising-your-monte-carlo-renders`.

[17] URL `http://halide-lang.org/tutorials/tutorial_introduction.html`.

[18] O. Shacham and M. Reynders. URL `https://www.blog.google/products/pixel/pixel-visual-core-image-processing-and-\machine-learning-pixel-2/`.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2018-11-01

EXAMENSARBETE Filtered Path tracing using Halide

STUDENT Fredric Berg
HANDLEDARE Michael Doggett (LTH)
EXAMINATOR Flavius Gruian (LTH)

Framtiden för bildbehandling?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Fredric Berg**

Halide är ett modernt programmeringsspråk designat för bildbehandling. Detta arbete utforskar dess styrkor och svagheter genom att använda det för optimeringen av en bildrekonstruktions algoritm.

Bildbehandling är en viktigt del av dagens samhälle. Det används i många olika fält som datorgrafik, medicin och datorseende. Det största problemet som förhindrar fortsatt utveckling är hårdvara och exekveringstider. I mobila enheter är det speciellt viktigt, då hårdvaran är ännu mer begränsad. En applikation som är begränsad av exekveringstid är path tracing. Det är ett sätt att rendera fotorealistiska bilder av hög kvalitet. Det största problemet för sådana verktyg är att renderingstiden är väldigt lång. Genom att ha väloptimerad kod kan man sträcka gränserna för vad som är möjligt ännu längre. Med utvecklingen av hårdvara och större fokus på optimering skulle det kunna bli en möjlighet att utnyttja path tracing eller liknande metoder för att rendera bilder i realtid. Halide är ett programmerinsspråk som är designat för bildbehandling. Principen bakom språket är att det ska bli lättare att optimera kod jämfört med andra språk som används i dagsläget. I språk som C++ är det krångligt att skriva effektiv kod och den blir också svår att tolka för oinsatta. I halide kan en optimerad version skrivas med enklare kod och med mycket bättre överblick. I det här examensarbetet implementeras en bildrekonstruktions algoritm med hjälp av Halide för att undersöka möjligheterna och begränsningar med språket. När man renderar en bild med path tracing så finns det mycket information att hämta från processen. Ju längre man renderar desto högre kvalitet får den slutgiltiga bilden. Genom att köra en snabb rendering får man en lågkvalitativ bild. Algoritmen som presenteras i detta arbete tar informationen från en snabb rendering och använder den för att återskapa bilden. På detta viset får man en hög kvalité på den slutgiltiga bilden men framförallt mycket kortare renderingstid.

Arbetet visar att Halide inte är den ultimata lösningen för optimering av all typ av kod. På grund av begränsningar inom språket är det inte lämpligt att använda det för alla typer av algoritmer. Halide stödjer bland annat inte data strukturer som träd, vilket är vanligt förekommande inom datorgrafik. Det medför problem som försämrar prestandan. För det här projektet blev det ingen förbättring av exekveringstid jämfört med en implementation i C++. Det kan istället fungera som en studie om vad Halide är lämpligt och olämpligt för. Det belyser det som gör Halide till ett effektivt språk och även problematiken i vissa användingsområden.