Universität des Saarlandes
Cluster of Excellence on Multimodal Computing and Interaction
Department of Computer Science

Master's Thesis

A GPU Line Sweeping Framework

submitted by
Constantin Berhard
in **2016-12-01**

Supervisor
Dr. Piotr Didyk

Advisers
Dr. Piotr Didyk
Dr. Tobias Ritschel

Reviewers
Dr. Piotr Didyk
Dr. Karol Myszkowski

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____
Date Signature

Abstract

Line Sweeping has been shown to be a well-performing algorithm to solve the Screen Space Ambient Obscurance[Tim13] problem on graphics hardware. This brings up the question on whether Line Sweeping[1] is a limited, special purpose algorithm, only suitable for Screen Space Ambient Obscurance or if it also has different scopes of application. In the latter case, we ought to explore how and to what degree Line Sweeping can be generalised, to which kind of problems it is applicable and how it performs, in terms of execution speed and memory footprint. Furthermore, we will develop a GPU Line Sweeping framework, vastly reducing the burden of implementing a Line Sweeping algorithm.

[1] not to be confused with Sweep Line algorithms, see section 4.1.2

Contents

Chapter 1

Introduction

In the real world, diffuse light reaches corners and otherwise obstructed regions less intensively than flat or even convex surfaces. The objective of Screen Space Ambient Occlusion (SSAO, chapter 3) is to simulate this effect convincingly for the human eye. It achieves this by observing the fragments[2] in the depth buffer. If the neighbourhood of a fragment is on average closer to the viewer than the fragment, some diffuse light is occluded and the respective fragment is shaded darker.

A naïve approach to this problem is for each fragment in the depth buffer to search for occluders in a specified area around it. Line Sweeping samples the depth buffer for (a finite subset of) all possible directions, *memorising* the last seen occluders. So at each fragment the sweep reaches, the largest occluders are already known, at least the ones which lie in the direction the sweep came from. In a second step, for each fragment, the information from all sweep directions is combined to find the largest occluders in all directions.

The Line Sweeping algorithm has a powerful advantage: It does not need to search the depth buffer for a near point shadowing the point in question. Line Sweeping already has the information about neighbours available when it reaches a point as those neighbours have previously been visited.

In the search for a generalisation of Line Sweeping (chapter 4), we recognise that it yields information about what interesting things[3] exist in a specific direction. We must acknowledge, however, that this is only efficient if we want this information for all points in some region. For only a few points it might be more efficient to trace rays individually than to traverse the whole problem space from many directions, like Line Sweeping does.

There are several problems where the former setting applies: In *diffusion inpainting* (chapter 6), the colour of an unknown point is determined by the known points in the area. So if we know for each unknown point, which known points are near and their colour, we can calculate its colour.

In *inside-outside testing* (chapter 7), a point is inside, if it is encircled by geometry (e.g. a tight triangle mesh). So if we know for each point, how much geometry surrounds

[2] as in *fragment shader*; If in doubt, think of it as "pixels".

[3] The vague nomenclature will be superseded in chapter 4.

it, we know whether it is inside our outside of the geometry. Note that this problem is relevant at least in the two- and three-dimensional settings. On the basis of this problem, we will examine the applicability of Line Sweeping to higher dimensions.

There are more problems (chapter 8), which can be solved using Line Sweeping algorithms. As Line Sweeping is such a versatile algorithmic strategy, we will develop a Line Sweeping framework (chapter 5). We will do this in the spirit of frameworks and domain-specific languages, such as AMD Radeon Rays[Dev16] and Simit[Kjo+16] – to mitigate the burden of development for high-performance GPU solutions to common problems. We will be building upon commonly used tools and libraries, such as C++, PHP, and OpenCL to facilitate the task of implementing a Line Sweeping algorithm for new problems, while achieving increased run-time performance by using Graphics Cards. Finally, we will analyse solutions based on Line Sweeping using that framework and observing how they interact with the framework.

Chapter 2

Contribution

Line Sweeping has been introduced in [TW10] as a technique for calculating the lighting of *height fields*[4]. In [Tim13] it has been applied to the *depth buffer*[5] for screen-space lighting.

The main contribution of this thesis is to show the generality of the Line Sweeping approach and to provide a Line Sweeping framework facilitating solutions using the technique. The individual contributions are listed in the following in order of their appearance in this document.

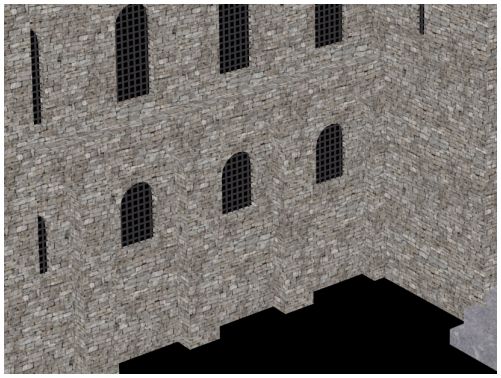
- We reconsider Line Sweeping Screen Space Ambient Occlusion with generality in mind. This leads to a generalisation of Line Sweeping to spaces of arbitrary finite dimensionality.
- Line Sweeping cannot be applied to any kind of problem, so we formulate criteria for effective applicability of this approach.
- We implement a framework facilitating Line Sweeping programming in CPU code, GPU code, and in the preprocessing of GPU code. The framework is built upon common technologies like C++, OpenCL and PHP for increased compatibility and usability. It is based on the general definition of Line Sweeping and is thus applicable to a variety of problems.
- We show how Line Sweeping can be applied to two-dimensional *diffusion inpainting* and to three-dimensional *inside-outside testing*. These use cases benefit from the presented framework, thereby demonstrating its utility.

The following chapters differ greatly in their respective related work. Therefore, each chapter includes its own relevant background separately.

[4]Height fields are planes with their height defined by an image or a function.
[5]The depth buffer stores the distance from the camera to each rendered scene point.

Chapter 3

Screen Space Ambient Occlusion



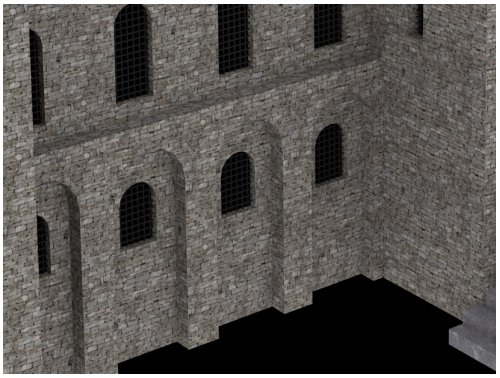


Figure 3.1: Renderings without (left) and with (right) ambient occlusion (Model from [And14], rendered using Blender[Fou16]). Note that in practice renderings without AO do not necessarily look so bad, as they often employ alternative forms of lighting.

Ambient Occlusion[ZIK98] is a technique to darken regions of a three-dimensional space, if there is geometry around it, which blocks some of the diffuse light from the environment. This applies to corners in a room, for example. Ambient Occlusion greatly improves the appearance of three-dimensional renderings (see figure 3.1).

Applying this to the screen space, using the depth buffer as the sole source of geometry information, leads to Screen Space Ambient Occlusion (SSAO). SSAO is widely used [Mit07] [07] [16b] [16a] and has numerous implementations [Mit07] [BS08] [RBA09] [MSW10] [McG+11] [TW10] [Tim13]. In this chapter, we will roughly examine Ambient Occlusion and Screen Space Ambient Occlusion Then we will have a closer look at SSAO implementations employing Line Sweeping.

3.1 Ambient Occlusion

Ambient Occlusion is a physical lighting effect that follows from the global illumination of a scene, specifically from its diffuse lighting. Diffuse lighting can be directly calculated

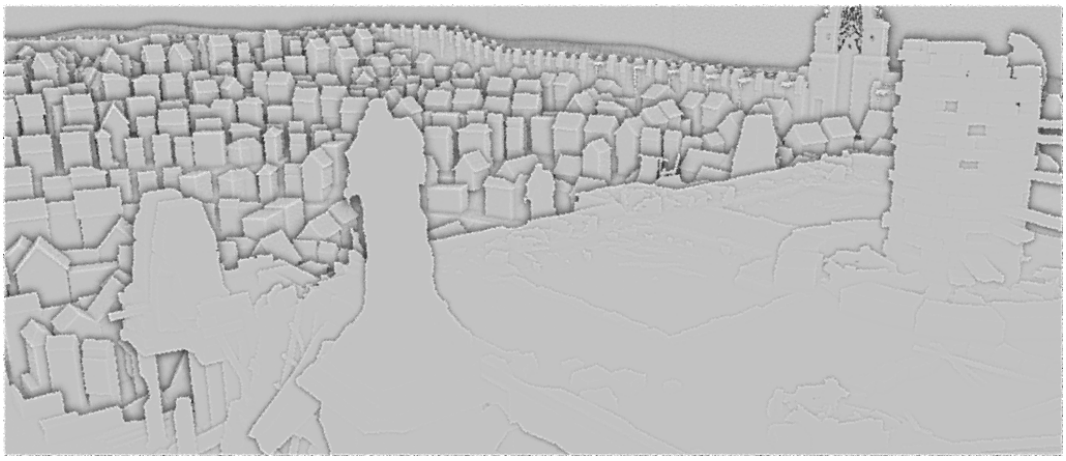


Figure 3.2: Screen Space Ambient Occlusion. The depth map is taken from the open source movie "Sintel"[But+12].

by costly ray tracing rendering methods, involving following many simulated light rays through the scene. For the sake of faster calculation, we only want to approximate this effect. Instead of simulating light sources explicitly, we will assume that "the environment is lit by a perfectly diffuse light coming from everywhere" – [ZIK98]. Zhukov et al. also give a mathematical description of ambient occlusion and its physical justification, which are beyond the scope of this work.

3.2 Screen Space Ambient Occlusion

Even with the simplification of not considering light sources, there is still a lot of geometry to process, which may well be infeasible in real-time applications.

Modern rasterising rendering engines generate a depth map as a byproduct of the rendered image. It is needed to decide the depth ordering of the rendered objects, to be able to skip those that are behind ones already drawn. Such a depth map (which can also be thought of as a height map) is a simplified description of the scene geometry. It shows only objects in the field of view and provides more detailed information about objects closer to the viewer than about those far away because of the usual perspective projection[6].

The process of calculating Ambient Occlusion using only this simplified geometry description is called *Screen Space Ambient Occlusion* (SSAO). Note that some geometry information may be lost in the transfer from the complete scene description to the depth buffer as some objects may be hidden behind others. Consequently, SSAO has no way of correctly calculating their shadows. Wide-spread use of SSAO shows that the resulting

[6]The latter part of this statement is not true for orthographic renderers. However, those are very uncommon and furthermore, SSAO remains a viable option in such a scenario.

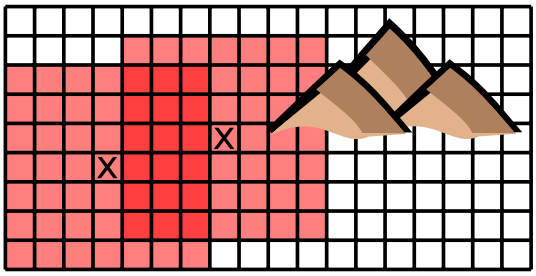


Figure 3.3: Traditional SSAO algorithms sample the neighbourhood (red) of each output point for occluders. Note the overlap: The same area is sampled many times. And note that the left sampling area does not contain the occluder on the right because it is outside of the sampling radius.

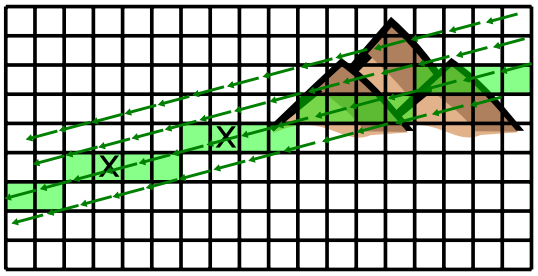


Figure 3.4: LSAO sweeps through the space, remembering occluders. The depicted sweep ray will inform both marked points of the occluder. Note that LSAO also samples the same pixels multiple times; once for each sweep direction.

inaccuracies are sufficiently minor and worth the performance boost. See figure 3.2 for an SSAO example rendering.

3.3 Line Sweeping Ambient Occlusion

Traditional SSAO algorithms sample the neighbourhood of a point for occluders[SA07][BSD08] (see figure 3.3). They do this for each pixel. In Line Sweeping Ambient Occlusion (LSAO), many threads iterate over the depth buffer in many directions to calculate the SSAO effect (see figure 3.4).

3.3.1 Ray Tracing Formulation of SSAO

We will now derive a ray tracing-compatible equation for Screen Space Ambient Occlusion from the general rendering equation[Kaj86][ICG86]. The formula we aim for stems from [BSD08]. Bavoil et al. do not attempt a derivation, but instead justify their equation using convincing images. Our derivation attempt aims at creating an understanding of why and how the formula works. It is not a proof or justification.

The rendering equation for a perfectly diffuse, colour-agnostic reflector is

$$L(p) = \int_{\Omega} L_i(p, \omega_i)(\omega_i \cdot n)\mathrm{d}\omega_i \tag{3.1}$$

L is the total radiance going out from point p on the surface of the reflector. Ω is the hemisphere over p. n is the surface normal at p. ω_i points towards the incoming light. L_i is the incoming radiance from direction ω_i.

If we split the hemisphere integral by its azimuthal directions[7] we get

$$L(p) = \int_0^{2\pi} \int_0^{\pi} L_i(p, \omega_{\phi,\theta})(\omega_{\phi,\theta} \cdot n) \mathrm{d}\phi \mathrm{d}\theta \tag{3.2}$$

The inner integral is the amount of light coming from an azimuthal direction including all its elevation angles. As there are no explicit light sources, but only occluders, we can rewrite the integral as

$$L(p) = \int_0^{2\pi} f(p, \theta, n, o(p, \theta)) \mathrm{d}\theta \tag{3.3}$$

$o(p, \theta)$ is the collection of information about any occluding geometry in direction θ visible from point p. $f(p, \theta, n, o)$ is a function calculating how much light arrives at a point from an azimuthal direction if the light is only blocked by the occluders o.

Using an exact formula for f is cumbersome and calculation-intensive. We will use approximative functions involving only the largest occluders[8] and a falloff function to limit their impact over long distances. [BSD08] provides a visually pleasing formula:

$$f(p, \theta, n, o) = V(p, \theta, n, o) W(|p - o_p|) \tag{3.4}$$

V is the fraction of the horizon, which is not occluded by the largest occluder in o. W is a weighting function depending on the distance of the largest occluder. The actual functions are not of utter importance for understanding the required concepts. For the sake of completeness, we provide them below.

$$V(p, \theta, n, o) = sin(h(o)) - sin(t(\theta, n)) \tag{3.5}$$

$$W(d) = \frac{R}{R + d^2} \tag{3.6}$$

$h(o)$ is the elevation angle of the largest occluder. $t(\theta, n)$ is the angle between the reflector surface and the depth buffer plane *in direction* θ. R is a constant to influence the area of effect for occluders to yield pleasing results. Note that the formula for W is the slightly modified version from [Tim13].

Discretising the formula for L yields the ray tracing formulation of the problem for k directions $d(i)$ well-spread in $[0, 2\pi]$ (e.g. $d(i) = 2\pi \cdot \frac{i}{k}$).

$$L(p) = \frac{1}{k} \sum_{i=0}^{k-1} f(p, d(i), n, o(p, d(i))) \tag{3.7}$$

This formula corresponds to calculating the arriving light for each point by tracing a ray $r = p + \lambda \cdot d(i)$ and calculating the amount of light coming from that direction (i.e. calculating f).

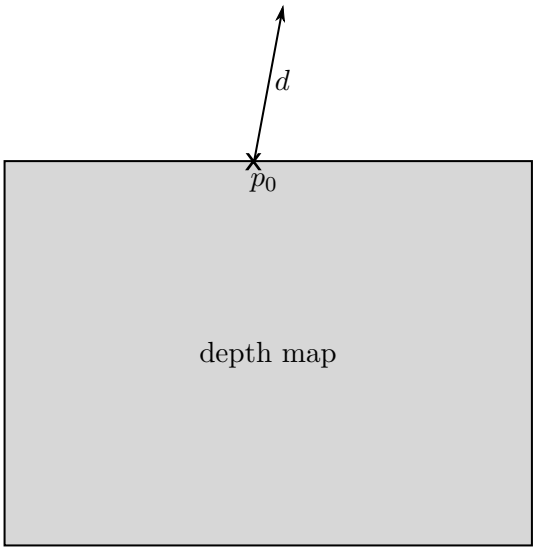


Figure 3.5: A point p_0 at the border of the depth map can not be occluded from a direction d pointing outside the depth map.

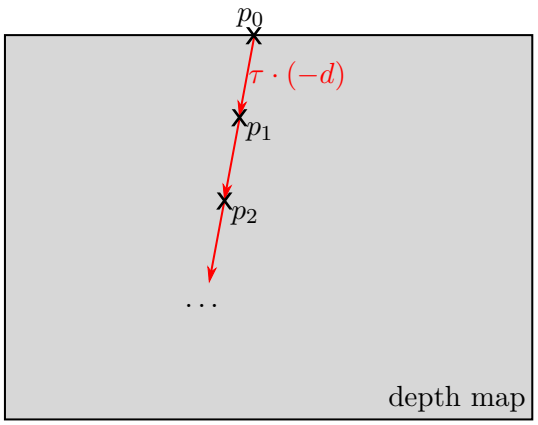


Figure 3.6: Iteration of the position p_i along direction $-d$ with steps of size τ

3.3.2 Application of Line Sweeping

Screen space ambient occlusion dictates, that all relevant occluders are arranged in the depth map. So, if we are at a border of the depth map at point p_0 looking in some direction d pointing outside the depth map, there can be no occluders blocking the light (see figure 3.5). So $o(p_0, d) = \perp$ and $f(p_0, d, n, \perp) = 1$ for all such border points.

We then iterate our position towards the inside of the depth map ($p_{i+1} := p_i + \tau \cdot (-d)$, with step size τ, see figure 3.6). Along the way, we memorise all information about important occluders as we pass them. This means we calculate $o(p_{i+1}, d)$ from the depth map value at p_{i+1} and $o(p_i, d)$. Note that while f only uses information about the largest occluder, o may need to store more information. Doing this in a way such that calculating occlusion values takes an amortised constant time is possible, but beyond the scope of this document. A detailed description of a suitable data structure can be found in [TW10].

3.3.3 Directional Occlusion

The described method can easily be expanded to support lighting from an environment map (e.g. cube map), resulting in *Line Sweeping Screen Space Directional Occlusion*

[7] i.e. directions parallel to the reflector surface
[8] "largest" in terms of occlusion for the point in question, not in terms of the raw depth map data

(LSDO). We know where and how high the largest occluder is. We can create a ray from the current position p_i to the peak of the occluder and track that ray into the environment map. The resulting point and those at greater elevation angles are what sheds light on p_i from the environment map. If we average all colours from the environment map in this angular region, we get the light colour reaching p_i from one direction. A faster alternative is using a pre-averaged environment map. In this case, each pixel in the environment map is already averaged with all pixels at the same azimuthal[9] angle and greater elevation angles. A single lookup yields the light colour. See figures 3.7 through 3.10 for examples.

The displayed environment maps are hemisphere maps. The horizontal axis is the azimuthal direction (0 to 2π) and the vertical axis is the elevation angle (0 to π). To simulate realistic shadowing, high-dynamic-range environment maps are recommended.

3.3.4 Benchmark

The Line Sweeping framework is designed in a way that allows for maximum performance. Actual optimisations are however beyond the scope of this thesis. The presented results are thus not to be considered as the maximum performance expectable from a Line Sweeping approach. A detailed discussion can be found in section 8.1.

The hardware for all benchmarks in this document is an *Intel Core 2 Q9550* with *4 GB* of *DDR2-800* memory and an *AMD Radeon HD 7850* with *2 GB* of *GDDR5 memory*. The software is *Microsoft Visual Studio 2013 Community* and the *AMD APP SDK v3.0*. All timings are the median of three runs. Line Sweeping timings are taken only for the execution of the relevant kernels (i.e. not including memory transfers etc.) beginning and ending after an OpenCL queue finish. The method of measurement is the C++11 `std::chrono::steady_clock` class.

For LSDO the size of the input buffer (i.e. the depth map) and output buffer are 1024x436 pixels. The environmental lighting is stored in a hemisphere map of the size 1024x256 pixels. Figures 3.7 and 3.8 have each been created using 16 sweep directions and a *grid size*[10] of 2 pixels. The time for the execution of the sweep kernel and the accumulation kernel combined is 94 ms. The fastest measured time for this task is 62 ms. This large deviation may imply that the method of measurement is erroneous (e.g. the OpenCL queue finish command we issue could have a large and unpredictable overhead). As optimisation is not the core priority of this thesis we refrain from investigating the matter further.

For 8 sweep directions and a grid size of 1.8 pixels, the required median time is 47 ms. If the sweeps are executed in four groups of two sweep directions each (see section 5.8), the total required time is 78 ms.

[9] parallel to the depth buffer in this case

[10] ray step size and space between neighbouring rays of the same sweep direction

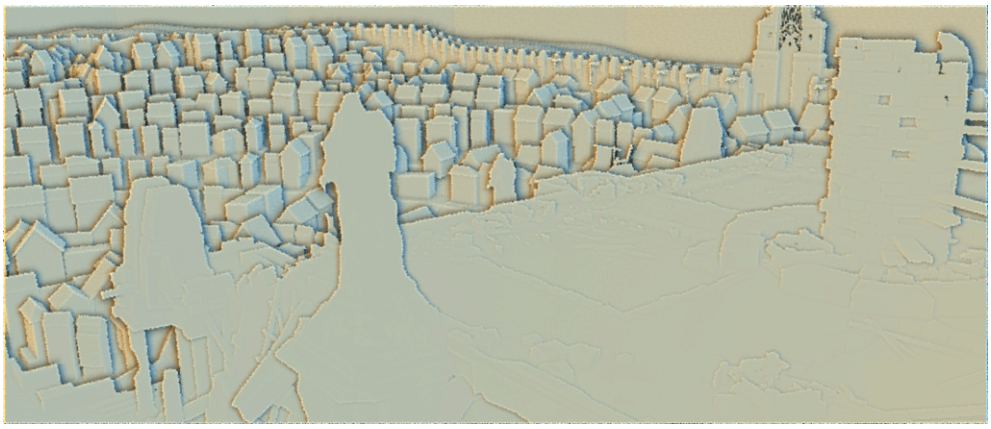


Figure 3.7: Lighting with a colourful environment map. The colour values are scaled for better visibility.

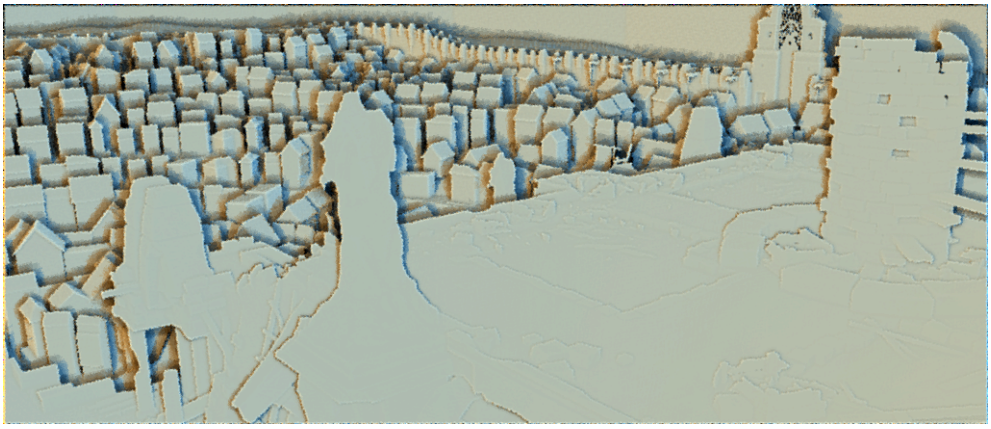


Figure 3.8: Environment map simulating a more dusky light situation. The colour values are scaled for better visibility.

Figure 3.9: The "colorful" hemisphere map of figure 3.7. Pre-averaging does not matter here, as the map is constant in the elevation angle direction.

Figure 3.10: The "dusky" hemisphere map of figure 3.8. It is used like a pre-averaged environment map, but a smoother gradient from black to colourful would be more realistic.

Chapter 4

Line Sweeping

In the previous chapter, we have observed how Line Sweeping can be applied to the Screen Space Ambient Occlusion problem. In this chapter, we will generalise the idea of Line Sweeping. We will examine what Line Sweeping is, how it works and what kind of information it can retrieve.

4.1 The Essence of Line Sweeping

The following list identifies entities in the Line Sweeping Screen Space Ambient Occlusion (LSAO) process, and for each presents a generalisation.

Geometry of the Frame Buffer All Line Sweeping calculations happen inside of the frame buffer. If a thread iterates any positions outside of the frame buffer, the result is not relevant because we are not supposed to render that area and furthermore the depth map is undefined there, so we cannot find information about any occluders. The frame buffer geometry also indicates where to arrange the iteration paths of the individual sweep threads. So it is reasonable to demand that for every Line Sweeping algorithm, there is a well-defined, finite operation space. We will call this the *problem space*.

There are problems, which are not solvable in a two-dimensional environment[11]. Especially three-dimensional problems are not uncommon in computer graphics. So we make no restriction on the problem space dimensionality other than that it has to be finite. To keep things as simple as possible we also require the problem space to be Euclidean[12].

Depth Map Data The depth map provides information about occluders, needed by each Line Sweeping thread. The data structure in use does not need to be a

[11] Ambient occlusion as discussed is a problem arising from a three-dimensional space. However, the problem space to be considered in Line Sweeping *Screen Space* Ambient Occlusion is two-dimensional. The third dimension is hidden only in the depth map and no iteration involving the third axis is required.

[12] This means e.g. we do not support Line Sweeping in a function space or on the surface of a sphere.

pixel map. In some situations, information about occluders could be given by a completely different data structure (e.g. a tree) or by a generator function. For entirely different problems, the information required to solve it might not even be related to occluders. In short, the required information may be arbitrary, so the generalisation of the depth map from LSAO is *input data*. This will often be a data structure mapping the domain of the problem space to some quantity, in which case we will call it *problem domain data*.

The Frame Buffer as a Canvas Besides suggesting *where* to work on the problem, the frame buffer also is a means of delivering output data to the display. Not all problems involve displaying images on a computer screen, so we need to generalise this concept. We distinguish between the *sweep data* (i.e. the data generated by each thread separately) and the *overall output* (i.e. what will be shown on the screen, written to a file etc.). The overall output is the accumulation of the sweep data from all sweep directions. In the case of LSAO, the accumulation operation is the calculation of the arithmetic mean. We will call the process of converting sweep data to overall output *data accumulation*.

Memorising the Occluders In Line Sweeping ambient occlusion, the information about occluders is stored into a data structure which yields the current occluder information (i.e. how far away and high are the obstacles occluding the current pixel or fragment). In General, the sampled input data must be rearranged so that the local memory requirements are not too demanding (e.g. copying all sampled information is infeasible) and that the currently important information is readily available. Rearranging the data must happen on-the-fly while iterating over the problem space. We will call this process *information condensation*.

The essence of Line Sweeping is iterating over a *problem space* from different sweep directions. While doing that, *input data* corresponding to the current position of the iteration is observed and *condensed* into a data structure. This information is instantly used to output the *sweep data*, which is afterwards[13] *accumulated* to create the *overall output* of the algorithm.

4.1.1 Applicability of Line Sweeping

Further considerations about the process reveal problem properties benefiting the application of Line Sweeping.

Density of the Result Line sweeping iterates over the whole problem space, yielding output at every point it passes. If there are some positions where no information is required, this may still be feasible. But for problems where the required results are sparsely spread in the problem space, a direct ray tracing approach might be more useful.

[13] Either immediately or in a second pass. See section 4.3

Large Area of Influence Line sweeping easily provides condensed information about remote parts of the problem space. If only information from a small neighbourhood is relevant, directly sampling it might be more useful.

Condensation Coherency The (condensed) data needed to calculate a result at one point of the Line Sweeping process should be similar to the data that will be required for the subsequent points in the Line Sweeping iteration. If every result position requires *different* information about the data residing in a common direction, it may be difficult to condense this information into a feasible amount of memory.

We observe that Line Sweeping is useful if for (nearly) all points in a region of a space, we need to know interesting facts about (possibly far away) data from other regions of that space. The essential advantage of Line Sweeping is that when the iteration arrives at a point, the relevant information is already available.

4.1.2 Similarities to Sweep Line Algorithms

Line sweeping should not be confused with *sweep line algorithms* (see e.g. [For87]). In a sweep line algorithm, the problem space is processed in only one direction. Whenever the sweep line hits something interesting, it is incorporated into *one* intermediate data structure. This is not an inherently parallel process. Sweep line algorithms can be understood as feeding data to a problem-solving function in a spatially sorted way.

Line sweeping, on the other hand, processes the problem space from many different directions. Instead of the sweep line, there are independent sweep rays. Each sweep ray does not intersect every part of the input in the problem space and they do not necessarily travel alongside each other, i.e. a ray can be far ahead of its neighbour ray during the iteration process.

4.2 Mathematical Definition and General Algorithm

While section 4.1 gave an intuition of Line Sweeping, exact considerations require formalisation. Line Sweeping needs a problem space P (e.g. a convex subset of $\mathbb{Q}^2$ or $\mathbb{Q}^3$, see Figures 4.1, 4.2) and a function f defined in P (i.e. the problem domain data). It can efficiently provide data about interesting function values of f for every point $p \in P$, even if they are far away from the currently considered point. The respective condensation function defines the meaning of "interesting function values". For LSAO this would be "What large occluders are in the depth map, and how far away and how large are they?".

Line Sweeping works like this (compare to algorithm 1): First a finite set of directions $D \subseteq \overrightarrow{P}$ is chosen. For each direction, there is a set of all rays pointing in that direction, starting at a border of the problem space and intersecting the problem space. This set is infinite, so it has to be sampled, in our case using a regular grid. For each such ray, a local data structure (i.e. the condenser) is created. The rays are then moved along their direction vector in small steps. After each step, f is sampled and the result is incorporated into the condenser. The condenser then outputs a value, derived from the

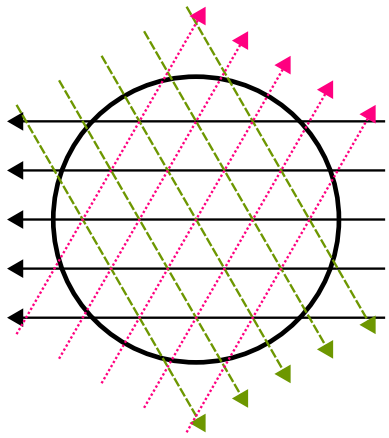


Figure 4.1: rays from three sweep directions ($D \subseteq \overrightarrow{P}$) in two dimensions over a circular problem space P $\subseteq \mathbb{Q}^2$

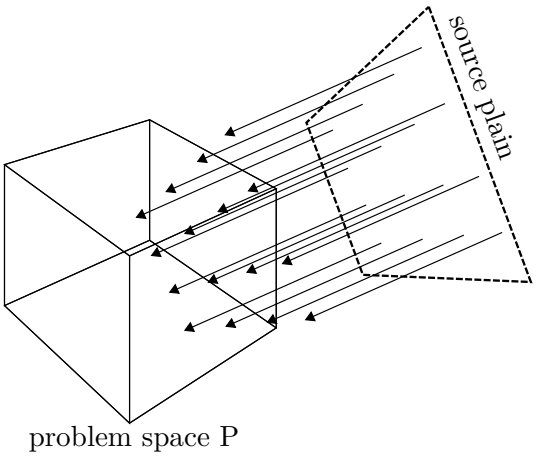


Figure 4.2: rays from a single sweep direction (D) in three dimensions over a cubic problem space P $\subseteq \mathbb{Q}^3$

samples of f it has seen so far. This value is stored as the *sweep data*. Iteration for a ray stops when it is outside of the problem space.

Algorithm 1 Basic Line Sweeping Algorithm

```
1  for d⃗ in D:
2    for ray in sample({ray = a+λ · d⃗ : ray hits P ∧ a ∈ Border(P)}):
3      c = new InformationCondenser();
4      while a ∈ P:
5        c.digest(a,d,f(a));
6        output[a, d⃗] = c.getInterestingData();
7        a += stepSize · d⃗;
```

The definition of "interesting" is hidden in the definition of `InformationCondenser`. In the case of LSAO, it stores the maxima of the depth map, their height, and positions, removing old occluders as their distance becomes sufficiently large for them to not influence subsequent points anymore.

Note that for an efficient implementation on a GPU, the condenser is not allowed to allocate memory dynamically. The space requirements for the condenser must be small and known in advance. For SSAO, a buffer of a fixed size is allocated for the occluder information. It does not run full because old, far away occluders are removed from time to time.

The algorithm is well-suited for highly parallel architectures like graphics processors: The iterations of the two outer loops can all be executed simultaneously. However, there

is a hidden problem in the assignment of `output`. Algorithm 1 contains all steps of Line Sweeping up to including the generation of the *sweep data* (see section 4.1), but not the *accumulation* into the *overall output*. We tackle this problem in the following section.

4.3 Data Accumulation

Algorithm 1 is missing an important piece, namely the accumulation of sweep data. For a single-threaded implementation (e.g. on a CPU), the solution is trivial: As soon as data becomes available, apply its impact on the overall output to the output buffer. With a multi-threaded implementation (e.g. on a multi-core CPU), locking of the output buffer or a transaction queue could be used. If the algorithm is implemented on a GPU (i.e. a many-thread architecture), such solutions become infeasible: We have thousands of compute cores, each generating a data point every few GPU cycles (i.e. nanoseconds on recent GPU hardware). We can identify two feasible methods of dealing with this situation.

4.3.1 Direct Accumulation

In the direct accumulation mode, the sweep threads directly write to the output buffer. As we want to execute these threads in parallel, synchronisation is mandatory. OpenCL does not support many thread synchronisation methods. It only supports a fixed set of atomic operations, and only on integer data types. If the desired data accumulation method can be achieved by the available operations on integers (i.e. simple operations like addition, minimum etc.), this approach may be used.

In the context of the above algorithm notation, using `atomic_op` as a placeholder for the actual operation to perform, direct accumulation can be written as follows:

Algorithm 2 Direct Data Accumulation

```
1  Output::operator[] (position, direction, data):
2      atomic_op(output_buffer[position], data);
```

Its main advantage is, that neither an additional computation step nor additional memory is required. The disadvantages are that atomic operations impose a synchronisation overhead and that not all accumulation functions may be reducible to the few atomic operations and data types available in OpenCL. For example, direct accumulation cannot be used to create a high-dynamic-range output because OpenCL does not support atomic operations on floating-point numbers.

4.3.2 Two-Pass Accumulation

We can solve the synchronisation issues (thereby lifting the requirement of atomic operations) if every thread is assigned a fixed, private portion of the global GPU memory.

Storing information in private memory does not require synchronisation (see algorithm 3). In a second pass (see algorithm 4), we use one thread per position in the result data structure (e.g. every pixel of the frame buffer in SSAO) to accumulate the sweep data from all directions. Note that this thread setup differs from the one we use for the sweeping process, where we use one thread per *ray*.

Algorithm 3 Two-Pass Data Accumulation – Pass One

```
1  Output::operator[] (position, direction, data)
2      i = get_private_memory_location(position, direction);
3      buffer[i] = data;
```

Algorithm 4 Two-Pass Data Accumulation – Pass Two

```
1  for p in P:
2      data = new AccumulatorData(p);
3      for d⃗ in D:
4          v = get_data(p, d⃗);
5          data.accumulate(v);
6      output_buffer[p] = data;
```

In algorithm 4, the outer `for`-loop is easily parallelisable. The inner loop is not, which is not a problem because the number of sweep directions is low.[14]

This approach leads to a massive increase in the required GPU memory: The entire the sweep data must be buffered. For problems in higher dimensions, this problem increases exponentially. The memory requirements can be alleviated by introducing some sequentiality: At first, we only gather sweep data for some of the sweep directions. We accumulate the results to the output buffer. Then we sweep further directions, accumulate their results and apply them to the output buffer. This process is repeated until all sweep directions are processed. Employing this method allows us to arbitrarily divide the memory requirements for the line sweep data at the cost of reducing parallelism and therefore increasing the run time.

Depending on the problem space, it can be challenging to come up with a useful memory layout for the line sweep data. The second pass (accumulation) must be able to quickly identify which portions of the line sweep data contribute to a specific point in the problem space. Iterating over all rays to identify whether they intersect the respective output position is not an option[15]. This means the rays' paths must be sufficiently predictable to quickly find all rays intersecting a specific point in the problem space, i.e. the `get_data` function from algorithm 4 must be fast. We will go into detail about this in section 5.7.2.

[14] Especially in two-pass accumulation mode, this number is strongly bounded by the GPU memory.
[15] for performance reasons

Chapter 5

The Line Sweeping Framework

Line Sweeping may be used to solve a variety of problems. However, there is significant programming involved in implementing a Line Sweeping algorithm, especially, if it is supposed to run on a graphics processing unit (GPU) for increased performance and decreased energy consumption. In this chapter, we try to lessen this burden by creating a Line Sweeping Framework which removes the repetitive programming work and requires the user to only provide the condensation and accumulation logic particular to their target program.

5.1 Technology

The Line Sweeping Framework must be versatile in its application, and yet supportive in function. It must leverage the GPU, build on proven technologies, and run on a broad variety of hardware. In this section, we explore the technological choices of the framework, mostly concerning programming languages.

5.1.1 Host Code

Every program leveraging the power of the GPU must be set up and started from the CPU. The portion of the code running on the CPU is also called "host code".

Every project should be able to use the Line Sweeping framework. C-Bindings are very common to interoperate between languages. Furthermore, languages which require a run-time environment are ruled out, as the run-time environment may not be present everywhere. The introduction of a garbage collector (GC) is also troubling because a real-time application without a GC may not want to include one because of *collection stalls*[16]. There are two common languages[EE16] which run without a run-time environment and a garbage collector while providing C-Bindings: C and C++. Both are formally specified by the "International Organization for Standardization" (ISO) and there are mostly-conforming and well-tested compilers for major platforms. C lacks some features greatly supporting programming convenience (e.g. classes with methods) and

[16] Unpredictable pauses in the program execution to free unused memory

error avoidance (e.g. generics), so we will use C++. The Rust programming language has also been considered but it is not as widely used yet and the ecosystem still lacks stability.

5.1.2 Device Code

Device code is the source code which will run on the parallel device. This is a graphics card in most cases, but can also be a CPU or a dedicated acceleration device.

There are very few programming languages capable of being compiled for a GPU. The only common general purpose languages are Vulkan, CUDA, and OpenCL. The former is still very young and prone to changes. The latter two systems are mature, well-tested, and proven backwards-compatible. CUDA is restricted to running on the GPUs of one particular manufacturer, while OpenCL runs on a wide variety of hardware. Consequently, the framework uses OpenCL.

5.1.3 Generating the Device Code

OpenCL code is usually compiled at the runtime of the host program. This is necessary because the compiled OpenCL program is specific to the device. A pleasant side-effect is that some values, which are variables in the host program but are constant throughout the execution of the device code, can be put into the source code as constants, rather than being set as a runtime parameter. The device program source code with the inserted constants will be passed to the OpenCL compiler during the runtime of the host program.

OpenCL has an integrated preprocessor comparable to that of C[17]. It does not suffice to provide a modern, feature-rich OpenCL preprocessing environment (e.g. it does not support classes and has no standard library with a rich support for string operations). Some projects have OpenCL source hard-coded in a C++ header file or load it from a text file. A more sophisticated technique is to use a specialised string builder. This dictates writing OpenCL code in C++ code by using specific function calls for every aspect of an OpenCL program's syntax tree.

A more pleasant approach to programming OpenCL is to directly write OpenCL code to a file, while still being able to insert host constants into the source, making static decisions about execution paths etc. This is exactly what template processors enable. Most template processors are tailored to be used in web servers, but they can be used to preprocess any code (not just HTML, JavaScript and CSS). The *PHP: Hypertext Preprocessor* is widely used, well-tested, easy to use, and adapts well to preprocessing OpenCL code instead of HTML. For a comprehensive assembly of the PHP part of the framework, see section 5.6.

[17] "The preprocessing directives defined by the C99 specification are supported." – [Gro09]

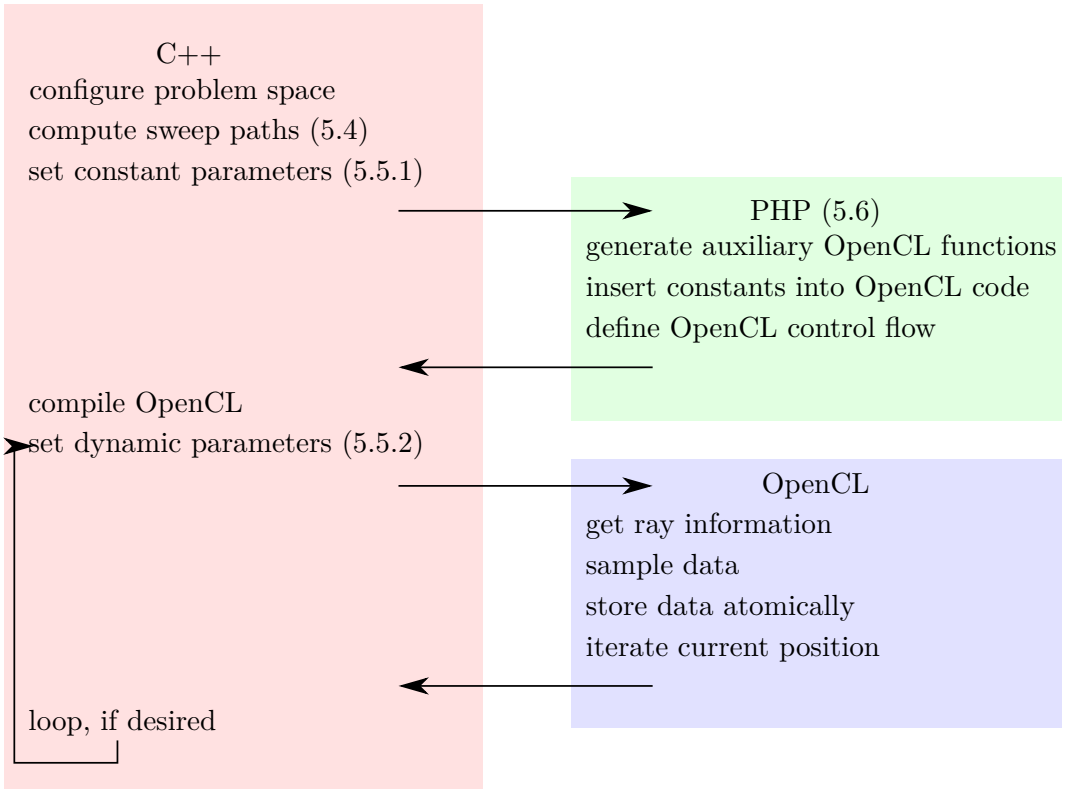


Figure 5.1: Execution flow overview in single-pass mode

5.2 Control Flow

The control flow is in principle up to the user of the framework because this setup provides the most flexibility and best interoperability with third-party software libraries. Yet it is obligatory to know what initialisation has to be made and how to interact with the framework once it is initialised.

The framework consists of code written in C++, OpenCL, and PHP. The control flow begins on the C++ side (see figure 5.1), where a user of the framework sets up the environment (e.g. a graphical user interface), loads resources and configures the framework. Subsequently, the PHP part of the framework is used to preprocess OpenCL code, allowing for the insertion of constants, adjustment of the execution flow, and increased convenience in programming for graphics cards. The resulting plain OpenCL code is compiled and run as often as needed, e.g. once, or for every frame. If the two-pass accumulation mode (introduced in section 4.3.2) is enabled, after each execution of the sweep kernel, the data accumulation has to run (see figure 5.2).

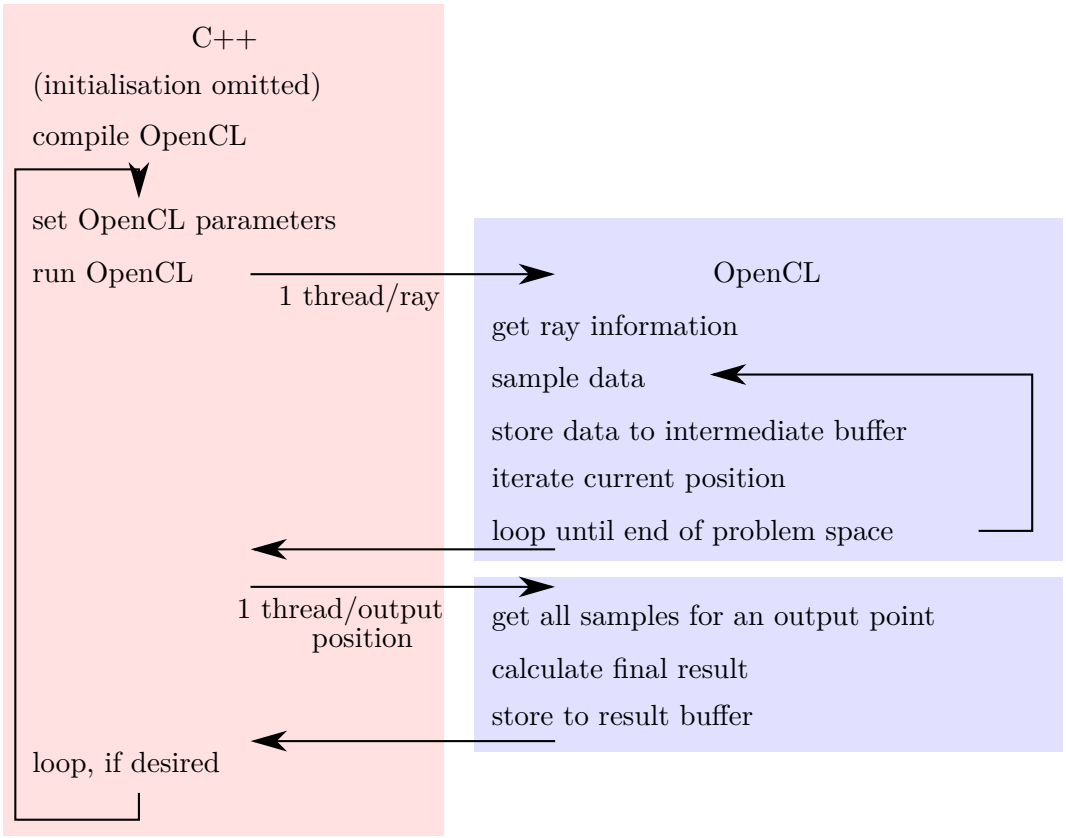


Figure 5.2: Execution flow overview in two-pass mode

5.3 Defining the Problem Space

In modern computing, data is often defined over rectangular grids (e.g. images, volume density maps). Therefore, the framework provides means to easily define a (hyper-)rectangular problem space (see algorithms 5 and 7).

Algorithm 5 Defining a Rectangular Cuboid Problem Space

```
1  const int DIMENSION = 3;
2  const cl_float sizes[] = {1024, 512, 512};
3  Rectangle<cl_float,DIMENSION> problem_space(sizes);
```

For special situations, other shapes might be more appropriate, including all relevant points, but leaving a smaller area to sweep than the over-approximation a rectangle would impose. Using an over-approximation of the problem space wastes GPU time and memory. For these cases, the problem space can be defined by providing a custom function, which intersects it with a ray, as well as giving a bounding diameter (see algorithm 6).

Algorithm 6 Defining an Irregular Problem Space

```
1  class SpecificSpace {
2  public:
3      void intersect(const Num_t* origin, const Num_t* direction,
4          Num_t* out_lambda_entry, Num_t* out_lambda_exit);
5      Num_t bounding_diameter();
6  }
```

The array lengths of `origin` and `direction` equal to the dimensionality of the problem space. `Num_t` is either `cl_float` or `cl_double`, depending on the framework configuration. It is also possible to write precision-agnostic code by using a template parameter for the number type as the `Rectangle` class does. Explicitly inheriting from an interface class using virtual methods is not necessary as a template parameter is used to pass the custom space definition to the framework.

5.4 Computing Sweep Paths

We established that Line Sweeping moves bulks of threads along rays across a problem space. The patterns in which those threads move through the problem space are relevant to the end result and the feasibility of two-pass data accumulation. In this section we examine how to carefully arrange the sweep paths, evenly sampling the problem space in a manner that enables two-pass sweeping. There are many ways to sample an n-dimensional space. The most common one is the distribution of the samples in a uniform n-dimensional grid.

In the case of Line Sweeping, this means laying a uniform grid of sampling points over the problem space, for each sweep direction. These grids subsequently define the path the rays take across the problem space. There may be useful sampling patterns, which do not offer coherent sweep directions, i.e. there may be no two sweep rays parallel to each other. Finding memory locations of samples in the two-pass accumulation mode is very difficult with chaotic sweep patterns. The framework supports two-pass accumulation only for sweep patterns employing *coherent* sweep directions (i.e. each sweep direction covers the whole problem space with parallel rays).

For the default sampling algorithm, we select a number of directions well-spread in the space of possible directions (see 5.4.1), and for each layout a uniform grid of samples over the problem space. This produces coherent ray bundles and simplifies data retrieval in the two-pass accumulation setup.

For an example on how to use the framework in this regard see algorithm 7, line 6. The default ray generator needs the desired data type (i.e. `cl_float` or `cl_double`), the problem dimensionality (in this case two), a definition of the problem space, a set of directions, the desired grid size, and the mode of accumulation (i.e. `TWO_PASS` or `SINGLE_PASS`).

Algorithm 7 LSAO Ray Generation Code

```
1  PointSpreader<cl_float, 2> direction_finder(direction_count);
2  direction_finder.optimise(0.001f);
3  direction_finder.jitter();
4  cl_float problem_space_dimensions[] =  imgwidth, imgheight ;
5  RectangleND<cl_float, 2> problem_space(problem_space_dimensions);
6  RayFactory::RayData *ray_data = RayFactory::default_rays<cl_float, 2>
7        (problem_space, direction_finder.points, grid_size, TWO_PASS);
```

5.4.1 Finding Directions

For Line Sweeping, we need a number of sweep directions. This number usually correlates with the quality of the final result as it is the number of direction samples that will be available at each point in space to compute the final result for that point. The main difference between Line Sweeping and other ray tracing methods is that the directions for all calculations are the same at each point in space for Line Sweeping, while they can be different (e.g. chosen randomly at each point) using other methods. This encourages finding a good set of directions. Random directions are infeasible because under-representing a specific direction can lead to biased results[18]. Stratified random sampling is a well-tried technique in sampling[Coo86].

The strategy is to first find a sampling that is as regular as possible (which is not

[18] e.g. in LSAO this could lead to the impression of directional environment lighting when it is not supposed to do so

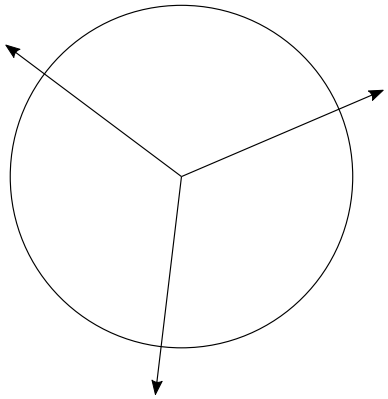


Figure 5.3: three directions evenly spread in two dimensions; note the lack of axis alignment

trivial on the surface of an n-sphere) and then jittering it. Other methods with adequate blue noise properties (e.g. Fast Poisson-Disk Sampling[Bri07]), could also be adapted to this problem and should work equally well. For an example of the respective framework functions, see the first three lines of algorithm 7.

N-Dimensional Spherical Stratified Random Sampling

We want to choose a specific number of directions in a way that they point in utterly different directions. We start out with the desired number of directions (vectors) being randomly chosen and normalised. Then we interpret these vectors as points on the unit n-sphere and assign a mutually repelling force to them while enforcing the constraint of staying on the unit sphere. Executing this simple physics simulation converges to a ray setup where all points are well-spread on the n-sphere (for an example see figure 5.3). This approach easily scales to any dimension and its spreading properties are an explicit part of the process (i.e. the repelling force). Performance-wise there is room for improvement. However, performance is of no concern here as this is executed only once in the initialisation phase of the framework. Furthermore, the number of directions is small because having many directions imposes a large computational burden during the actual Line Sweeping process.

To introduce jitter we can now apply a disturbance to each direction. The maximum offset is chosen to be half the distance between two neighbouring points. So the points cannot switch places but stay in their part of the n-sphere. This algorithm achieves a stratified random sampling of vectors in an n-dimensional space. See figure 5.4 for an example.

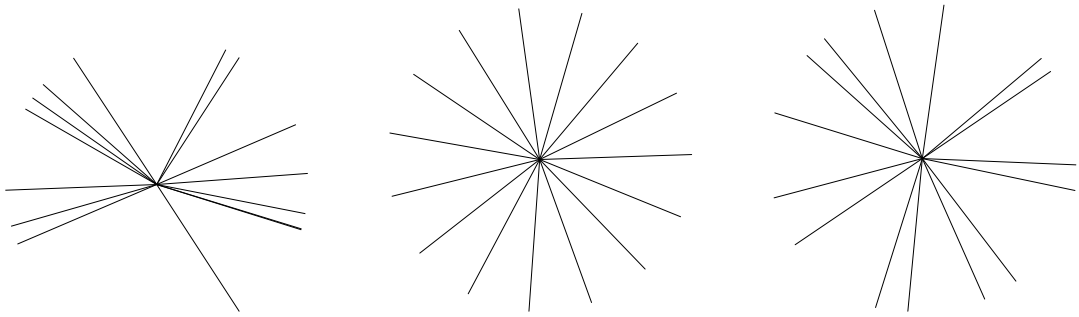


Figure 5.4: Sampling: 15 directions; from left to right: random, straight, stratified random. Note the two nearly identical directions and two large gaps in the random scenario.

5.4.2 Jitter

Stratified random sampling helps greatly in improving ray tracing results[Mit96]. This technique modifies a regular sampling scheme in a way that it moves all sampling positions by a small bit (i.e. the samples are *jittered*) In this section, we will explore to what extent it is applicable to Line Sweeping.

The sweep directions can be chosen arbitrarily and thus can be easily jittered, without influencing any part of the framework. This functionality is provided by the framework via the `jitter` method of the C++ class `PointSpreader` (see the line three of algorithm 7 for an example).

For single-pass Line Sweeping, individual samples may be jittered, or even completely offset. Going to two-pass Line Sweeping, the problem arises, that the accumulation algorithm (i.e. the second pass) needs a method to inverse the jitter (respective the offset) efficiently. For any given point in the problem space, it needs to be efficiently determinable which samples from each sweep direction need to be considered and where they reside in the GPU memory.

Assuming a small jittering of the samples by a repeatable pseudo-random process and a small problem dimensionality, the accumulator may still find the correct sample quite efficiently: Candidate samples are chosen according to the straight (i.e. not jittered) grid. Then the jitter is reapplied, and whichever of those candidate samples is closest then, is the overall closest sample. Note that the number of candidate samples to consider increases as the maximum jitter radius increases. This method is not part of the framework as the accumulation pass is generally the responsibility of the user. It would require too many assumptions about the desired output data structure and thus be a limiting factor to the framework's versatility. The framework does, however, provide auxiliary OpenCL functions to assist in finding the (candidate) sample(s) for a point in the problem space (see section 5.7.2).

5.5 Sweep Parameters

The OpenCL program needs input from the host code concerning input and output buffers, sweep paths, parameters to the implemented algorithm (e.g. a falloff radius in the case of LSAO) etc. In this section, we will discuss how to get this data from the host to the device.

5.5.1 Static Parameters

Static parameters are variables which are constant throughout the lifetime of the compiled OpenCL program. They are passed to the framework in the OpenCL preprocessing step, such that the PHP engine can directly paste them into the OpenCL source code. The framework internally uses the widespread *JSON*[19] data format to pass the values to the PHP interpreter.

Algorithm 8 LSDO Code for Static Parameters

```
1  JSONBuilder constants;
2  constants.add("image_width", (long)imgwidth)
3        ->add("image_height", (long)imgheight)
4        ->add("horizon_angles_around", (long)horizon_angles_around)
5        ->add("horizon_angles_up", (long)horizon_angles_up)
6        ->add("hull_buffer_size", 256)
7        ->add("falloff_radius", 60);
```

Definition of static parameters uses the `JSONBuilder` class from the framework (see algorithm 8 for an example). Static parameters can be accessed in the PHP context via the framework-defined function `A`[20]. `HullEntry hull[<?=A("hull_buffer_size")?>];` expands to `HullEntry hull[256];`, assuming the definition from algorithm 8. `<?=...?>` is the PHP short syntax for "insert the value of ... into the output". It is equivalent to `<?php echo ...;?>`.

5.5.2 Dynamic Parameters

Dynamic Parameters are values which may change with each execution of the OpenCL program. They are the inputs and outputs of the OpenCL code. To allow for maximum flexibility the user sets the parameters with the usual OpenCL library calls.

5.5.3 Struct Transfer

When defining the same struct in C++ and OpenCL, the memory may be laid out differently. This problem can be solved by fine-tuning the layout of each struct, inserting

[19] JavaScript Object Notation, [Cro02]
[20] Its use is so common that a short name is mandatory.

manual padding as needed. A less cumbersome variant is using the *boost compute* library[13]. Being entirely template-based, it also has its limitations. For example, struct definitions cannot be split up between different C++ files.

The framework provides the `CLStructDef` class. Instances represent a struct definition and can be used to transfer structured data[21] between the host and the device. The sweep layout is transferred using this method, but simple algorithms may not have to directly interact with it. In fact, none of the presented algorithms require it directly, as their inputs and outputs are unstructured buffers. An example where it may be needed is in the case of ray tracing using a *bounding volume hierarchy*[22] on the GPU. The bounding volume hierarchy is structured data and may be generated by the CPU, but will be used by the GPU.

A `CLStructDef` can be created and customised at the run-time of the C++ program. Once their definition is complete, the class calculates an efficient, device-compatible memory layout and generates an OpenCL struct definition for it. It has the following caveats: The CPU and GPU endianness must match (although this could be overcome with some adjustments). Handling `void` pointers is required on the C++ side because struct offsets are calculated at runtime.

Under the circumstances of being part of a framework, the benefits outweigh those, namely: Structs can be defined in parts, i.e. the framework can add some basic members and the user can add custom members. Members are reordered for minimal padding, possibly saving a considerable amount of memory and bandwidth usage.

5.6 OpenCL Preprocessor

Writing raw OpenCL code has some limitations. It is difficult to write type-generic code. OpenCL does not provide a simple way to include constants which are only known at the OpenCL compile time (i.e. C++ runtime). There is a built-in way for compile-time execution path alteration (`#ifdef`), but it does not support loops (e.g. to run some code for each element of a vector).

In employing the *PHP: Hypertext Preprocessor* some degree of programming comfort as well as flexibility is achievable. The standard library of PHP is tailored towards the manipulation of HTML documents. There are general string and advanced control flow manipulation features available, but no features specifically for OpenCL code generation or manipulation. Functions specific to Line Sweeping are also inherently missing. So the framework must add such functions.

5.6.1 Vector Types in OpenCL

Line sweeping requires linear algebra computations. The framework supports Euclidean spaces of arbitrary dimensions. As we will see now, OpenCL is not immediately ready for the task and needs some workarounds to achieve this.

[21] everything beyond an array of values of the same type
[22] a tree data structure used to accelerate ray tracing

In OpenCL, there are types for vectors of fixed sizes (e.g. float4, double2, uint16). Commonly used functions and operators are provided for the built-in vector types (e.g. the dot product, vector normalisation, vector addition; see algorithm 9 for example code). The framework allows for the usage of vectors of dimensions for which there are no built-in types in OpenCL. For those, fixed-size arrays of the base type are used (e.g. `float[5]`). The previously mentioned commonly used functions provided by OpenCL do not apply to those (e.g. there is no function to add two vectors of type `float[3]`). We provide such functionality using the PHP processor. This is done by requesting a specific function from the framework by means of a PHP function call (see algorithm 10). The framework will, in turn, replace this by the respective OpenCL function call and prepend the generated function definition to the OpenCL code (see algorithm 11). We rely on the OpenCL compiler to optimise this code (e.g. by inlining the generated function and by applying the SIMD instructions of the target device).

Algorithm 9 Addition for Built-in Vector Types in OpenCL

```
1  float2 a = ...;
2  float2 b = ...;
3  float2 c = a + b;
```

Algorithm 10 Requesting a Vector Addition Function for Non-built-in Types

```
1  float a[3];
2  float b[3];
3  ... // initialisation of a and b
4  float c[3];
5  <?=add("float[3]","a","float[3]","b","float[3]","c")?>;
```

Algorithm 11 Expansion of the Request by the Preprocessor

```
1  add_float_3_float_3_float_3(float* p1, float* p2, float* r) {
2    r[0] = p1[0] + p2[0];
3    r[1] = p1[1] + p2[1];
4    r[2] = p1[2] + p2[2];
5  }
6  ...
7  float a[3] = ...;
8  float b[3] = ...;
9  float c[3];
10 add_float_3_float_3_float_3(a,b,c);
```

It is automatically recognised, whether the result must be rounded, i.e. if there are floating point operands, but the result type is integer. Rounding operations are inserted as necessary. The provided functions do not only work for array types, but also for the OpenCL-integrated types. This facilitates the writing of type-agnostic code, which is required for writing dimension-agnostic code. If memory from the `__global` section of the device is involved, it must be reflected in the OpenCL type signature of the function. To mark a parameter being resident there, `#G` must be prepended to its type name.

5.6.2 Reference

The following is a comprehensive list of the functionality provided by the framework on the OpenCL preprocessing level, concerning the support of vector data types.

Binary Operations All binary operation requests have the same structure:
`operation(type1, parameter1, type2, parameter2, out type, out parameter)`

Addition `add`

Subtraction `sub`

Element-wise Multiplication `mul`

Element-wise Division `div`

Dot Product `dot`

Unary Operations Unary operation requests have this structure:
`operation(input type, input parameter, out type, out parameter)`

Vector Length Calculation `length`

Assignment `assign` This can also be thought of as a copy operation. It also doubles as type conversion tool, if the given out type and input type differ.

Vector Normalisation `norm`

Constants The following constants are set in the PHP environment.

`$ST` The scalar floating point type corresponding to the used precision. This is either "float" or "double".

`$VT` An instance of a subclass of the PHP class `SimpleType` corresponding to the type used for positions and directions. This is not a simple string because of a quirk in C (and thus OpenCL) syntax. Variable declaration syntax depends on the type of that variable. Consider this code: `float4 x;`. This works for scalar types and for built-in vector types. But for fixed-size arrays, it looks like this: `float x[4];`. Note that the length comes *after* the variable name. For this purpose the `$VT` instance has these two parameterless methods

`decl0` Returns the part of the declaration before the type name.

`decl1` Returns the part of the declaration after the type name (i.e. the empty string for scalar types and built-in vector types).

Furthermore, it provides these methods

`access($i)` Returns a string to access the `$i`-th element of the vector, e.g. `".s2"` for the third element of a built-in vector type or `"[2]"` for an array type.

`arity` Returns the number of elements the vector has.

5.6.3 Usage

This section outlines the necessary steps to initialise and use the OpenCL preprocessor of the framework. Prior knowledge of PHP is useful but not required.

The PHP-enriched OpenCL part of the framework must be included like any PHP library: `<?php require_once("linesweep.php"); ?>` And at the end of the file, the framework must be activated by this line of code: `<?php footer(); ?>` Only between these lines, the functions of the framework (e.g. algebraic functions) can be used. Technically this is required for the framework to forge the complete OpenCL code into a string (using PHP's "output buffering"), while recording the requests for algebraic functions. This enables the generation of those functions and prepend them to the code where they are used. OpenCL (like C++) requires functions to be declared before their first usage in the source code.

The framework will also insert the declarations of the `RayDescriptor` and `DirectionDescriptor` structs, which describe the memory layout of the sweep path descriptions. A programmer may use the following functions to benefit from the Line Sweeping capabilities of the framework:

`init_framework` This function initialises the Line Sweeping process. It finds the ray path to walk corresponding to the respective execution thread id, and copies the ray's initial information (e.g. start position in the problem space) into the provided `FrameworkState` struct.

`sweep_step` This function advances to the next step on the ray's sweep path, which includes updating the current position and output memory location.

There are also functions for gathering sweep data in the accumulation pass of the two-pass operation mode. They are presented in section 5.7.2.

5.7 Accumulating Data

The final step of Line Sweeping is data accumulation, as discussed in section 4.3. In this section, we will explore how the framework facilitates this process.

5.7.1 Direct Accumulation

For direct accumulation, the OpenCL sweep code can directly modify the output buffer(s) atomically. This lies in the responsibility of the user of the framework. The framework does not interfere or help in this case as regular OpenCL atomic operations can be employed in a usual manner.

5.7.2 Two-Pass Accumulation

For two-pass accumulation, the framework provides means to find the right place in memory to store and read intermediate sweep results. Each ray has the information of its memory begin address and increments the offset at every sampling point it passes. For the accumulation pass, the situation is slightly more involved: The memory location must be found using only the position of the output point (e.g. pixel, voxel).

Base vectors are stored on the GPU for every sweep direction. They are the vectors, which define how the sampling grid for that direction is laid out. As a first step, the output point position is transformed into the base of the respective sampling grid[23]. Removing the sweep direction component, we get coordinates of a coordinate system suited to identify the ray which hit (or came closest to) the output point. Next, these coordinates are linearised, using the maximum diameter information from the problem space definition (see 5.3). For these linearised coordinates, there is a lookup table in GPU memory mapping them to ray descriptors. In the ray descriptor, we find the position of the private memory for that particular ray. The addition of the previously omitted component of the transformed position vector (i.e. the number of steps in the sweep direction) then yields the memory position.

Not having to implement this is a major benefit of using this Line Sweeping framework. The following functions of the framework can be used to find memory locations in the context of OpenCL code (see algorithm 12 for an example).

Algorithm 12 Excerpt from the LSAO Accumulation Code

```
1 for (uint i=start_direction; i < end_direction; i++) {
2     ___global IntermediateStorage*is_ptr = get_intermediate_storage(sweep_data,
3         rays, directions, ray_lookup_table, posvec, i, ray_interval);
4     if (!is_ptr || is_ptr->v.w == 0) {
5         continue;
6     }
7     accumulator += convert_int4(is_ptr->v);
8     K++;
9 }
```

[23] All directions are iterated.

`get_intermediate_storage_projection` This function projects the position on the sweep grid's base vectors, which in turn can be used with the following function.

`get_storage_preprojected` This function calculates the memory location of a position already projected into the base of a specific sweep direction. Applying ceiling or flooring to individual members of the projected position vector allows receiving neighbouring memory locations, facilitating the implementation of linear interpolation of sweep values.

`get_intermediate_storage` This is a combination of the above two functions without the possibility to modify the projected coordinates. It provides a convenient way to directly sample the sweep data using a nearest-neighbour approach. Usual artefacts observed due to nearest-neighbour sampling are attenuated by the fact that multiple unaligned grids are superimposed in the Line Sweeping process.

5.8 Sweep Separation

Intermediate storage space is needed in the two-pass operation mode. It depends on the size of the problem space, the spacing between samples, the number of sweep directions, and the amount of information to be stored for each sweep step. In extreme cases, it may be too large for the device memory. We can resolve the issue by dividing the sweep directions into smaller portions. For each portion, we sweep and then gather the results. This way we can reuse the intermediate storage for the next portion of sweep directions.

For this purpose, the framework provides the class `SplitRayDataGpu`. Saving and restoring the accumulation state between calls to the result gathering kernel lies in the responsibility of the framework user. For the Screen Space Directional Occlusion example code, the intermediate accumulation result fits in the output image buffer, which is allocated as "CL_MEM_READ_WRITE" instead of "CL_MEM_WRITE_ONLY" for this purpose.

Chapter 6

Diffusion Inpainting

Figure 6.1: The areas to be inpainted are marked with a checkerboard pattern.

Inpainting is a problem setting where a data set (e.g. an image) is only partly known and the missing parts must be plausibly supplemented (see figure 6.1). A well-tried method for inpainting of smooth regions is diffusion inpainting[Car88]. It works by promoting smoothness in the interpolated area while enforcing equality to the known data at its borders.

In this chapter, we will roughly examine the traditional rendering methods for diffusion inpainting, examine a ray tracing formulation of the problem and finally apply the Line Sweeping framework to it.

6.1 Mathematical Problem Setting

Smoothness in this context can be defined by this integral:

$$\int\int (\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2 \mathrm{d}x\mathrm{d}y \tag{6.1}$$

Producing a smooth solution requires the *minimisation* of this integral. Colour images are treated like multiple greyscale images – one for each colour channel.

6.1.1 Traditional Solution

The minimisation of the integral is traditionally implemented using the gradient-descent method. It starts with a guess of the solution (e.g. a constant colour in the most simple case). Then it uses the derivation of the optimality criterion function to estimate the direction in which the proposed solution has to be changed, in order to come closer to optimality. The proposed solution is updated accordingly and the process is iterated until some convergence criterion is satisfied (e.g. "after one million iterations" or "when the smoothness integral is smaller than X").

Applying this method to the optimisation of smoothness and two-dimensional, discrete images yields the following iterative formula[24]:

$$f_{i+1}(x,y) := (1-\tau)\cdot f_i(x,y) + \frac{\tau}{4}(f_i(x-1,y)+f_1(x+1,y)+f_1(x,y-1)+f_1(x,y+1)) \tag{6.2}$$

τ is the "step size". Its careful choice has a great impact on the stability and performance of the algorithm. See [GWB10] for the state-of-the-art in choosing it appropriately. *Multi-grid algorithms*[Bra77] may also be employed. For a recent overview over different techniques see [Wei+16]

6.2 Ray Tracing Formulation

It has been shown that diffusion inpainting can also be formulated as a ray tracing problem[BLW11]. In this approach, the area to be inpainted is regarded as a diffuse reflector. The defined areas are treated as solid light sources[25]. In a mathematical formulation, this is

$$I(p) = \int_0^{2\pi} L(r(p,\theta))w(|p - r(p,\theta)|)\mathrm{d}\theta \tag{6.3}$$

$I(p)$ is the total radiance reaching point p. $r(p,\theta)$ is the closest light source that is hit, when a ray is shot from point p in direction θ. $L(p)$ is the light emitted by the light source at point p. $w(d)$ is a weighting function implementing a brightness falloff for distant light sources.

[24] See [Car88] for a detailed derivation.

[25] They emit light and block light that would pass through them.

The implementation of this formula (as presented in [BLW11]) uses stochastic ray tracing for each unknown point to find $r(p, \theta)$. It is a direct translation of the integral to a `for`-loop.

6.3 Application of Line Sweeping

For all pixels in the unknown area, we need to know which light sources are visible. We need to know their colour and distance. This is a paragon for Line Sweeping.[26] Instead of tracing rays for each unknown pixel, we trace rays over the whole unknown space and push the relevant information to each pixel. The formula to calculate the light at each point stays the same. Just the computation of $r(p, \theta)$ is done differently (i.e. employing Line Sweeping).

6.4 Application of the Framework

We know that Line Sweeping is applicable to the problem at hand. Were it not for the Line Sweeping framework, we would now have to implement the whole Line Sweeping algorithm. Employing the framework, we only have to set up the framework and implement a *condenser* and an *accumulator*. For high-quality results, a large number of sweep directions is required. This would impose a high synchronisation overhead using *direct accumulation*. So we will command the framework to use *two-pass accumulation*.

Please note that the given implementation is not a complete implementation of diffusion curves (as presented in [Orz+08]). We assume information about the light sources to be stored in a regular grid already. For a complete implementation, we would either need a curve rasteriser or a two-dimensional GPU ray-curve intersection algorithm. Diffusion curves furthermore support expressive shaders in curve space and in image space. These *could* all be implemented on top of the given Line Sweeping inpainting implementation. However, it is beyond the scope of this thesis.

6.4.1 Information Condensation

This is trivial. We only need to store the last seen light source.

Algorithm 13 Condenser for Inpainting

```
1 def InpaintingCondenser::digest(a, d, fa):
2   if fa != NULL:
3     this.last_seen = fa;
4
5 def InpaintingCondenser::getInterestingData():
6   return this.last_seen;
```

[26] Yet, there are other ways to improve on it[PJS15].

Figure 6.2: Line Sweeping reconstruction of figure 6.1, using a grid size of 1 pixel and 128 sweep directions.

In algorithm 13, the parameters `a`, `d` and `fa` are defined as in algorithm 1. `a` is the current position and `d` the sweep direction. `fa` is $f(a)$, which is the information about the known regions in the case of inpainting. If `a` is to be inpainted, `fa` contains no information, which we define as `NULL` in the code.

6.4.2 Data Accumulation

The sweep data contains information about all visible light sources in all sweep directions. This information corresponds to the term $r(p, \theta)$ in equation 6.3. All that is left to do is evaluate the weighting function and sum up the result.

Algorithm 14 Accumulator for Inpainting

```
1  def InpaintingAccumulatorData::constructor(position):
2      this.position = position;
3      // initially no radiance
4      this.radiance = 0;
5
6  def InpaintingAccumulatorData::accumulate(light):
7      this.radiance += light.brightness * w(|this.position-light.position|);
```

Figure 6.3: Line Sweeping reconstruction of figure 6.1, using a grid size of 2 pixels and 32 sweep directions.

6.4.3 Results

Implementing inpainting using the ray tracing formulation from [BLW11] is a simple task employing the Line Sweeping framework. The resulting Line Sweeping inpainting program can utilise the performance of any device supported by OpenCL.

The size of the image to be inpainted is 1155x866 pixels. For figure 6.2 we use 128 sweep directions at a grid size of 1 pixel. This high-quality result takes 3,969 ms to compute. A result with a lower quality (figure 6.3), using 32 sweep directions at a grid size of 2 pixels, can be obtained in 188 ms.

Chapter 7

Inside-Outside Testing

The objective in inside-outside testing is to find out if a point is inside of an object or not. We will specifically look into the three-dimensional version of this problem with the "object" being defined by a triangle mesh. Furthermore, we will not only test one point, but we will generate a whole voxel grid and decide for each point if it lies inside.

Furthermore, we will assume that the triangles are not oriented (i.e. their normal vector does not necessarily point outside of the object) and we will allow for meshes which are not water tight[27]. For the openings in the mesh, we want the algorithm to appropriately close the mesh.

7.1 Ray Tracing Formulation

There is an approach for this based purely on ray tracing[Tak+14]. It uses the information about a point being inside or outside of the object for *facet orientation correction*[28]. For our purpose the pure information about being inside or outside suffices.

The basic idea is that for every point we trace some rays and count how many times we intersect with the object before we reach infinity. If this number is even (e.g. 0 if we miss completely, or 2 if we enter the object and then exit it again), the point is outside of the object. If the number is odd, the point is inside the object (see figure 7.1). The openness of the meshes is a problem: A point could be inside of the object, but one of its rays could still have no intersections with the object if it points through a gap in the mesh. However, if we trace many rays and assume that the gaps in the mesh do not dominate, most of the rays will get the answer right (see figure 7.2). We implement a democratic voting system for rays to decide if a point is inside or outside of a triangular mesh, using Line Sweeping.

[27] i.e. there may be small openings

[28] In computer graphics, the plane normals of triangles are often oriented and supposed to point to the outside of an object. Facet orientation correction repairs meshes which violate this constraint.

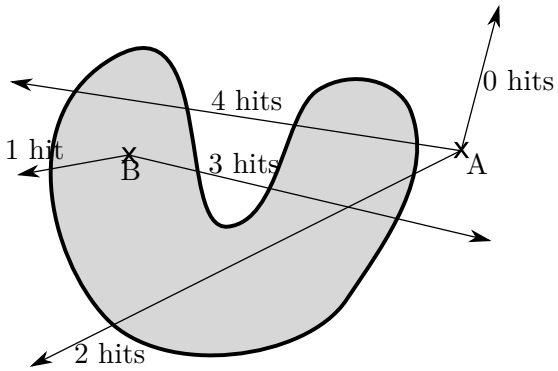


Figure 7.1: Point A is outside of the shape, so all its rays have an even number of intersections with the shape. Point B is inside, so all rays have an odd number of intersections.

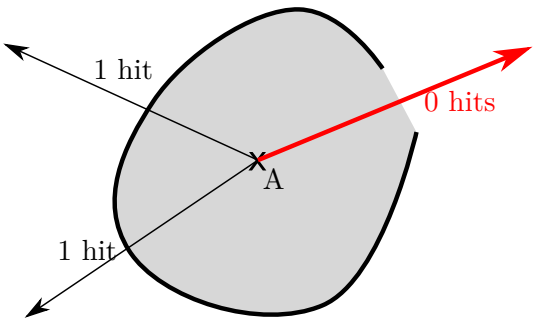


Figure 7.2: The shape is not closed, so *some* test rays have an unpredictable intersection count.

7.2 Application of Line Sweeping

In Line Sweeping, rays do not start at some arbitrary point (see figure 7.3). They always start at borders of the problem space (i.e. an axis-aligned bounding box of the mesh). As they start at a bounding box, they are always outside of the mesh at first. When they hit the mesh for the first time, they are inside. Whenever the mesh is intersected, the condenser negates its current guess about being inside of the mesh. At every passed point, the condenser casts its vote to the ballot buffer. Note again, that a few rays might well be off in their guess, as open meshes are allowed (see figure 7.4). The accumulation is adding the votes and calculating the winner for each voxel.

7.2.1 Intersecting the Mesh

The mesh is initially given as a list of triangles. This is a problem because we cannot trivially get a triangle for a position in the space. Instead, we employ ray-triangle intersection code. When a ray starts or hits a triangle, we calculate at which point we will hit the next triangle. There are many efficient ways to do this on a GPU[San+14][SP12]. We will assume, we have an efficient way to test this. The actual implementation will simply iterate over all triangles and do an intersection test[MT97] for each. This is feasible for simple meshes, as it is only done as often as the ray intersects the mesh.

7.3 Application of the Framework

As the output data can already quickly become quite large with *voxel* images, we choose to run the framework in the direct accumulation mode. For this to work, we need to find

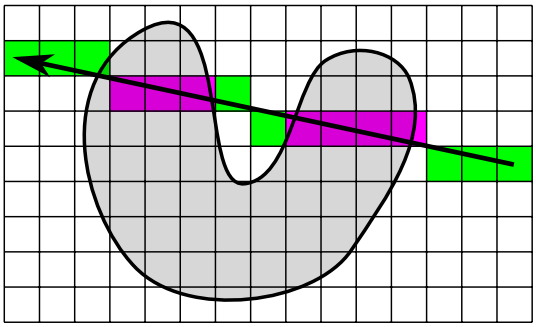


Figure 7.3: Line sweeping starts from a border of the interesting space and marks all passed voxels as inside (pink) or outside (green).

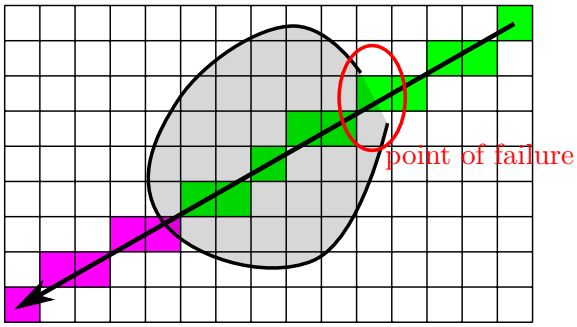


Figure 7.4: A ray pointing through a gap votes wrong on every single voxel, beginning at the marked "point of failure".

a data structure capable of holding votes together with an *atomic* OpenCL operation to cast a vote. The only data structures capable of atomic operations in OpenCL are integers. For performance and space efficiency reasons we will divide each 32-bit integer into two 16 bit counters. The first counter stores the number of overall votes and the second one stores the number of votes for being inside of the mesh. We define the "insideness" of a voxel to be the division of the second counter by the first. A voxel is to be considered inside of the mesh if its insideness is greater than 0.5.

7.3.1 Information Condensation

We need to remember if the current amount of intersections with the mesh is an even or odd number (one boolean, `this.inside` in algorithm 15). Furthermore, we need to calculate how many steps it will take until the next intersection with the mesh happens and count this number down on each step. When it reaches zero, we must update this number to the next intersection with the mesh and flip the state of `inside`. If there is no further intersection, `ray_mesh_intersection` shall return the maximum attainable value for the data type.

We must initialise the condenser such that it evaluates the distance to the next intersection in its first step and knows it is outside the mesh[29]. By setting the `steps_to_hit` to 1 and `inside` to true, we achieve this. The decreased counter will become zero in the first sweep step. This will trigger a recalculation of the steps and a flip of the `inside` variable to `false`.

Note that there is no $f(a)$ as a parameter to the digest method in the sense of algorithm 1. The only input data is the ray-triangle test, but it does not make sense to evaluate it after each step.

[29]We know that we are outside of the mesh at first because Line Sweeping always starts at the border of the space.

Algorithm 15 Condenser for Inside-Outside testing

```
1  def InsideCondenser::construct():
2   this.inside = true; // will be changed in the first digest call
3   this.steps_to_hit = 1;
4
5  def InsideCondenser::digest(a, d):
6   this.steps_to_hit = this.steps_to_hit - 1;
7   if this.steps_to_hit == 0:
8     this.steps_to_hit = ray_mesh_intersection(a,d);
9     this.inside = not this.inside;
10
11 def InsideCondenser::getInterestingData():
12  return this.inside;
```

7.3.2 Data Accumulation

For the direct accumulation mode, an accumulator is not required. We merely need to define the storage operation (as in algorithm 2). At every point, we need to cast the vote atomically (see algorithm 16).

Algorithm 16 Inside-Outside Data Accumulation

```
1  Output::operator[] (position, direction, inside):
2     v = 1«16;
3     if inside:
4        v += 1;
5     atomic_add(output_buffer[position], v);
```

7.4 Possible Improvements

As mentioned in section 7.2.1, an efficient GPU ray tracing data structure, as well as ray-triangle intersection method can be employed.

If the normals of the triangles can be trusted to always point outside of the geometry, accuracy can be greatly improved. If the best guess for a ray is, that it is inside the object, but the next intersection shows that it only *then* will *enter* the mesh, it knows that its current data is not reliable. So it can completely refrain from voting until its certainty is restored at the next intersection with the mesh.

At points where two triangles meet there can be numerical errors such that of two collisions with the triangles only one is recognised (see figure 7.5). This is due to the fact that collisions right after another collision are ruled out to prevent the algorithm from colliding with the same triangle over and over again. This could be tackled by allowing

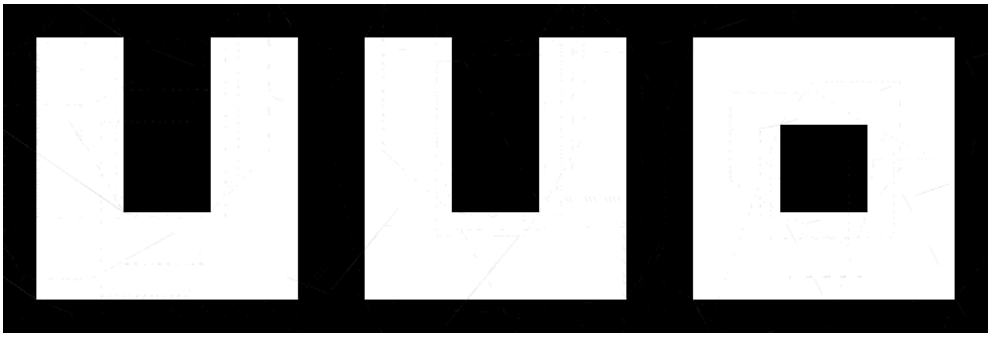


Figure 7.5: Three central cross sections through the voxel result of the non-convex shape. We use 12 sweep directions and a total of 2,348,486 rays. Note the slightly darker lines inside the object. They are artefacts of single rays mistaking two triangle intersections in a small area for one.

multiple collisions in a small distance, but also tracking the most recently intersected triangles and disallowing collisions with those.

The presented method also produces slight noise at object borders. This comes from the sweep rays not being aligned to voxel centres and can be solved by doing so (see section 8.1).

7.5 Results and Benchmark

We do not employ a bounding volume hierarchy to accelerate ray-triangle intersection tests. Therefore, models with many triangles are prohibitive. We use an octahedron (8 triangles) and a box with a hole in it (25 triangles). The latter is relevant because it is not convex. We will use the former to demonstrate the implementation on an open mesh. For this purpose *all* triangles making up the octahedron will be slightly moved away from the centre. The grey values of the shown images are the votes for "inside" divided by the number of total votes, scaled to the interval $[0, 255]$. For all examples, the output volume has a size of 256^3 voxels.

The algorithm performs well on closed meshes, even if they are non-convex (see figure 7.5). For non-closed meshes, the noise increases drastically (see figure 7.6). Regions with fewer gaps are less affected (see figure 7.7). The result is at some points too noisy for a smooth thresholding. A slight blurring of the voxel data can help. If a higher-quality solution is required, we can simply increase the number of rays (see figures 7.8, 7.9)

The results using the lower number of rays (i.e. 2,348,486; figures 7.5, 7.6) takes 1,078 ms to sweep and accumulate for the octahedron and 2,109 ms for the non-convex shape. This increase is not surprising because no ray tracing acceleration data structure is used. Sweeping the octahedron with the higher number of rays (i.e. 12,543,424; figure 7.7) takes 7,375 ms. The generation of 2,370,864 rays on the CPU takes 2,250 ms. For the generation of 12,543,424 rays the CPU needs 25.5 seconds.

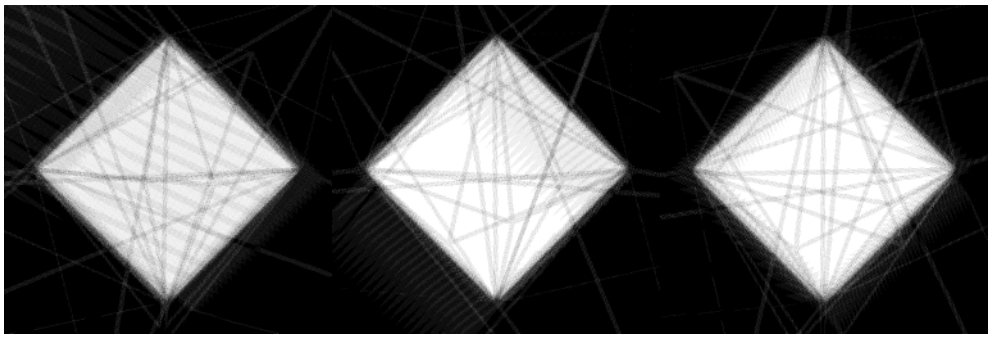


Figure 7.6: Three central cross sections through the octahedron with gaps between all triangles. We use the exact same rays as for figure 7.5. Note that *all* visible edges are open in this example. These cross sections depict the most noisy parts of the result.

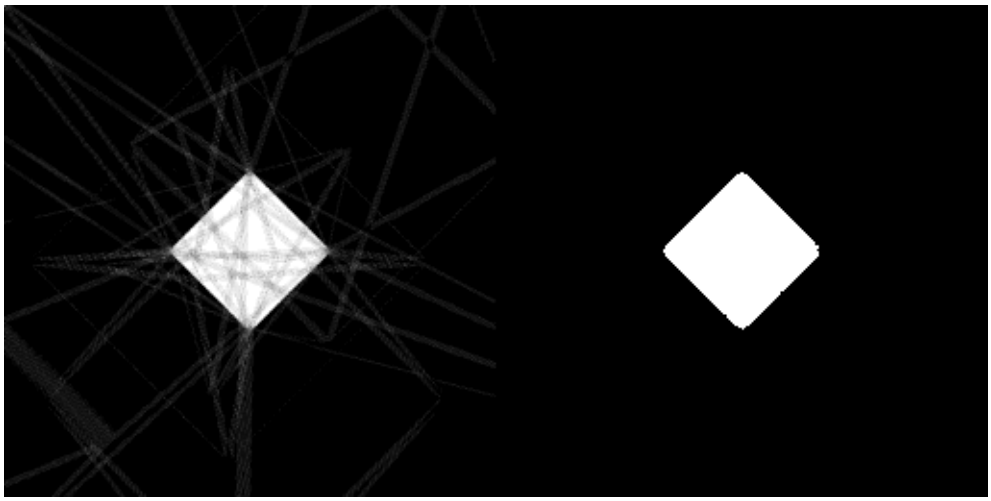


Figure 7.7: Non-central cross section through the octahedron voxel data from figure 7.6 (left), also with thresholding applied (right). Note the jagged edges which are a consequence of rays not being aligned to voxel centres. Also note the slight noise at the corners, which is where the gaps in the mesh are.

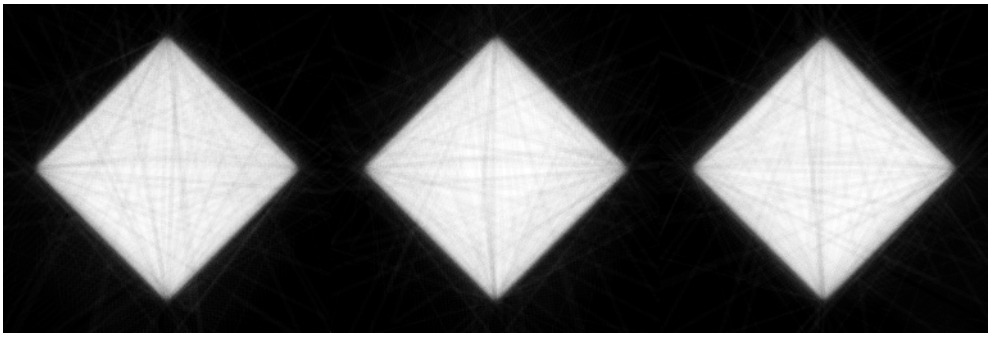


Figure 7.8: Three central cross sections through the octahedron with gaps between all triangles. This result uses 64 sweep directions and a total of 12,543,424 rays. Note the smoothness of the result even at gaps.

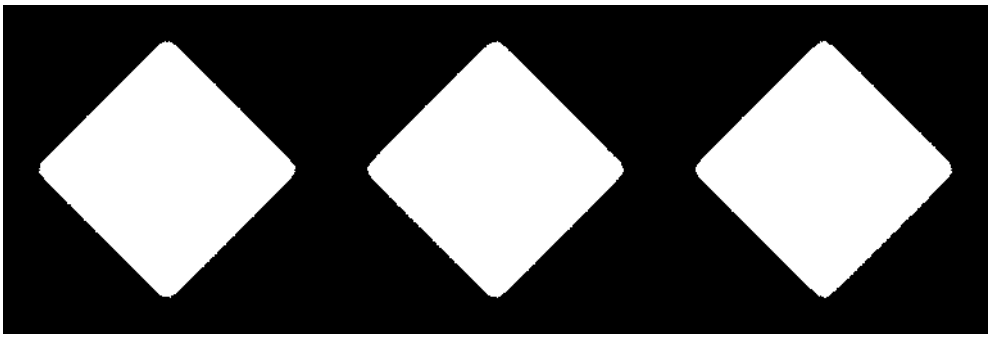


Figure 7.9: Thresholding of figure 7.8. Note the smoothly rounded corners at gaps in the mesh. Also note that the artefacts of improper ray alignment (i.e. jaggy lines) remain.

Chapter 8

Future Work

8.1 Performance Analysis and Optimisation

The optimisation of the huge amount of C++ and OpenCL code that is the presented Line Sweeping framework is beyond the scope of this thesis. Presented execution timings do by far not represent the maximum possible performance for Line Sweeping algorithms on a GPU. The following list explains some possible optimisations and outlines what steps must be taken to incorporate them into the context of the framework.

Snap-to-grid Sweep Patterns It is possible[Tim13] to generate sweep patterns in such a way, that sampling occurs only at pixel centres (see figure 8.1). This requires different step sizes for the different sweep directions, which is already supported by the framework. The only step left to do for this is writing a ray generator for this algorithm, complementing the `default_rays` generator. Expansion of the sampling scheme to higher dimensions is trivial and should be considered, as the framework is built to support them. There are reasons to use the `default_rays` generator (e.g. constant step size, user-defined number of sweep directions), so this feature should become an opt-in feature.

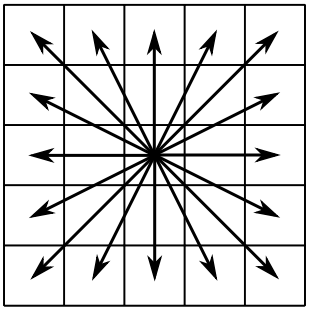


Figure 8.1: Example of the sweep sampling scheme from [Tim13] for two dimensions and a 5x5 grid

Sequential Writing In the current implementation of the framework, for two-pass accumulation, each GPU thread gets a private, *consecutive* memory region. This implies, that each thread, in each cycle writes to locations widely-spread in the GPU memory. For current GPU architectures, it is more performant if multiple threads[30] write to memory locations close to each other. One strategy to achieve this with Line Sweeping is shown in [TW10]. Inclusion in the framework is possible by shuffling the private thread memory such that blocks of memory are assigned to a number of threads (32 might be a well-performing number for current GPU architectures) at once. Instead of incrementing the memory location by 1 in the `sweep_step` function, it would be incremented by that number (e.g. 32). The inverse of that shuffling must be applied in the memory location lookup function for the second pass of two-pass accumulation. Note that this implementation wastes some GPU memory as it requires all threads in such a thread group to reserve the same amount of memory. The threads with a smaller requirement will thus be assigned more memory than they require.

This improvement can be added to the framework completely transparent to its user. The algorithm would *just run faster*.

Ray Generation The C++ code for generating the sweep rays in the first place is currently unoptimised. For two-dimensional sweeps, ray generation takes less than a second. In three dimensions, it may already take several seconds. For most use cases this is not a problem at all, as the rays are generated once at the beginning of the program ("loading time") and re-used for each call to the GPU Line Sweeping. They can even be stored to and loaded from disk, removing the generation time altogether (from an end user's perspective). However, if the problem topology changes quickly, faster ray generation may be worth investigating.

Currently, the ray generation algorithm is implemented in a single-threaded way. Multi-core CPUs may be leveraged to radically shorten the ray generation times. Note however that this is not trivial because the code in question is quite complex and for two-pass accumulation, global bookkeeping about reserved memory regions has to be done.

Overhead Analysis The creation of frameworks and programming libraries involves abstraction from the actual problem. Sometimes this produces an overhead in execution speed, memory requirements, dependencies or the size of the executable program. Some of these overheads are obvious and expected (e.g. the dependency to PHP), while others may be interesting to analyse. This could be done by implementing a Line Sweeping algorithm once with the framework and once without it. A comparison of the implementations will reveal any overhead the framework may introduce.

Incremental Rendering The Line Sweeping framework already supports partial sequentialisation of the sweeps (see section 5.8). If the order of execution is chosen

[30] Especially those in the same "warp", i.e. sharing a program counter

such that the result is not too biased after any sweep part (i.e. each part sweep contains opposing sweep directions), the partial results may be used for a preview of the final result. Implementing this in the framework requires only an extension to the `PointSpreader` class to sort the sweep directions accordingly.

8.2 Specialisation

This thesis features several Line Sweeping algorithms. None of them have been implemented with great care for optimisation or features. Creating a full-featured, optimised version of each may be interesting. This is the only way, a fair performance comparison to state-of-the-art algorithms can be made. The missing features of each presented algorithm can be found in their respective chapter.

8.3 Facet Orientation Correction

In facet orientation correction, a (possibly not completely closed) triangle mesh is given. For each of the triangles, it must be calculated, which of the two possible surface normal directions points to the "outside" of the object.

From the robust[31] implementation of inside-outside testing (chapter 7), one could easily derive an implementation for facet orientation correction. In inside-outside testing, the sweep data (whether a position is inside or outside) is written to a buffer representing a voxel grid. Instead, we can assign this information to a data structure related to the triangles of the mesh. Then each triangle normal could be made to point in the direction that was most voted for. This may be an especially rewarding quest, as current state-of-the-art implementations still do all calculations on the CPU[Tak+14].

8.4 Volume Rendering

Rendering inhomogeneous volumes in ray tracing situations is a hot topic in computer graphics. Line Sweeping is very close to one of the most common methods to achieve it: Photon Mapping.

8.4.1 Photon Mapping

In Photon Mapping, for every light source, there are rays of light shot into the medium. They are then spread into the medium according to its properties and stored in a 3D data structure. The stored light information is passed to view rays intersecting it.

8.4.2 Similarities to Line Sweeping

To simulate diffuse lighting, one can assume many light sources to be placed outside the medium. Shooting rays from many directions into an interesting medium sounds familiar.

[31] with respect to open meshes

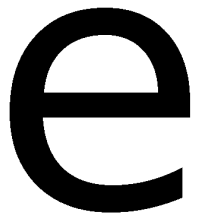


Figure 8.2: The letter "e" rendered on a 512x512 canvas

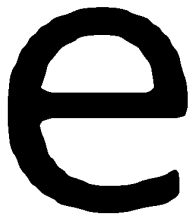


Figure 8.3: 512x512 bilinear interpolation reconstruction of a 32x32 image, thresholded. Note the stair artefacts.

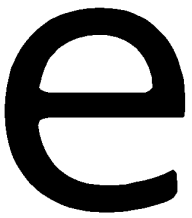


Figure 8.4: 512x512 reconstruction of a 32x32 signed distance field

Each sweep direction of the Line Sweeping process corresponds to all the photons from one (parallel) light source being part of the diffuse lighting simulation. Each ray corresponds to a photon (in a "Photon Mapping" sense, not a physical). Along its way, each ray can sample the medium density at each position and decide how much light will be absorbed, passing through and stored in the photon map. Line sweeping even allows for partial absorption, passing through and storage. Scattering the photon into a different direction would only work in the direct accumulation mode, as it would be impossible for the second pass data gathering kernel to know which photon went where. Scattering would also lead to a more inhomogeneous sampling of the medium, which may however not lead to visible artefacts given a reasonable scattering behaviour of the medium.

8.4.3 Line Sweeping Photon Mapping

To implement Line Sweeping photon mapping using the framework, we would use the parallel light directions as the sweep directions. We must not use the `PointSpreader`.

The condenser would internally store the amount of radiance arriving at the current point (i.e. the radiance coming from the light source minus the absorbed or scattered radiance). At each point in the medium, it would store the amount of radiance scattered at that point (as sweep data). The accumulator would add the radiance for each point in the medium, creating a volumetric scattering image. This would be the preparation of the actual volume rendering.

To render the volume, the view rays would march through the medium collecting the scattered light along the way, adding it up while correcting for absorption.

8.5 Signed Distance Field Generation

A signed distance field is the discretisation of a signed distance function (i.e. storage of its value at regular grid points, usually into a texture). A signed distance function

measures the distance to something (e.g. the closest point of an object). For points inside the thing, the values are negative and measure the distance to the closest point *outside* of the object.

There are applications for two-dimensional[Gre07] (see figures 8.2 through 8.4) and higher-dimensional signed distance fields[BMF03][GBF03][Sud+06]. There are different scenarios for the source data to be in. It can be given in a regular grid, defining only if a point belongs to the object or not (e.g. a black-and-white[32] rendering of a font character), or it can be given in a vector format (e.g. a curve description of a font character or a triangle mesh). Line Sweeping is applicable to both situations, regardless of the dimensionality of the problem.

8.5.1 Information in a Regular Grid

The Line Sweeping strategy for this is similar to how we implemented inpainting (chapter 6). Instead of storing the last seen light source, the condenser merely needs to store at which point the object last occurred (respectively not occurred if we are currently inside the object). In the accumulation step we do not average the individual sweep results, but take the value of the result with the smallest absolute value (i.e. the minimum for points outside of the object and the maximum for points inside the object). This means, we find all the direct paths from the object border to a point and choose the shortest.

8.5.2 Information in a Vector Format

For this situation, we need a ray-object intersection test as in inside outside testing (chapter 7). During a sweep, the condenser stores the previous and the upcoming intersection with the object. If both are empty, store positive infinity as the distance to the object. If there is no previous intersection (i.e. in the beginning of a sweep), store the remaining distance to the object. If there is no upcoming intersection (i.e. at the end of a sweep), store the distance to the previous intersection with the object. If there is both, store the distance to the closer intersection, using a negative sign iff we are currently inside the object. The accumulation is the same as in the case of the regular grid. Note that in this case, an open mesh can lead to severe artefacts.

[32] Not greyscale

References

[Bra77] Achi Brandt. 'Multi-Level Adaptive Solutions to Boundary-Value Problems'. In: *Mathematics of Computation* 31.138 (Apr. 1977), pp. 333–390. ISSN: 00255718. DOI: 10.2307/2006422. URL: http://dx.doi.org/10.2307/2006422.

[Coo86] Robert L. Cook. 'Stochastic Sampling in Computer Graphics'. In: *ACM Trans. Graph.* 5.1 (Jan. 1986), pp. 51–72. ISSN: 0730-0301. DOI: 10.1145/7529.8927. URL: http://doi.acm.org/10.1145/7529.8927.

[ICG86] David S. Immel, Michael F. Cohen and Donald P. Greenberg. 'A Radiosity Method for Non-diffuse Environments'. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 133–142. ISSN: 0097-8930. DOI: 10.1145/15886.15901. URL: http://doi.acm.org/10.1145/15886.15901.

[Kaj86] James T. Kajiya. 'The rendering equation'. In: *Computer Graphics*. 1986, pp. 143–150.

[For87] Steven Fortune. 'A sweepline algorithm for Voronoi diagrams'. In: *Algorithmica* 2.1 (1987), p. 153. ISSN: 1432-0541. DOI: 10.1007/BF01840357. URL: http://dx.doi.org/10.1007/BF01840357.

[Car88] Stefan Carlsson. 'Sketch based coding of grey level images'. In: *Signal Processing* 15.1 (1988), pp. 57–83. ISSN: 0165-1684. DOI: http://dx.doi.org/10.1016/0165-1684(88)90028-X. URL: http://www.sciencedirect.com/science/article/pii/016516848890028X.

[Fou16] Blender Foundation. *Blender*. https://www.blender.org. 1995–2016.

[Mit96] Don P. Mitchell. 'Consequences of Stratified Sampling in Graphics'. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 277–280. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237265. URL: http://doi.acm.org/10.1145/237170.237265.

[MT97] Tomas Möller and Ben Trumbore. 'Fast, Minimum Storage Ray-triangle Intersection'. In: *J. Graph. Tools* 2.1 (Oct. 1997), pp. 21–28. ISSN: 1086-7651. DOI: 10.1080/10867651.1997.10487468. URL: http://dx.doi.org/10.1080/10867651.1997.10487468.

[ZIK98] S. Zhukov, A. Iones and G. Kronin. 'An ambient light illumination model'. In: *Rendering Techniques '98: Proceedings of the Eurographics Workshop in Vienna, Austria, June 29—July 1, 1998*. Ed. by George Drettakis and Nelson Max. Vienna: Springer Vienna, 1998, pp. 45–55. ISBN: 978-3-7091-6453-2. DOI: 10.1007/978-3-7091-6453-2_5. URL: http://dx.doi.org/10.1007/978-3-7091-6453-2_5.

[Cro02] Douglas Crockford. *JavaScript Object Notation*. 2002. URL: http://json.org.

[BMF03] R. Bridson, S. Marino and R. Fedkiw. 'Simulation of Clothing with Folds and Wrinkles'. In: *Symposium on Computer Animation*. Ed. by D. Breen and M. Lin. The Eurographics Association, 2003. ISBN: 1-58113-659-5. DOI: 10.2312/SCA03/028-036.

[GBF03] Eran Guendelman, Robert Bridson and Ronald Fedkiw. 'Nonconvex Rigid Bodies with Stacking'. In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 871–878. ISSN: 0730-0301. DOI: 10.1145/882262.882358. URL: http://doi.acm.org/10.1145/882262.882358.

[Sud+06] Avneesh Sud, Naga Govindaraju, Russell Gayle, Ilknur Kabul and Dinesh Manocha. 'Fast Proximity Computation Among Deformable Models Using Discrete Voronoi Diagrams: Implementation Details'. In: *ACM SIGGRAPH 2006 Sketches*. SIGGRAPH '06. Boston, Massachusetts: ACM, 2006. ISBN: 1-59593-364-6. DOI: 10.1145/1179849.1180069. URL: http://doi.acm.org/10.1145/1179849.1180069.

[07] *Ambient Occlusion in the Unreal Engine 3*. [Online; accessed 2016-10]. 2007. URL: https://udn.epicgames.com/Three/PostProcessEffectReference.html#AmbientOcclusionEffect.

[Bri07] Robert Bridson. 'Fast Poisson Disk Sampling in Arbitrary Dimensions'. In: *ACM SIGGRAPH 2007 Sketches*. SIGGRAPH '07. San Diego, California: ACM, 2007. ISBN: 978-1-4503-4726-6. DOI: 10.1145/1278780.1278807. URL: http://doi.acm.org/10.1145/1278780.1278807.

[Gre07] Chris Green. 'Improved Alpha-tested Magnification for Vector Textures and Special Effects'. In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: ACM, 2007, pp. 9–18. ISBN: 978-1-4503-1823-5. DOI: 10.1145/1281500.1281665. URL: http://doi.acm.org/10.1145/1281500.1281665.

[Mit07] Martin Mittring. 'Finding next gen: CryEngine 2'. In: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*. San Diego, California: ACM, 2007, pp. 97–121. DOI: 10.1145/1281500.1281671. URL: http://dx.doi.org/10.1145/1281500.1281671.

[SA07] Perumaal Shanmugam and Okan Arikan. 'Hardware Accelerated Ambient Occlusion Techniques on GPUs'. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D '07. Seattle, Washington: ACM, 2007, pp. 73–80. ISBN: 978-1-59593-628-8. DOI: 10.1145/1230100.1230113. URL: http://doi.acm.org/10.1145/1230100.1230113.

[BS08] Louis Bavoil and Miguel Sainz. 'Screen space ambient occlusion'. In: *NVIDIA developer information: http://developers. nvidia. com* 6 (2008).

[BSD08] Louis Bavoil, Miguel Sainz and Rouslan Dimitrov. 'Image-space Horizon-based Ambient Occlusion'. In: *ACM SIGGRAPH 2008 Talks*. SIGGRAPH '08. Los Angeles, California: ACM, 2008, 22:1–22:1. ISBN: 978-1-60558-343-3. DOI: 10.1145/1401032.1401061. URL: http://doi.acm.org/10.1145/1401032.1401061.

[Orz+08] Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot and David Salesin. 'Diffusion Curves: A Vector Representation for Smooth-Shaded Images'. In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)*. Vol. 27. 2008. URL: http://maverick.inria.fr/Publications/2008/OBWBTS08.

[Gro09] Khronos Group. *The OpenCL Specification Version: 1.0*. 2009. URL: https://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf.

[RBA09] Christoph Reinbothe, Tamy Boubekeur and Marc Alexa. 'Hybrid ambient occlusion'. In: *Proceedings of the Eurographics Symposium on Rendering*. Vol. 236. 2009.

[GWB10] Sven Grewenig, Joachim Weickert and Andrés Bruhn. 'From Box Filtering to Fast Explicit Diffusion'. In: *Proceedings of the 32Nd DAGM Conference on Pattern Recognition*. Darmstadt, Germany: Springer-Verlag, 2010, pp. 533–542. ISBN: 3-642-15985-0, 978-3-642-15985-5. URL: http://dl.acm.org/citation.cfm?id=1926258.1926320.

[MSW10] Oliver Mattausch, Daniel Scherzer and Michael Wimmer. 'High-Quality Screen-Space Ambient Occlusion using Temporal Coherence'. In: *Computer Graphics Forum*. Vol. 29. 8. Wiley Online Library. 2010, pp. 2492–2503.

[TW10] Ville Timonen and Jan Westerholm. 'Scalable Height Field Self-Shadowing'. In: *Computer Graphics Forum*. Vol. 29. 2. Wiley Online Library. 2010, pp. 723–731.

[BLW11] John C. Bowers, Jonathan Leahey and Rui Wang. 'A Ray Tracing Approach to Diffusion Curves'. In: *Comput. Graph. Forum* 30.4 (2011), pp. 1345–1352. DOI: 10.1111/j.1467-8659.2011.01994.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2011.01994.x.

[McG+11] Morgan McGuire, Brian Osman, Michael Bukowski and Padraic Hennessy. 'The alchemy screen-space ambient obscurance algorithm'. In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. ACM. 2011, pp. 25–32.

[But+12] D. J. Butler, J. Wulff, G. B. Stanley and M. J. Black. 'A naturalistic open source movie for optical flow evaluation'. In: *European Conf. on Computer Vision (ECCV)*. Ed. by A. Fitzgibbon et al. (Eds.) Part IV, LNCS 7577. Springer-Verlag, Oct. 2012, pp. 611–625.

[SP12] V. Shumskiy and A. Parshin. 'GPU Ray Tracing–Comparative Study of Ray-Triangle Intersection Algorithms'. In: (2012).

[13] *Boost Compute – A C++ GPU Computing Library for OpenCL*. [Online; accessed 2016-10]. 2013. URL: https://github.com/boostorg/compute.

[Tim13] Ville Timonen. 'Line-Sweep Ambient Obscurance'. In: *Computer Graphics Forum (Proceedings of EGSR 2013)* 32.4 (2013), pp. 97–105. URL: http://wili.cc/research/lsao/.

[And14] Daniel Andersson. *3D model of a medieval church*. 2014. URL: http://opengameart.org/content/medieval-church.

[San+14] Artur Lira dos Santos, UFPE Center, Av Jornalista Anibal, Cidade Universit'aria, Veronica Teichrieb and Jorge Lindoso. 'Review and Comparative Study of Ray Traversal Algorithms on a Modern GPU Architecture'. In: (2014).

[Tak+14] Kenshi Takayama, Alec Jacobson, Ladislav Kavan and Olga Sorkine-Hornung. 'A Simple Method for Correcting Facet Orientations in Polygon Meshes Based on Ray Casting'. In: *Journal of Computer Graphics Techniques (JCGT)* 3.4 (Dec. 2014), pp. 53–63. ISSN: 2331-7418. URL: http://jcgt.org/published/0003/04/02/.

[PJS15] Romain Prévost, Wojciech Jarosz and Olga Sorkine-Hornung. 'A Vectorial Framework for Ray Traced Diffusion Curves'. In: *Computer Graphics Forum* 34.1 (Feb. 2015), pp. 253–264. ISSN: 1467-8659. DOI: 10.1111/cgf.12510.

[Dev16] Advanced Micro Devices. *Introducing the Radeon Rays SDK*. 2016. URL: https://gpuopen.com/wp-content/uploads/2016/08/169798-A_AMD_RadeonRays_Intro_FNL.pdf.

[EE16] Institute of Electrical and Electronics Engineers. *The 2016 Top Programming Languages*. http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages. 2016.

[Kjo+16] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik and Saman Amarasinghe. 'Simit: A Language for Physical Simulation'. In: *ACM Trans. Graph.* 35.2 (May 2016), 20:1–20:21. ISSN: 0730-0301. DOI: 10.1145/2866569. URL: http://doi.acm.org/10.1145/2866569.

[16a] *Screen space ambient occlusion — Wikipedia, The Free Encyclopedia*. [Online; accessed 2016-10-05]. 2016. URL: https://en.wikipedia.org/w/index.php?title=Screen_space_ambient_occlusion&oldid=739371638.

[16b] *Tom Clancy's The Division – Patch notes 1.01*. [Online; accessed 2016-10]. 2016. URL: http://tomclancy-thedivision.ubi.com/game/en-US/news/152-241765-16/the-division-day-1-patch-notes.

[Wei+16] Joachim Weickert, Sven Grewenig, Christopher Schroers and Andrés Bruhn. 'Cyclic Schemes for PDE-Based Image Analysis'. In: *Int. J. Comput. Vision* 118.3 (July 2016), pp. 275–299. ISSN: 0920-5691. DOI: 10.1007/s11263-015-0874-1. URL: http://dx.doi.org/10.1007/s11263-015-0874-1.