Cover page

Exam information
NDAB12006E - Bachelorprojekt i datalogi (Christian, Jens, Jonas og Mathias)

Handed in by
Mathias Friis Rasmussen
tjc725@alumni.ku.dk

Jonas Kristensen
fpk406@alumni.ku.dk

Jens Nissen-Juul Sørensen
qrw992@alumni.ku.dk

Christian Dybdahl Troelsen
tfp233@alumni.ku.dk

Exam administrators
DIKU Eksamen
uddannelse@diku.dk

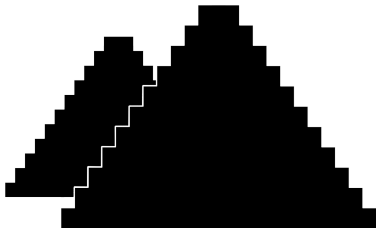
Assessors
Troels Henriksen
Examiner
athas@di.ku.dk
+4535335718

Patrick Bahr
Co-examiner
paba@itu.dk

Hand-in information

Titel, engelsk: FutSpace - A Parallelizable Implementation of the Voxel Space Rendering Algorithm
Tro og love-erklæring: Yes

UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Bachelor Project

Jonas Kristensen (fpk406) Mathias Rasmussen (tjc255)
Jens Sørensen (qrw992) Christian Troelsen (tfp233)

FutSpace

A Parallelizable Implementation of the Voxel Space Rendering Algorithm

Advisor : Troels Henriksen

Handed in : June 8, 2020

Abstract

This project examines how basic ray casting techniques work and how the voxel space algorithm relate to these techniques. It then examines how data parallelism can be applied to the voxel-based graphics engine, and whether the application of data parallelism has any quantifiable benefits over the existing squential voxel-based rendering algorithm. The report outlines the creation of an implementation of the Voxel Space algorithm in the functional parallel programming language Futhark. In addition to the standard Voxel Space implementation it also outlines a series of improvements made to the engine in order to increase the fidelity of the rendered images.

Contents

1 Introduction **1**

2 Voxel Space **3**
2.1 Historical Background 3
2.2 Algorithm 4
2.3 *VoxelSpace* – A JavaScript Implementation 13

3 Data Parallelism **19**
3.1 General Idea 19
3.2 Programming in Parallel 21
3.3 Futhark 22

4 The Translation **25**
4.1 Overview 25
4.2 Generating Color-Height Pairs 25
4.3 Rendering a Frame 28

5 Implementation Analysis **33**
5.1 Establishing a Cost Model 33
5.2 Work and Span 33
5.3 A Cost Model for Futhark 35
5.4 Intermezzo – Correctness of Implementation 39
5.5 Applying the Cost Model 42
5.6 Benchmarking 44

6 Improvements **48**
6.1 Generalising the algorithm 48
6.2 Shadow rendering 50
6.3 Bilinear Filtering 55

7 Conclusion **58**

Appendices **59**

A Code **60**

B Proofs **74**
B.1 Work and Span of First-order Combinators 74
B.2 Associativity Proofs 75

C Simplification of rotation matrix **77**

Chapter 1

Introduction

This report covers the implementation of *FutSpace*, a voxel-based 2.5D graphics engine, based on the 1992 Voxel Space 3D engine created and patented by Freeman[Fre96].
Futspace is implemented using the parallel programing language Futhark created by Elsman, Henriksen, and Oancea[EHO20]. **Chapter 2** starts with a brief historical look at videogame graphics and continues on by examining the common pseudo 3D rendering methode known as raycasting and how this methode relates to the voxel space algorithm. The chapter contains a matematical examination of the voxel space algirthm. **Section 2.3** covers the examination of the JavaScript implementation for the voxel space algorithm created by Macke[Mac20] and language specific features used in this implementation.
Next **Chapter 3** explores the general idea of data prallelism and how the JavaScript implementation can be altered to take advantage of the data prallelism offered by the Futhark language. **Chapter 4** goes into detail on how we made the translation from the sequential JavaScript code to the parallel Futhark implementation.**Chapter 5** establishes a suitable cost model for analysing the Futspace implementation and goes through an runtime analysis of the implementation and how this compares to the JavaScript implementation. Lastly **Chapter 6** explore a series of improvements that we have made to the original Voxel Space engine. These improvements include a generalization that allows for the drawing of arbitrary mathematical functions and various visual improvements. The full implementation and installation guide can be found on github[Kri+20]

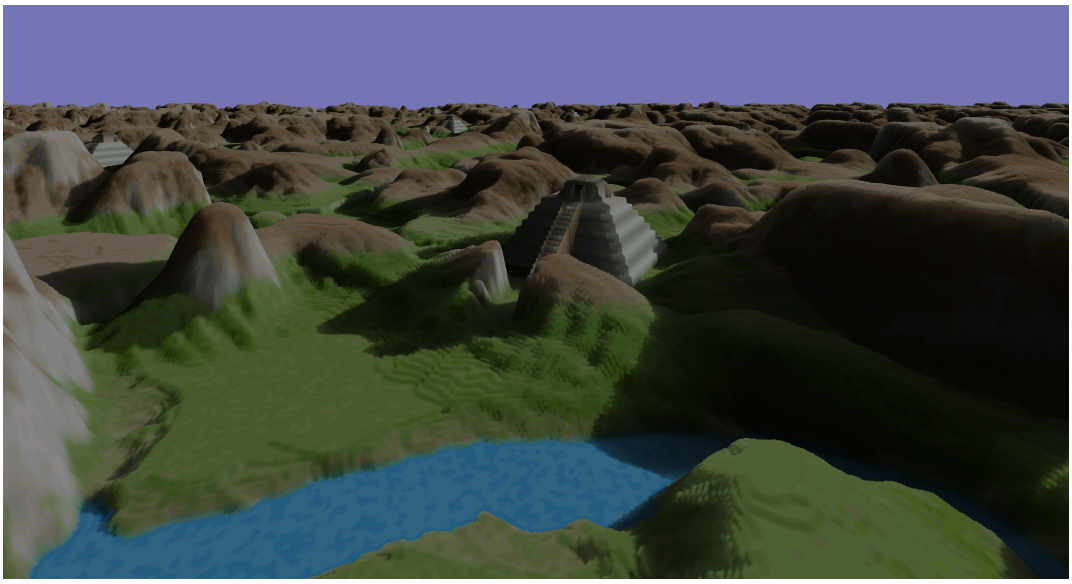


Figure 1.1: Screenshot of a frame rendered by *FutSpace*

Chapter 2

Voxel Space

2.1 Historical Background

Computer graphics has existed since the inception of the computer. There exists two types of computergraphics: Raster graphics and Vector graphics. This report will not touch any further on the topic of vector graphics. The raycasting which will be explored in **Section 2.2.1** and the Voxel Space algorithm on which we base our project are both rasterised graphics.

In 1992 ID software released *Wolfenstein 3D* which redefined the first-person-shooter genre and the demand for 3D gameworlds. There had been 3D games before this point. They were mainly flight simulators with little widespread popularity. Due to the popularity of *Wolfenstein 3D* the request for 3D-graphics grew. Due to the limited computing power available at the time full 3D rendering was not available to the average consumer. This gave rise to pseudo-3D or 2.5D as it is also called. A very popular technic for creating 2.5D graphics was the raycasting method used by *Wolfenstein 3D*

Voxel Space is a 2.5D engine and its approach to computer graphics is similar to the ray casting method in *Wolfenstein 3D* with some noticable differences that will be explored in **Section 2.2**.
The Voxel Space 3D engine was developed in 1992 by Kyle Freeman at Novalogic for the video game *Comanche: Maximum Overkill*. In 1996 Novalogic was awarded the patent for the engine[Fre96]. The engine was upgraded in 1997 for the games *Comanche 3* and *Armored Fist 2* to allow for the inclusion of traditional texture-mapped polygons[96]. In 1999 the last generation of the engine Voxel Space 32 was released. This iteration of the engine featured 32-bit colors and 360 degrees terrain rotation[Sta00]. These two upgraded engines were patented in 2000[Fre00].

The term GPU as we know it today, was coined by Sony in 1994 with the release of the the Playstation 1 and popularized by Nvidia with the launched their Geforce 256. The term GPU was defined by Nvidia as: *"a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second"*[Nvi]. Today GPU accelerated calculation has become ubiquitous. However there still exist a large number of algorithms from before the introduction of the GPU, that might benefit from the added computational power provided by the GPU. The aim of this project is therefore to go back and examine if the voxel space algorithm can be translated to parallel execution and benefit from

the improvements of mordern hardware.
The patent for the Voxel Space 3D engine expired in 2013. On September 23. 2017 Github user Sebastian Macke (s-macke) created a Javascript implementation of Kyle Freemans original Voxel space 3D engine. This implementation while written in a more modern language than the original Voxel Space engine, still uses a sequential approach to the problem and not a parallelized approach which can take greater advantage of the GPU's power. It is this implementation that we have based our project on.

2.2 Algorithm

2.2.1 General Method

In order to explain the Voxel Space algorithm, it is first necessary to understand what ray casting is. Ray casting is a rendering method for creating images that appear 3-dimensional from data that is 2-dimensional. Core to the concept of ray casting is the idea of shooting a number of *rays* into a scene from a point typically called the *camera*. Typically the amount of rays used is equal to the width of the final image, such that each ray corresponds to one unique column in the final image. Each ray will step through the scene over a number of iterations until either the ray intersects with a scene object or a maximum ray length is reached. In the case of a ray intersecting with an object in the scene, its corresponding column in the final image is colored with the color of the intersected object, and the ray iteration ends.

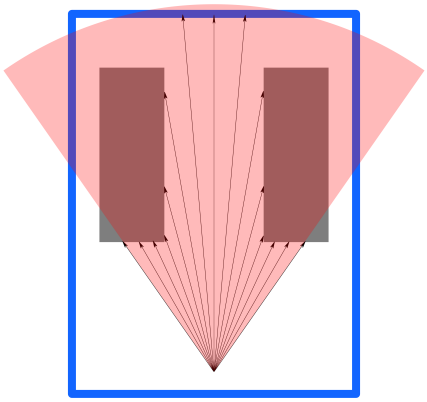


Figure 2.1: Rays shot from camera into two-dimensional scene that will be raycasted

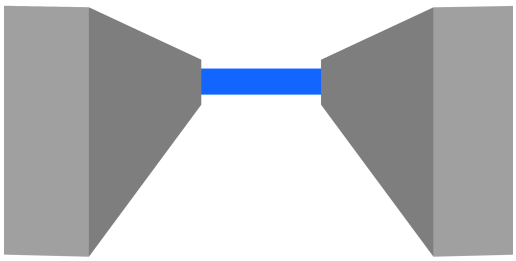


Figure 2.2: Approximate resulting render of ray casting scene in **Figure 2.1**, but with a much larger amount of rays

However, in order to create a sense of vertical perspective, it is not enough to simply color the entire column. Instead, a perspective calculation is performed, such that the part of the column that is colored depends on the distance between the camera and the position of the intersection. This, as **Figure 2.2** shows, makes the rear wall of the scene look further away than the foreground. Furthermore, the horizontal perspective is directly related to the sum of the relative angles of the rays, and this sum is known as the *field of view*. The field of view is highlighted in **Figure 2.1** as the semi-transparent red circle segment that encloses all the rays. The circle perimeter is the maximum

ray length from the camera. Typically the rays are evenly distributed within the field of view, like in **Figure 2.1**, such that the relative angle between any two rays is the same.

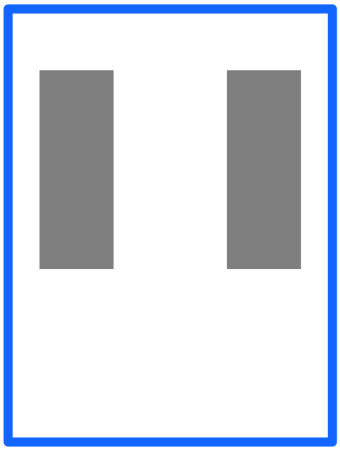


Figure 2.3: Scene from **Figure 2.1** without camera and rays. White indicates no object

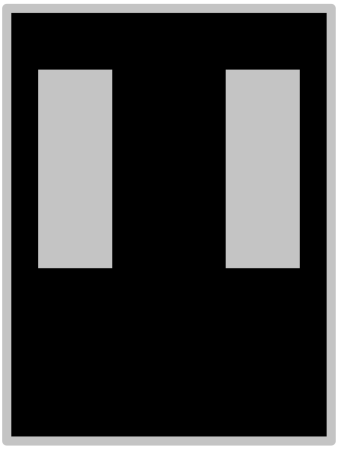


Figure 2.4: Grayscale representation of heights in scene. Black is the smallest possible height

While the ray casting outlined above is of course about as rudimentary as ray casting gets, it does offer an opportunity to highlight the most defining feature of the Voxel Space algorithm, namely the fact that the Voxel Space algorithm introduces the concept of object and camera height. The only information available to the ray caster is the scene in **Figure 2.3**, which by the use of color indicates where objects are positioned, such that if a ray enters a part of the scene that is not white, an intersection will occur. However, the Voxel Space algorithm treats every part of the scene as some object, including the parts that are white. Each of these objects have a corresponding height value attached to them, which is visualized in **Figure 2.4**.

This will, after a redefinition of what rays and intersections are in Voxel Space, allow us to render overlapping objects while still technically staying in a two-dimensional scene. As we in Voxel Space consider every part of the scene in **Figure 2.3** an object, we can no longer rely on the same intersection logic as the basic ray caster. Instead, when drawing the image column for some ray, we need the color and height information from **Figure 2.3** and **Figure 2.4** for every iteration of the ray, such that we can use this information to build the image column in layers.

In Voxel Space the color and height information are represented by sets C and H respectively. These sets can be seen as discrete finite functions mapping from to $\mathbb{N}^2$ to $\mathbb{N}$. We will use the notation $\mathcal{D}(C)$ to denote the domain of C and $\mathcal{I}(C)$ to denote the image of C. The domain and image of H are denoted similarly. Even though C and H are finite functions, Voxel Space manages to create the illusion of an endless terrain. This is achieved through the use of tiling, which is defined in **Equation 2.9**.

For each layer, the color and height information for a given intersection is used to perform a modified version of the perspective transformation described for the basic ray caster. However, the order in which these layers are drawn on the image column is important for the correctness of the algorithm. If the layers are drawn in the order that the intersections occured, then you risk that the

part of the image column corresponding to the foreground is overwritten by some later occurring intersection. The problem with occluding layers in foreground, that have already been written, will simply be referred to as the occlusion problem. By drawing the layers in the reverse order that the intersections occured in, the foreground will always be drawn last, which is a solution to the occlusion problem that we will refer to as *back-to-front*. An issue with the back-to-front method is that it is wasteful; we end up writing colors to the column that will be overwritten in some later step of the method.

A more efficient solution to the occlusion problem is *front-to-back*. In the front-to-back method we draw layers in the order of increasing distance from the camera by keeping track of the parts of each column that have already been written to in a previous iteration. Then, by simply checking whether a new layer for some column has a lower height in the screenbuffer than the maintained value for that column, we avoid overwriting any useful information and simply move onto the next layer. As we in Voxel Space want the color and height values for all objects that a ray intersects until the maximum ray distance, we can think of the intersections for all rays as a set of points withins the field of view that we want to sample, thus we define a different field of view system for Voxel Space than the one defined for the basic ray caster.

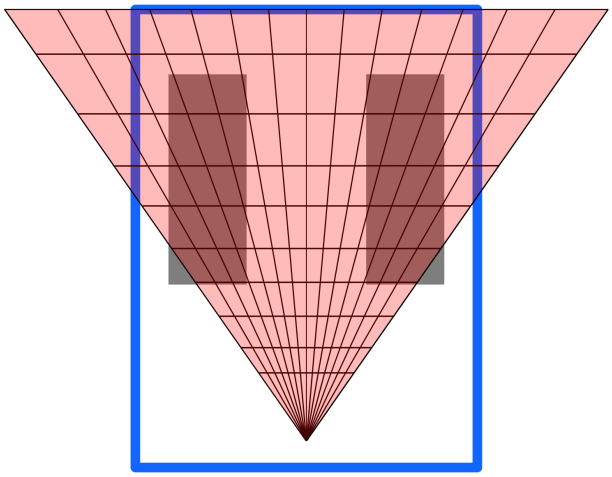


Figure 2.5: Voxel Space-equivalent of **Figure 2.1**

As seen in **Figure 2.3**, the new representation of the camera and field of view does not offer a 1:1 translation of the ray intersections from **Figure 2.1** and **Figure 2.2**. Instead, approximate positions of the ray intersections are found by creating a set of line segments that run horizontally across the field of view, which also causes the field of view to no longer be a circle segment but instead a triangle.

We will now explain how the components of a basic ray caster can be defined mathematically such that we can explain how the field of view system in **Figure 2.5** can be derived, and exactly what is performed in the Voxel Space algorithm.

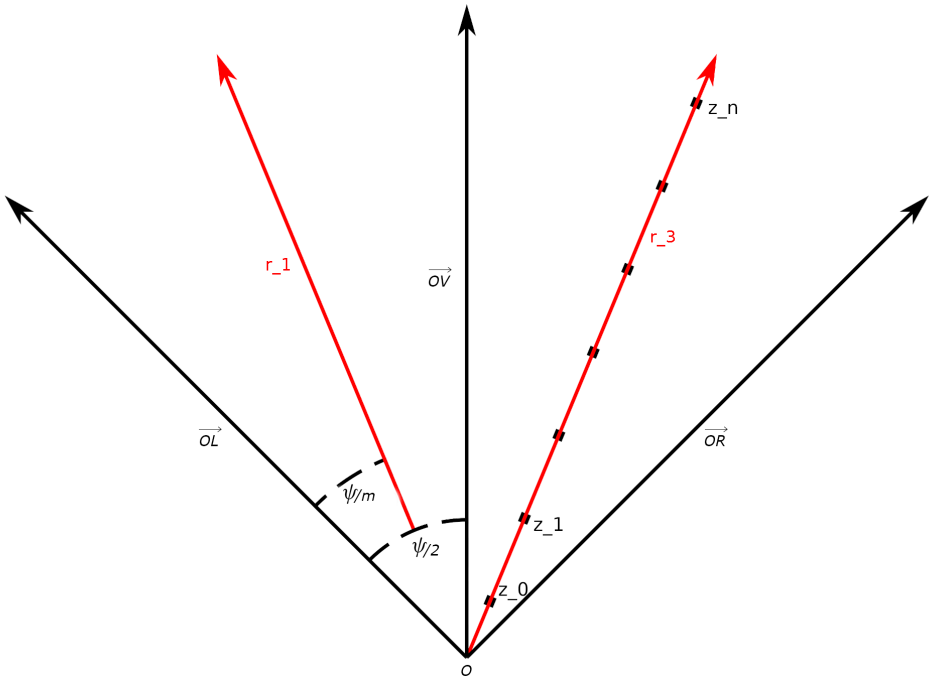


Figure 2.6: Figure of ray casting. r_0, r_2 and r_5 not shown for clarity

Mathematically, ray casting is a way of rendering a scene $W \in \mathbb{R}^2$ such that we produce an image of dimensions $l \times m$. Rays r_j, $j \in \{0, 1, ..., m\}$ are cast and iterated n times from a point $O \in W$, and if r_j and an object in the scene intersect at distance $z_i, i \in \{0, 1, ..., n\}$, some part of column j in the final image assumes the color of the object. The direction of the rays is determined by a view vector $\overrightarrow{OV}$ that represents the direction of the rendering. The field of view is bounded by the vectors $\overrightarrow{OL}$ and $\overrightarrow{OR}$. We let $\overrightarrow{OL}$ be $\overrightarrow{OV}$ rotated by $\frac{1}{2}\psi$ and $\overrightarrow{OR}$ be $\overrightarrow{OV}$ rotated by $-\frac{1}{2}\psi$. The relative angle of any two rays in the field of view is then found by dividing the angle between $\overrightarrow{OL}$ and $\overrightarrow{OL}$ by m. The vertical perspective of the final image can be considered a transformation in $\mathbb{N} \to \mathbb{N}$, such that the height of a rendered column for ray r_j decreases as z_k increases.

2.2.2 Voxel Space algorithm

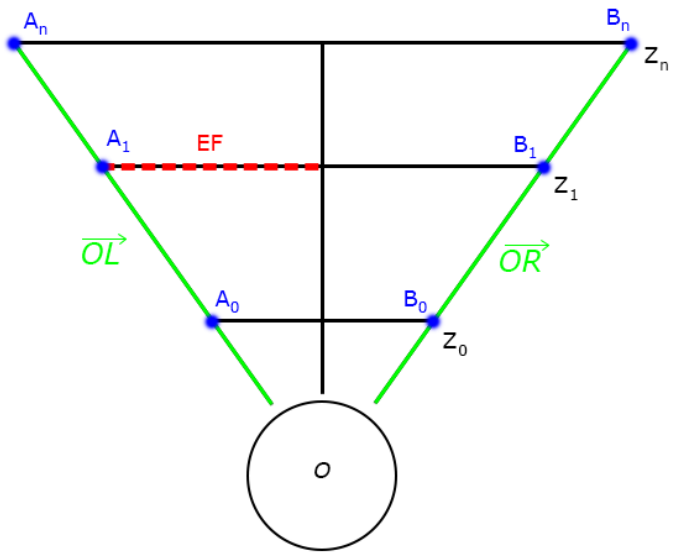


Figure 2.7: crude image of the mathematical entities used in the voxel space algorithm

The Voxel Space algorithm, however, does not have any concept of rays. Instead an approximation of the position of the intersection test for r_j at z_k is found by introducing line segments $\overline{A_k B_k}, k = \{0, 1, ..., n\}$ for which it is the case that A_k intersects $\overrightarrow{OL} \cdot z_k$ and B_k intersects $\overrightarrow{OR} \cdot z_k$. Each $\overline{A_k B_k}$ is split into m segments $\overline{EF}_j$, such that the position of the intersection test of r_j at z_k is approximated by the start point of $\overline{EF}_j$ of $A_k B_k$. We furthermore define M_k as the midpoint of $\overline{A_k B_k}$ and ω as the camera height. The horizontal perspective of the final image is then still a function of the total field of view, and the vertical perspective transformation is the same as in the ray casting scenario outlined above.

2.2.3 Determining A_k and B_k

A_k and B_k are the end-points of vectors $\overrightarrow{OA_k}$ and $\overrightarrow{OB_k}$ with origin $O = (x_0, y_0)$. Hence they may be characterized by the following equations:

$$A_k = \overrightarrow{OA_k} + O \tag{2.1}$$

$$B_k = \overrightarrow{OB_k} + O \tag{2.2}$$

Assume $\overrightarrow{OV}$ is the vector defined by $\sin(\theta)$ and $\cos(\theta)$:

$$\overrightarrow{OV} = \begin{pmatrix} \sin(\theta) \\ \cos(\theta) \end{pmatrix} \tag{2.3}$$

We then have:

$$\overrightarrow{OM_k} = z_k \cdot \overrightarrow{OV} = z_k \cdot \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} \tag{2.4}$$

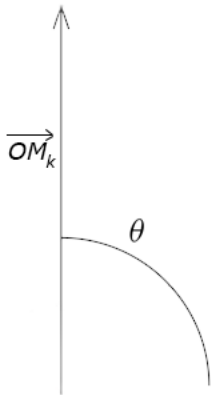


Figure 2.8: Determining $\overrightarrow{OM_k}$ by scaling $\overrightarrow{o}$ with z_k

$\overrightarrow{OA_k}$ can now be determined by rotating $\overrightarrow{OM_k}$ $\frac{1}{2}\psi$ degrees counter-clockwise. To perform this rotation we multiply $\overrightarrow{OM_k}$ with the rotationmatrix $R\left(\frac{1}{2}\psi\right)$:

$$\begin{aligned}\overrightarrow{OA_k} &= \overrightarrow{OM_k} \cdot R(\psi/2) \\ &= z_k \cdot \begin{pmatrix}\cos(\psi/2) & -\sin(\psi/2) \\ \sin(\psi/2) & \cos(\psi/2)\end{pmatrix} \cdot \begin{pmatrix}\cos(\theta) \\ \sin(\theta)\end{pmatrix} \\ &= z_k \cdot \begin{pmatrix}\cos(\theta) \cdot \cos(\psi/2) - \sin(\theta) \cdot \sin(\psi/2) \\ \cos(\theta) \cdot \sin(\psi/2) + \sin(\theta) \cdot \cos(\psi/2)\end{pmatrix}\end{aligned} \tag{2.5}$$

$\overrightarrow{OB_k}$ can be determined by rotating $\overrightarrow{OM_k}$ $\frac{1}{2}\psi$ degrees clockwise. To perform this rotation we multiply $\overrightarrow{OM_k}$ with the rotationmatrix $R\left(2\pi - \frac{1}{2}\psi\right)$:

$$\begin{aligned}\overrightarrow{OB_k} &= \overrightarrow{OM_k} \cdot R(2\pi - \psi/2) \\ &= z_k \cdot \begin{pmatrix}\cos(2\pi - \psi/2) & -\sin(2\pi - \psi/2) \\ \sin(2\pi - \psi/2) & \cos(2\pi - \psi/2)\end{pmatrix} \cdot \begin{pmatrix}\cos(\theta) \\ \sin(\theta)\end{pmatrix} \\ &= z_k \cdot \begin{pmatrix}\cos(\theta) \cdot \cos(2\pi - \psi/2) - \sin(\theta) \cdot \sin(2\pi - \psi/2) \\ \cos(\theta) \cdot \sin(2\pi - \psi/2) + \sin(\theta) \cdot \cos(2\pi - \psi/2)\end{pmatrix}\end{aligned} \tag{2.6}$$

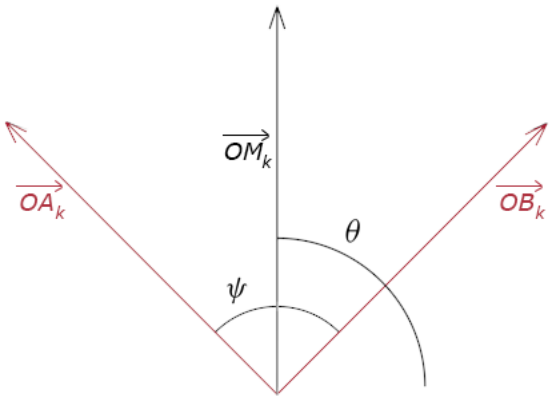


Figure 2.9: Determining $\overrightarrow{OA_k}$ and $\overrightarrow{OB_k}$ by rotating $\overrightarrow{OM_k}$ $\psi/2$ degrees counter-clockwise and $\psi/2$ degrees clock-wise respectively

2.2.4 Segmenting $\overline{A_k B_k}$

We wish to raster $\overline{A_k B_k}$ so that it matches the number of columns on the screen to be rendered on. This can be done by segmenting $\overline{A_k B_k}$ into m sublines $\overline{EF}_j$ with dimensions (dx, dy):

$$\begin{pmatrix} dx \\ dy \end{pmatrix} = \frac{\overline{A_k B_k}}{m} = \frac{\begin{pmatrix} A_k.x - B_k.x \\ A_k.y - B_k.y \end{pmatrix}}{m} \tag{2.7}$$

The start-point E_j of $\overline{EF}_j$ can then be defined as

$$E_j = \begin{pmatrix} dx \\ dy \end{pmatrix} \cdot j \tag{2.8}$$

$\overline{EF}_j$ has an associated color $c \in \mathcal{I}(C)$ and base-height $h \in \mathcal{I}(H)$. To determine c and h, tiling needs to be taken into account. This can be done by introducing modular arithmetic. Because C and H are $q \times r$-dimensioned discrete spaces, we define a function p such that:

$$p(x, y) = \begin{pmatrix} \lfloor x \rfloor \bmod q \\ \lfloor y \rfloor \bmod r \end{pmatrix} \tag{2.9}$$

c and h may then be derived by applying p to E_j and mapping the resulting index (x', y') into C and H:

$$c = C\big(p\big(E_j.x, E_j.y\big)\big), \tag{2.10}$$

$$h = H\big(p\big(E_j.x, E_j.y\big)\big) \tag{2.11}$$

2.2.5 Creating a Vertical Perspective

The color value from C collected by each ray at $p(k, j)$ in the subspace U, needs to be projected into the discrete finite $l \times m$ space T, representing the screen space. This projection has to account for creating a horizontal and a vertical perspective on the j-axis and h'-axis of T, respectively.

Since the number of segments on each depth-layer is consistent since it represents each ray, the horizontal perspective is created by the process of generating the rays themselves. Keeping the number of rays equal to m simplifies the projection for the j coordinate in T, to simply be the same coordinate as in U.

To create the effect of a vertical perspective, the greater the distance each depth-layer is from the origin O in U, the lower the value of h' in T it needs to have [1]. This effect follows a number series converging toward zero.

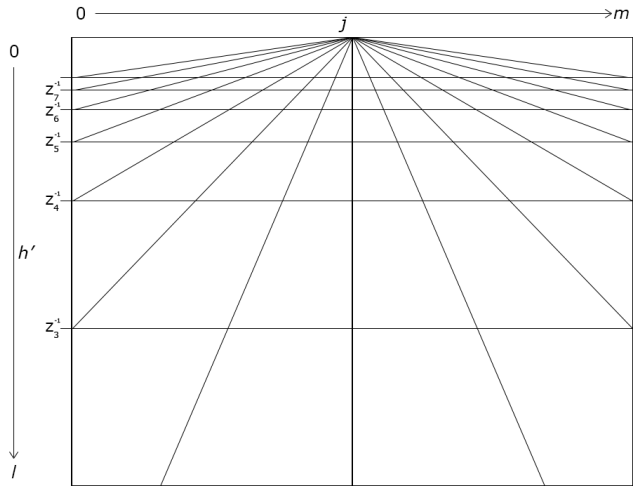


Figure 2.10: Projection from U into T

The effect can be created by using a number series starting at one, and which diverges towards $+\infty$ to generate the distance scalar z_k for each depth-layer. The reciprocal z_k^{-1} can then be used in the projection to achieve the desired effect.

The value of z_k^{-1} only has a range of $[1, 0)$ and a scaling constant β is needed to map across the range l in T.

The projection from $U \rightarrow T$ becomes

$$project(p(k, j)) = \begin{bmatrix} z_k^{-1} \cdot \beta \\ j \end{bmatrix}_T \tag{2.12}$$

2.2.6 Accounting for Height and Camera Tilt

The definition of **Equation 2.12** can be extended, to take the value of H and ω into account.

By letting $\gamma_{k,j} = \omega - H(p(k, j))$ height can be created in the terrain.

The projection now becomes

$$project(p(k, j)) = \begin{bmatrix} \gamma_{k,j} \cdot z_k^{-1} \cdot \beta \\ j \end{bmatrix}_T \tag{2.13}$$

[1] Since it represents the screen space, the origin of U is located at the top and grows down on the h' axes

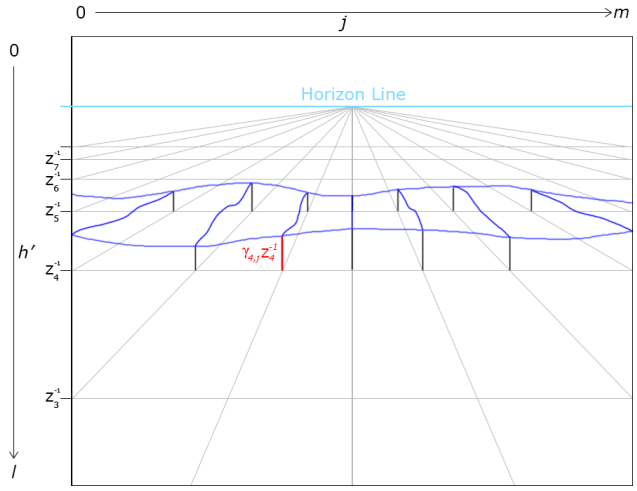


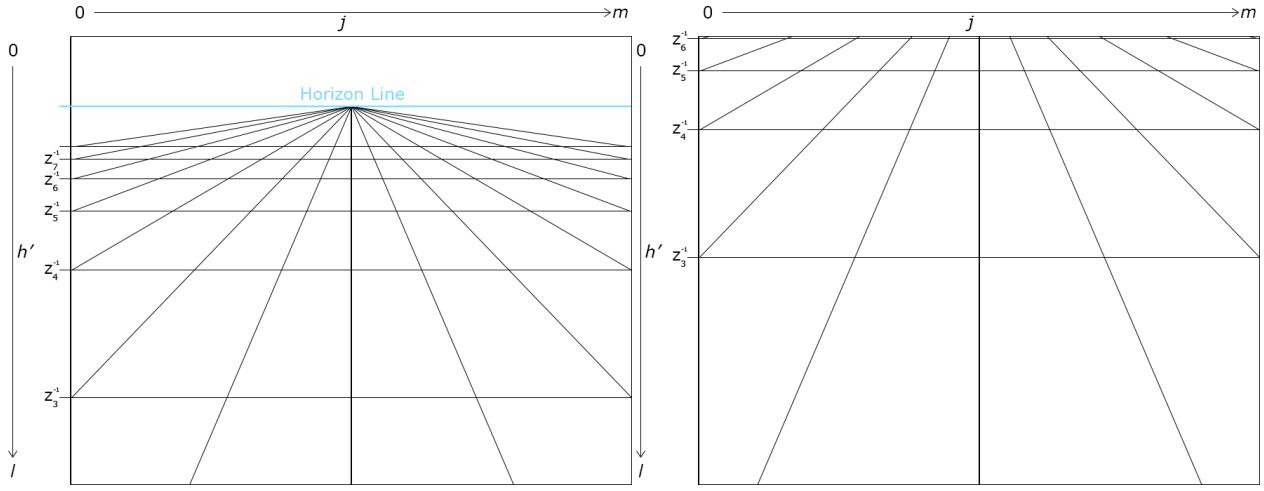
Figure 2.11: Terrain created by using $\gamma_{k,j}$ values for each $p(k, j)$

In this projection the value $h' = 0$ in T, represents the horizon line of the perspective. By introducing a value ϵ, the h' value of the horizon in T can be set.

The projection now becomes

$$project(p(k, j)) = \begin{bmatrix} \gamma_{k,j} \cdot z_k^{-1} \cdot \beta + \epsilon \\ j \end{bmatrix}_T \tag{2.14}$$

Projected values of $p(k, j)$ can result in values $h' > l$, but still has the vertical perspective effect applied to them. By using a negative value of ϵ, these values can become visible in T and create the illusion of tilting downward.



(a) Positive ϵ value

(b) Negative ϵ value

Figure 2.12: Using ϵ to tilt up and down

2.3 *VoxelSpace* – A JavaScript Implementation

In Section 2.2 we introduced the Voxel Space rendering algorithm and explained its steps in detail. Now we will look at how it may be implemented in a straightforward sequential fashion. We will base our explanation on *VoxelSpace*, which is a reverse-engineered implementation of the original Comanche engine written in JavasScript[Mac20]. The full implementation code can be found in Listing A.1. A substantial part of this code implements an interactive application, which is not relevant to us. The constructs that are relevant are the `camera`, `map` and `screendata` objects defined in lines 6-44 and the `DrawVerticalLine()` and `Render()` functions defined at lines 235-251 and lines 272-320 respectively. We will start by explaining the most important fields in the `camera`, `map` and `screendata` objects. This explanation will then allow us to delve deeper into how `Render()` in conjunction with `DrawVerticalLine()` renders a frame on screen using the front-to-back approach described in Section 2.2.1.

The `camera` object is defined as:

```
var camera =
{
    x: 512.,
    y: 800.,
    height: 78.,
    angle: 0.,
    horizon: 100.,
    distance: 800
};
```

Listing 2.1: Definition of `camera` object

Fields `x` and `y` in Listing 2.1 together make up what we referred to as origin O in Section 2.2. `height`, `angle` and `horizon` represent ω, angle θ, and vertical tilt ϵ of the virtual observer (i.e. camera) respectively. Finally `distance` designates the max rendering distance d. Now lets proceed to the definition of the `map` object:

```
var map =
{
    width: 1024,
    height: 1024,
    shift: 10,
    altitude: new Uint8Array(1024 * 1024),
    color: new Uint32Array(1024 * 1024)
};
```

Listing 2.2: Definition of `map` object

Listing 2.2 contain all information related to C and H. The fields `width` and `height` are the dimensions q and r of C and H. C and H themselves are represented by fields `color` and `altitude` which are one-dimensional arrays of length $q \cdot r$ storing respectively 32-bit unsigned

integers and 8-bit unsigned integers in row-major order. The `shift` field is a scalar used for index calculations. We will return to its specific function later on. We finish our discussion of objects by defining `screendata`:

```
var screendata =
{
    canvas: null,
    context: null,
    imagedata: null,

    bufarray: null,
    buf8: null,
    buf32: null,

    backgroundcolor: 0xFFE09090
};
```

Listing 2.3: Definition of `screendata`

The most important fields in **Listing 2.3** are `canvas` and `buf32`. These are defined without initialization, so we will describe their semantics based on their intended usage. `canvas` represents a browser-canvas object which, amongst other things, contain information on the height l and width m of the screen to be rendered on. The actual screen is represented by `buf32`. To be specific, `buf32` is meant to be a one-dimensional array of length $l \cdot m$ containing the colors to be displayed on screen in the form of 32-bit unsigned integers.

The `Render()` function may conceptually be interpreted as generating a color-height pair (c, h) at each point in a $n \times m$-dimensioned discrete space G. The function is built around a double-nested for-loop which iterates over depth slices z_k in its outer range and iterates over screen columns j in its inner range. The outer-range iterates from $z_0 = 1$ to $z_k < d$ with a step-size starting at 1 and increasing by $\Delta = 0.005$ for each iteration:

```
var deltaz = 1.;

for (var z = 1; z < camera.distance; z += deltaz) {
    ...
    deltaz += 0.005;
}
```

Listing 2.4: Outer range of double-nested for-loop

Listing 2.4 may be thought of as implementing the skeleton of the front-to-back approach discussed in **Section 2.2.1**. The fact that an increasing step size is used, is simply one way of ensuring that the difference between succeeding depth-slices z_k and z_{k+1} nears infinity, as one approaches the horizon. By lowering Δ towards 0 one may limit the rate at which $|z_k - z_{k-1}|$ increases as a function of k. This, in turn, will result in more depth samples and hence a smoother rendered frame in general. More depth samples of course also means more data to be processed and hence an

increase in overall runtime. These issues are further explored in **Chapter 6**.

As explained in **Section 2.2.1**, one may think of depth-slice z_k as constituting a line $\overline{A_k B_k}$ between two points A_k and B_k on the boundary of the virtual observers field of view. The body of the outer-loop determines A_k and B_k relative to O (i.e. $\overrightarrow{OA_k}$ and $\overrightarrow{OB_k}$) for a given depth slice z_k using the logic encapsulated by **Equation 2.5** and **Equation 2.6** respectively:[2]

```
    var plx = -cosang * z - sinang * z;
    var ply = sinang * z - cosang * z;
    var prx = cosang * z - sinang * z;
    var pry = -sinang * z - cosang * z;
```

Listing 2.5: Determining A_k and B_k relative to O given a depth-slice z_k

Having determined A_k and B_k relative to O, the body of the outer-loop determines the segment size of $\overline{A_k B_k}$ by implementing **Equation 2.7**:

```
var dx = (prx - plx) / screenwidth;
var dy = (pry - ply) / screenwidth;
```

Listing 2.6: Determing the segment size of $\overline{A_k B_k}$

The absolute position of A_k is then found by incrementing (`plx`, `prx`) with O:

```
plx += camera.x;
ply += camera.y;
```

Listing 2.7: Determing absolute position of A_k

Now lets discuss the body of the inner for-loop in `Render()`. The first thing to note is that it ultimately increments (`plx`, `ply`) with (`dx`, `dy`):

```
for (var i = 0; i < screenwidth | 0; i = i + 1 | 0) {
    ...
    plx += dx;
    ply += dy;
}
```

Listing 2.8: Skeleton of inner for-loop

Using this approach the loop is able to process each segment $\overline{EF}_j$ of $\overline{A_k B_k}$ in sequence. Each $\overline{EF}_j$ has an associated color (c, h) which, as described in **Section 2.2.4** can be determined by applying a tiling function p to E_j and mapping the resulting (x', y') into C and H respectively. The inner-loop implements p via the following construct:

[2] **Listing 2.5** of the general case for vector rotation defined in **Equation 2.5** and **Equation 2.6**. See **Chapter C**.

```
var mapoffset = ((Math.floor(ply) & mapwidthperiod) << map.shift)
              + (Math.floor(plx) & mapheightperiod) | 0;
```

Listing 2.9: implementation of tiling function f

Listing 2.9 takes into account the fact that C and H are represented by one-dimensional arrays `map.color` and `map.altitude` using row-major order. The value `mapoffset` hence is a single offset rather than a pair of values (x', y'). The steps taken to compute `mapoffset` are threefold. First `ply` is floored and subsequently reduced modulo r by applying the bitwise AND operator to it and value `mapwidthperiod` $=$ `map.width` $- 1$[3]. The result is then shifted `map.shift` bits to the left, thereby producing a value γ, which corresponds to y'. Finally, `plx` is floored and reduced modulo r symmetrically and the result χ added to γ. `mapoffset` $= \gamma + \chi$ then corresponds to (x', y'). (c, h) can then be retrieved by indexing into `map.color` and `map.altitude` with `mapoffset`.

The next construct in the inner-loop consists of a sequence of operations which transform h to h' by accounting for vertical perspective as well as `camera.height` and `camera.horizon`:

```
var heightonscreen = (camera.height - map.altitude[mapoffset])
                   * invz + camera.horizon | 0;
```

Listing 2.10: Transforming h to h'

Listing 2.10 is more or less a straightforward adaptation of the steps described in Section 2.2.5 and Section 2.2.6. Noting that `invz` corresponds to $z_k^{-1} \cdot \beta$, the reader should be able to connect the dots themselves.

This concludes our discussion of how `Render()` generates a (c, h') pair at each point in a $n \times m$-dimensioned discrete space G.

Drawing on screen

Having determined (c, h') for an individual column j, the implementation has to write a line of color c to `buf32`, for that column. To handle the occlusion problem in the front-to-back method, an array called `hiddenY` with an index for each column is created. Each value in the array is initialized with the value of l, and indicates the last h' value, where a line was written to `buf32`.

```
function Render()
{
    ...
    var hiddeny = new Int32Array(screenwidth);
    for(var i=0; i<screendata.canvas.width|0; i=i+1|0)
        hiddeny[i] = screendata.canvas.height;
    ...
};
```

Listing 2.11: initialization of hiddenY

[3]This only makes sense because `map.width` $=$ `map.height` $=$ 1024 is a power of two

Once (c, h') has been determined in the inner `for`-loop, the function `DrawVerticalLine(x, ytop, ybottom, col)` is called to write the line, for column j to `buf32`. Here x is the column j, `ytop` is h', `ybottom` is the current value of `hiddenY` for column j and `col` is c.

Before the function writes anything to `buf32` the value of `ytop` is checked. First if the value of `ytop` is negative, it is set to zero, since a negative value, is a placement above the screen and would require writing to an index, outside the range of `buf32`. Second if the value of `ytop` is larger than `ybottom`, the function returns. This can occur if the value of `ytop` greater than l, or the line that has to be written is occluded, by a line that have already been written on a previous layer.

```
function DrawVerticalLine(x, ytop, ybottom, col)
{
    ...
    if (ytop < 0) ytop = 0;
    if (ytop > ybottom) return;
    ...
};
```

Listing 2.12: Setup of DrawVerticalLine

If the function does not return by the previous step, the line is written to `buf32`. The start position for `ytop` in `buf32` is first calculated and saved in `offset` by using `ytop`, `screenWidth` and x.

A `for`-loop is then used to write the value of `col` to the position of `offset` in `buf32`. At each iteration the position of `offset` is moved to the next position in the same column of `buf32`. This continuous until the position of `ybottom` is reached.

```
function DrawVerticalLine(x, ytop, ybottom, col)
{
    ...
    var offset = ((ytop * screenwidth) + x)|0;
    for (var k = ytop|0; k < ybottom|0; k=k+1|0)
    {
        buf32[offset|0] = col|0;
        offset = offset + screenwidth|0;
    }
};
```

Listing 2.13: Drawing the line

When the function returns, the last thing to do is to update the value of `hiddenY`, in the inner `for`-loop of `Render()`. This is done by simply checking if the value of `heightonscreen` is less than the value of `hiddenY` for the current column. If it is, the value of `hiddenY` is set to the value of `heightonscreen`.

```
function Render()
{
    ...
    if (heightonscreen < hiddeny[i]) hiddeny[i] = heightonscreen;
    ...
};
```

Listing 2.14: Updating the value of hiddenY

Chapter 3

Data Parallelism

3.1 General Idea

To understand data parallelism we must first understand the general principle of ***parallel computing***. In parallel computing, independent operations are carried out simultaneously, generally within different cores of a processing unit. Though closely related, it is important to distinguish it from ***concurrent computing***. In concurrent computing, scheduling is used to execute tasks during overlapping time periods, however the execution is not necessarily parallelized. An obvious example is the use of time slicing on a single-core system.

A common abstract model used in parallel computing is the ***Parallel Random Access Machine***(***PRAM***) [EG88, p. 234-245] .

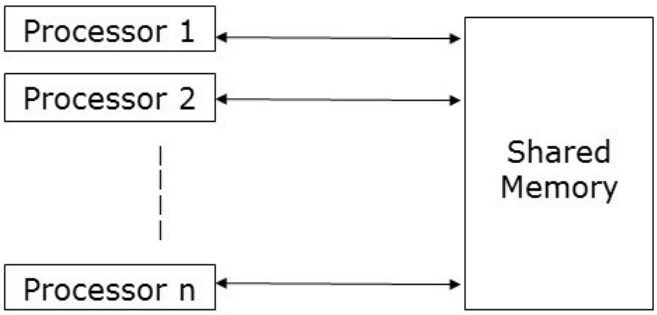


Figure 3.1: A diagram of a Parallel Random Access Machine. Source: tutorialspoint (https://www.tutorialspoint.com/parallel_algorithm/parallel_random_access_machines.htm)

The PRAM is the parallel equivalent of the RAM abstract machine used when reasoning about sequential algorithms. The model assumes several processing units are connected via a shared memory from which data can be read and written. The majority of algorithms we are interested in can function within a ***Exclusive-Read Exclusive-Write*** (***EREW***) PRAM, meaning a PRAM in which each processing unit has exclusive read and write access to each memory cell.

In any case, the PRAM neglects important technical details such as cache access, space limitations, communication overhead and heterogenous memory access. This makes mapping it to actual hardware complicated and hence reasoning about the runtime of parallel algorithms using this approach is not easy. We will later look at an alternative approach using ***Language Based Cost Models*** (**Section 5.1**), however for now the PRAM should suffice as an abstract model.

Given this model, we may introduce data parallelism by contrasting it with ***task parallelism***. Task parallelism is the simultaneous execution of different operations on different data, i.e. the simultaneous execution of different tasks, and thereby a potential form of concurrent programming with nondeterministic features. Data parallelism, on the other hand, is the simultaneous execution of the same operation on different data and is almost always deterministic and therefore easier to reason about. At the hardware side, task parallelism is usually employed with the ***Multiple Instruction Multiple Data*** (***MIMD***) architecture while data parallelism uses the ***Single Instruction Multiple Data*** (***SIMD***) architecture [Fly72].

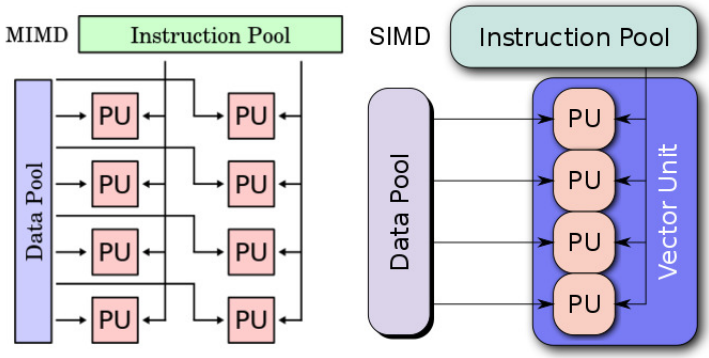


Figure 3.2: left: MIMD architecture. right : SIMD architecture. Source: Colin M.L. Burnett (https://en.wikipedia.org/wiki/User:Cburnett)

As can be inferred from **Figure 3.2** the MIMD architecture is more complex than the SIMD architecture and hence takes up more space on a chip. In terms of throughput, one is generally better off using the SIMD architecture, however its capabilities are limited in comparison. In a sense, the difference in these architectures highlights two different approaches to designing computer hardware: given a limited number of transistors, should these be spent on raw computational power or easy of programmability? On a CPU (Central Processing Unit) the focus is on the latter. In fact, CPUs are mostly task parallel units utilising MIMD (as well as many other exotic subunits) to simplify irregular code execution. GPUs (Graphics Processing Units) on the other hand focus on the former. These are data parallel units which use the SIMD architecture to increase throughput but are quite hard to program.

Intuitively the mechanics of a SIMD-based GPU are quite simple however. As shown on **Figure 3.2**, a set of values is represented as a vector unit which is operated over in parallel. With this setup it should be clear how data parallelism may help optimize a rendering algorithm such as Voxel Space. From what has been covered in **Chapter 2** we know significant parts of the pixel calculations in the algorithm are identical and independent of each other. By offloading such calculations to the GPU and executing them in parallel we could improve its overall runtime. This should come

as no surprise, as GPUs are primarily intended for optimizing the rendering of graphics. This analysis of course does not take into account the many finer points of the Voxel Space algorithm. However, before we can proceed with its actual parallelized implementation, (in **Chapter 4**) we need a framework for expressing such an implementation. This is the subject of the next section.

3.2 Programming in Parallel

As mentioned in the previous section, programming a GPU is complicated. In fact, it is complicated enough that mapping a data-parallel implementation of the Voxel Space algorithm directly to it is infeasible. What is needed is a high-level language able to express the intended functionality of the algorithm in such a way that a compiler can translate it to a low-level framework, such as OpenCL [Kae+15], which more or less directly mirrors its parallel target architecture. Several such high-level languages exists including the popular NumPy [20c].

NumPy features straightforward sequential and deterministic semantics while offering ***parallel potential***, by which we mean it can be parallelized by the right compiler, as long as the code is written in a parallel style[1]. NumPy code written in the parallel style usually involve performing bulk operations on entire arrays. As an example compare the following two functions for incrementing each value in an n-dimensional array by one:

```
import numpy as np

  def inc_seq(x):
    for i in range(len(x)):
      x[i] = x[i] + 1

  def inc_par(x):
    return x + np.ones(x.shape)
```

Listing 3.1: Sequential vs. parallel implementation of array incrementation in NumPy. Source: Adapted from "Deterministic parallel programming and data parallelism" [Hen19b, s. 23].

The first sequentially increments each value using a for-loop while the second takes a parallel approach by summing the input array `x` with an array of ones created via NumPy's built-in `np.ones()` function. Moreover the two approaches use different programming paradigms: imperative and functional respectively. This is no coincidence, as most NumPy code is ***purely functional***, meaning it has no side-effects.

Unfortunately, NumPy's ability to express parallelism is limited by it being a ***first order language***. Such languages are parametrized over simple values and hence only allow functions to do one thing. Previously mentioned `np.ones()` is a prime example of this. While it may compute values in parallel, it can only compute one type of value (specifically ones). This makes implementing more elaborate constructs, such as per-item control flow, cumbersome. Assume for example we wish to

[1] In practice NumPy is not usually parallel but rather implemented in efficient C and Fortran, however this a technical point not essential for our analysis

apply the following mathematical function to each element x in a n-dimensional array:

$$f(x) = \begin{cases} \sqrt{x} & \text{if } 0 < x \\ x & \text{otherwise} \end{cases} \tag{3.1}$$

A sequential implementation would look something like:

```
def apply_sqrt_seq(xs):
  xs = xs.copy()
  for i in range(len(xs)):
    if xs[i] > 0:
      xs[i] = np.sqrt(xs[i])
  return xs
```

Listing 3.2: Sequential implementation of **Equation 3.1** in NumPy. Source: Adapted from "Deterministic parallel programming and data parallelism" [Hen19b, s. 32].

Which is fairly neat (except for the initial copy operation). The same, however, cannot be said for the following implementation written in a parallel style:

```
def apply_sqrt_par(xs):
  xs_nonneg = xs >= 0
  xs_neg = xs < 0
  xs_zero_when_neg = xs * xs_nonneg.astype(int)
  xs_zero_when_nonneg = xs * xs_neg.astype(int)
  xs_sqrt_or_zero = np.sqrt(xs_zero_when_neg)
  return xs_sqrt_or_zero + xs_zero_when_nonneg
```

Listing 3.3: Parallel implementation of **Equation 3.1** in NumPy. Source: Adapted from "Deterministic parallel programming and data parallelism" [Hen19b, s. 34]

What is needed in cases like this is a ***higher-order parallel language***, which is parametrised over not only simple values but also functions. The next section presents such a language.

3.3 Futhark

Futhark [EHO20] [20a] is a purely functional programming language with parallel potential like NumPy. It supports openCL and CUDA compilers which can translate programs written in a parallel style into efficient GPU code, but is itself hardware-agnostic. As alluded to at the end of **Section 3.2**, it is higher-order, with functions treated as first-class citizens, i.e. allowed to be passed as arguments to other functions (with some caveats we will not get into here). This is relevant, as it allows for array programming with ***second-order array combinators***, or ***SOACs*** for short.

SOACs accept an array A along with other arguments including a function f, and then use these to transform A into some result, which may itself be an array. A simple example is Futharks `map` which accepts A of type $[]\alpha$ and some function $f : \alpha \to \beta$, and then produces a new array of type $[]\beta$ by applying f to each element in A. Because each application of f is independent, they can, in

principle, easily be executed in parallel on a SIMD architecture assuming A is mapped to a vector unit. This allows us to write data-parallel programs in a neat and efficient way. In fact, a parallel implementation of Equation 3.1 in Futhark requires only:

```
let apply_sqrt_par (xs : []f32) : []f32 =
  map (\x ->
         if x > 0
         then f32.sqrt x
         else x) xs
```

Listing 3.4: Parallel implementation of Equation 3.1 in Futhark

In this case `map` accepts an array `xs` of type `[]f32` (`f32` is a 32-bit floating point value) and an anonymous function f : `f32` $\rightarrow$ `f32` which is applied to each element in `xs`. Note how `map` works only on regular arrays containing values of a uniform type α. This is no coincidence. In fact all SOACs in Futhark have this restriction. However, α can itself be an array of type $[]\beta$, meaning multidimensional arrays are supported. Such multidimensional arrays can be created and manipulated in parallel as well because Futhark allows ***nested parallelism***. The basic idea is that a function f passed to a SOAC can itself use a SOAC. For instance, the following program uses a double-nested `map` to transpose a two-dimensional matrix in parallel:

```
let transpose [n][m] 't (xss: [n][m]t) : [m][n]t =
  map (\j ->
           map (\i -> xss[i, j]) (iota n)
           ) (iota m)
```

Listing 3.5: Transposing a two-dimensional matrix in Futhark

The `iota` function in Listing 3.5 accepts an integer n and then produces an array $A = [0, \ldots, n-1]$. It is an example of a ***first-order array combinator***, which always performs the same array transformation. Another useful first-order array combinator is `zip`, which accepts two arrays of type $[]\alpha$ and $[]\beta$ and then merges them into an array of type $[](\alpha, \beta)$. `unzip` performs the inverse operation.

Listing 3.5 also showcases a special feature of Futhark called ***size-annotations***. Size-annotations are language constructs which can be used to impose constraints on the shape of the parameters and result of functions. The `[n]` and `[m]` used in the signature of `transpose` indicate it accepts a two-dimensional array `xss` of shape $n \times m$ and returns a two-dimensional array of shape $m \times n$. The `t` is a type-annotation indicating `transpose` is a polymorphic function, meaning `xss` can contain elements of any (uniform) type `t`, in which case the array returned contains elements of type `t` as well.

Futhark becomes very powerful when you combine several different SOACs. One popular choice is combining `map` with `reduce`. `reduce` accepts an array A of type $[]\alpha$, a function $\oplus : \alpha \rightarrow \alpha \rightarrow \alpha$ acting as a binary operator and its neutral-element $0_\oplus$. It then produces a value of type α by folding $\oplus$ over each pair of elements in A starting with $(0_\oplus, A_0)$. The ***MapReduce*** programming paradigm can for example be used to compute the dot-product of two vectors in one line:

```
let dotprod [n] (xs: [n]f32) (ys: [n]f32) : f32 =
  reduce (+) 0 (map2 (*) xs ys)
```

Listing 3.6: Computing the dot-product of two vectors in Futhark

The `map2` in **Listing 3.6** is a variant of `map` which applies a function $f : \alpha \to \beta \to \gamma$ to each pair of (identically placed) elements in two arrays A and B of type $[]\alpha$ and $[]\beta$ respectively. Futhark also supports `map3`, `map4` and `map5` which behave symmetrically.

`scan` is somewhat similar to `reduce`. It accepts the same parameters but returns an array of type $[]\alpha$ with all the intermediate results of the fold. For both SOACs $\oplus$ is required to be an associative operator if the fold is to be executed in parallel. $(\oplus, 0_\oplus)$ is then said to be a ***monoid***. In **Section 5.1** we will look more closely at why this requirement is needed for parallel execution.

Though not technically a SOAC, `scatter` is also relevant to mention. It accepts array A of type $[]\alpha$, array B of type `[]int`, array C of type $[]\alpha$ and then updates A_i to v for each pair of (identically placed) elements (i, v) in B and C. In this way it allows in-place updates to be executed in parallel, which, in most cases, will be more efficient than filtering through a whole array (even if it is done in parallel as well). Note that B and C must have the same outer size and that the result is unspecified if B contains duplicates (i.e. several writes are performed to the same location). Moreover, because Futhark is a purely functional programming language, the original array A is required to not be used in any execution path following the in-place update in order to preserve referential transparency.

As a final remark it should be mentioned that Futhark being a purely functional language also means it is prohibited from reading input or writing output (as both are side effects). For this reason it cannot be used as a general-purpose language, but rather is intended to provide compiled library files which can be imported and used for perfomance sensitive parts of larger applications written in conventional languages. In fact Futhark supports compilation to both C and Python libaries in addition to openCL and CUDA executables.

Chapter 4

The Translation

4.1 Overview

Having introduced parallel programming via Futhark in **Chapter 3**, we are now at a point where we can develop an efficient implementation of the Voxel Space algorithm. We will be using an incremental approach, translating each logic step in the Javascript implementation one at a time. We encourage the reader to check the full implementation in **Listing A.2** for reference.

As explained in **Section 2.3**, the Javascript implementation basically consists of two parts. In the first part a double-nested for-loop generates a color-height pair at each point in a depth $\times$ width-shaped discrete space G using an imported color/height-map pair (C, H). In the second part a frame of height $\times$ width dimensions is rendered on screen by having a third for-loop draw a vertical line for each generated color-height pair.

Before we cover the translation of each of these parts, we need to establish how the various objects in the JavaScript implementation should be represented in Futhark. We recall three objects are used to represent camera, (C, H) and screen respectively. We do not need the screen object as it contains strictly browser-specific information. The two other objects can be represented in Futhark with records. The question remains how C and H themselves should be implemented. The JavaScript implementation uses one-dimensional arrays of size $q \cdot r$. We will use two-dimensional arrays of shape $q \times r$ as they are much easier to work with.

4.2 Generating Color-Height Pairs

For the first part we need some way of expressing a double-nested for-loop in a parallel style. Not surprisingly, this can be done with a double-nested `map`. However, the translation is complicated by the fact that the outer loop in the Javascript implementation uses a non-linear increment sequence starting at `deltaz` $=$ 1.0 and increasing with 0.005 for each iteration performed. This is an example of an arithmetic sequence (a_n) with first term $a_0 = 1$ and $\Delta = 0.005$ as the common difference. Because the looping variable `z` (representing the current depth-slice) also starts at 1.0 and is incremented with `deltaz` at the end of each iteration, its value z_k during the kth iteration

can be determined as the ***partial sum*** of (a_n) at $i = k - 1$:

$$\sum_{i=0}^{k-1} a_i = \sum_{i=0}^{k-1} (a_0 + i\Delta) = \frac{k}{2}(2a_0 + (k-1)\Delta) = \frac{k}{2}(2 + (k-1) \cdot 0.005) \tag{4.1}$$

Noting that the outer-loop exits when z_k is greater than or equal to the chosen rendering distance d, we can determine its number of iterations n by setting **Equation 4.1** equal to d, solving for k and then rounding down. Because $\Delta = 0.005 \neq 0$ and we are only interested in positive solutions, this gives us:

$$n = \left\lfloor \frac{\sqrt{(\Delta - 2a_0)^2 + 8\Delta d} - 2a_0 + \Delta}{2\Delta} \right\rfloor = \left\lfloor \frac{\sqrt{(-1.995)^2 + 0.04d} - 1.995}{0.010} \right\rfloor \tag{4.2}$$

Given **Equation 4.1** and **Equation 4.2**, we can determine the full sequence of z_ks using a range-`map` combination. For better flexibility we wrap the code in a `get_zs` function parametrised over both Δ, d and z_0:

```
1  let get_zs (delta : f32) (d: f32) (z_0 : f32) : []f32 =
2    let sqrt =  f32.sqrt ((delta-2*z_0)**2 + 8*delta*d)
3    let num = sqrt - 2*z_0 + delta
4    let div = 2*delta
5    let n = i32.f32 (num / div)
6    let is = map (\i -> f32.i32 i) (1...n)
7    in map (\i -> (i/2) * (2*z_0 + (i-1) * delta)) is
```

Listing 4.1: Function generating the full sequence of values z_k taken on by `z` in the outer-loop.

Note that line 5 in **Listing 4.1** takes advantage of the fact that type conversion from `f32` to `i32` in Futhark always rounds towards zero. Other than this the code should be self-explanatory. We can now implement the outer loop by passing the array `zs` returned by `get_zs` to a `map` along with an anonymous function f_1 performing the necessary logical steps in the body of the loop:

```
map (\z -> ... ) zs
```

Listing 4.2: Outer-loop implementation skeleton

The first of these logical steps is to compute the start-point $A = (x_0, y_0)$ and segment dimensions (dx, dy) of the horizontal line $\overline{AB}$ intersecting the visible space at the current depth-slice z_k. This can be done in Futhark using more or less the same language constructs as in the original JavaScript implementation. We will not go into further detail here but note that the relevant code is encapsulated in a `get_h_line` function which accepts a depth-slice `z`, a camera record `cam` and a screen-width `m` and then returns a `line` record containing the aforementioned start point and segment dimensions.

Along with this `line` record we also need a scaling variable `inv_z` for the heights at depth-slice z_k. Having computed this variable, using again the same method as in the JavaScript implementation, we have all we need to start the inner loop. The inner loop needs to iterate over each pixel p_j in

a horizontal line on a screen with width m, so we will use a `iota-map` combination to implement it:

```
map (\j -> ... ) (iota m)
```

Listing 4.3: Inner-loop implementation skeleton

The question is how the anonymous function f_2 in **Listing 4.3** should implement the body of the inner loop. We know the body initially computes the start-point $E = (x, y)$ of the segment $\overline{EF}$ in $\overline{AB}$ corresponding to the current pixel p_j by flooring (x_0, y_0) and ends by incrementing (x_0, y_0) with (dx, dy) so the process can be repeated during the next loop iteration. We cannot increment (x_0, y_0) with (dx, dy) in our translation because Futhark does not support mutable variables. However, we can emulate the increment operation by summing (x_0, y_0) and $j(dx, dy)$. This allows us to define a `get_seg_point` function which accepts a line record `l` along with an index `j` and then computes (x, y) as follows:

```
let get_seg_point (l : line) (j : i32) : (i32, i32) =
  let x = i32.f32 (l.x_0 + (f32.i32 j) * l.dx)
  let y = i32.f32 (l.y_0 + (f32.i32 j) * l.dy)
  in (x, y)
```

Listing 4.4: Function computing the start-point (x,y) of the segment $\overline{EF}$ in $\overline{AB}$ corresponding to pixel p_j.

The next logical step is to index C and H with (x, y) to retrieve its associated color $c_{x,y}$ and height $h_{x,y}$. Because we represent both C and H with two-dimensional arrays of shape $q \times r$ the operation is fairly simple. All we need to do to ensure proper tiling is reduce (x, y) modulo (r, q):

```
let seg_point_color = lsc.color[y%q,x%r]
let seg_point_height = lsc.altitude[y%q,x%r]
```

Listing 4.5: Indexing C and H with (x,y)

What remains is computing the relative on-screen height $h'_{x,y}$. This requires no real translation, however we have to take into account $h'_{x,y}$ may fall outside the range $(0 \ldots l-1)$. To solve this problem we use Futharks built-in `i32.max` and `i32.max` operators in conjunction:

```
let bounded_height = i32.min (l-1) (i32.max 0 (i32.f32
 relative_height))
```

Listing 4.6: Accounting for negative relative heights $h'_{x,y}$ using Futharks built-in `max` function

We return the color-height pair $(c_{x,y}, h'_{x,y})$ at the end of f_2. Consequently, the double-nested `map` will return a $n \times m$-shaped two-dimensional array of such pairs, i.e. one pair at each point in a depth $\times$ width-shaped discrete space G.

4.3 Rendering a Frame

Given G we wish to render a frame on screen. Doing this in a way that takes advantage of Futharks parallel potential requires a somewhat conceptually complicated approach. The main complication stems from having to account for occlusion when drawing vertical lines on screen. We know the JavaScript implementation solves this problem by processing each color-height pair in G in front-to-back/left-to-right order using a continually updated variable `hiddeny` to indicate down to which point vertical lines should be drawn. We would like to process as many color-height pairs as possible in parallel so we need a different solution.

The first thing to note is that color-height pair $g_{k,j} = (c, h')$ logically maps to a c-colored vertical line $\overline{PL}_c$ bounded by points $P = (h', j)$ and $L = (l - 1, j)$ on screen. This means there is a one-to-one correspondance between columns in G and on screen:

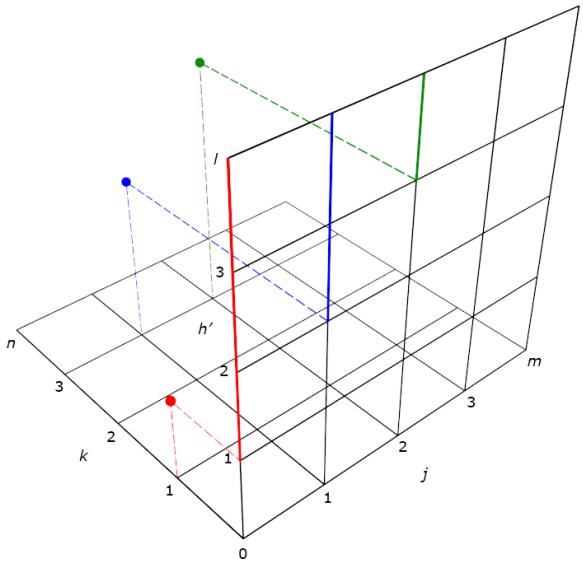


Figure 4.1: 1-1 correspondance between columns in G and columns on screen. Example (c, h') are for clarity visualized in terms of both color and height and their mapping to vertical lines $\overline{PL}_c$ on screen are drawn as well. Note that the screen has been vertically inverted.

It should be self-evident that vertical lines in different columns on screen cannot possibly occlude each (otherwise reassure yourself by consulting **Figure 4.1**). Hence it follows that occlusion only applies when processing color-height pairs from the same column in G. Because occlusion is the only source of logical dependency when rendering on screen, we can use a `map` to process each column of color-height pairs in G in parallel:

```
map (\j -> ... ) (transpose color_height_pairs)
```

Listing 4.7: Using a map to process each column of color-height pairs in G in parallel

The next question is how occlusion should be implemented for each column in G. Assume we

are processing column $g_{*,j}$ and that $\overline{P_k L}$ denotes the vertical line which $g_{k,j}$ maps to. Given $g_{1,j} = (c_1, h'_1)$ and $g_{2,j} = (c_2, h'_2)$ such that $h'_1 \leq h'_2$, we expect $\overline{P_1 L}_{c_1}$ to occlude all of $\overline{P_2 L}_{c_2}$. By extension, if $g_{3,j} = (c_3, h'_3)$ and $h_1 \leq h_3$, we expect $\overline{P_1 L}_{c_1}$ to occlude all of $\overline{P_3 L}_{c_3}$ as well.

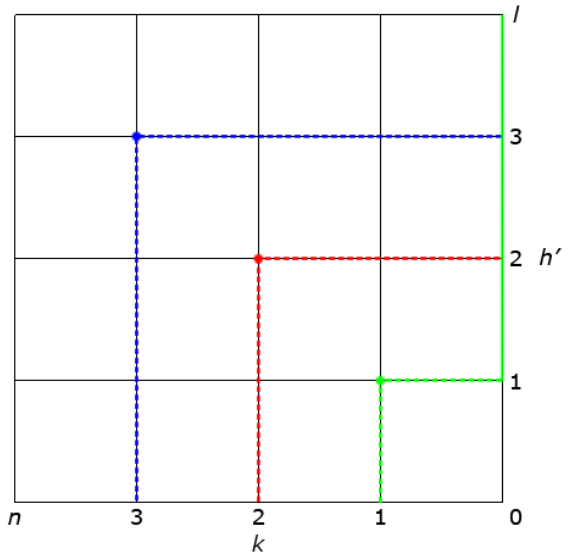


Figure 4.2: $\overline{P_1 L}_{c_1}$ occluding all of $\overline{P_2 L}_{c_2}$ and $\overline{P_3 L}_{c_3}$.

A simple way of representing this occlusion is by transforming $g_{2,j}$ and $g_{3,j}$ into $g'_{2,j} = (c_1, h'_1)$ and $g'_{3,j} = (c_1, h'_1)$ respectively:

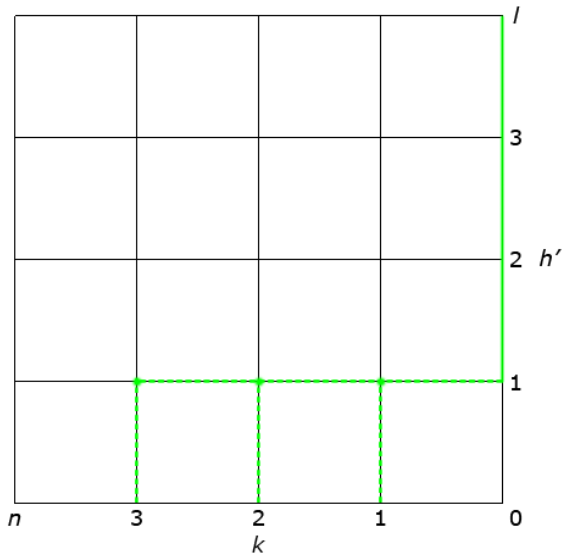


Figure 4.3: Representing occlusion of $\overline{P_2 L}_{c_2}$ and $\overline{P_3 L}_{c_3}$ by transforming $g_{2,j}$ and $g_{3,j}$ into $g'_{2,j}$ and $g'_{3,j}$.

With this approach we can perform occlusion over all $g_{k,j} \in g_{*,j}$ in a semi-parallel manner. The idea is to use Futharks `scan` in conjunction with an `occlude` operator:

```
1  let occlude (c_1 : i32, h_1 : i32 )
2              (c_2 : i32, h_2 : i32 ) : (i32, i32) =
3    if (h_1 <= h_2)
4    then (c_1, h_1)
5    else (c_2, h_2)
6  ...
7  let col_occluded = scan (occlude) (0, 1) j
```

Listing 4.8: `occlude` operator used in conjunction with `scan` to implement occlusion for a column $s_{*,j}$ in G

The definition of `occlude` in **Listing 4.8** should be more or less self-explanatory. Given $g_{k,j} = (c_k, h'_k)$ and $g_{k+1,j} = (c_{k+1}, h'_{k+1})$ in $g_{*,j}$, it implements occlusion on a local level by returning $g'_{k+1,j}$ if $h'_k \leq h'_{k+1}$ and $g_{k+1,j}$ otherwise. The subsequent `scan` application in line 7 propagates local occlusion by folding `occlude` over each $(g_{k,j}, g_{k+1,j}) \in g_{*,j}$ beginning with $(0_{\texttt{occlude}}, g_{0,j})$. Because all intermediate results of the fold are saved, it follows that `col_occluded[k]` will contain either $g'_{k-\epsilon,j} = (c_{k-\epsilon}, h'_{k-\epsilon})$, if $c_{k-\epsilon}$ should occlude c_k at $P = (h', j)$ on screen, or $g_{k,j}$ if not. As an example, **Figure 4.4** and **Figure 4.5** shows the occlusion mechanism applied to a $g_{*,j}$ column with a wide range of h' values.

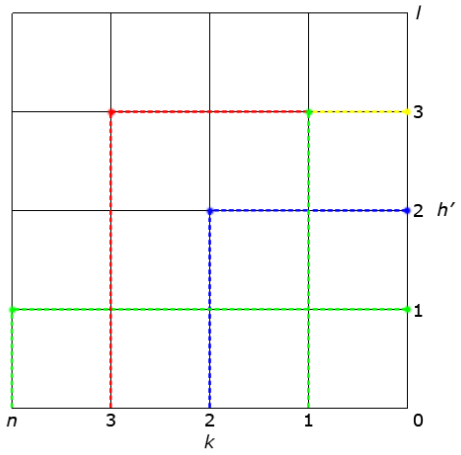


Figure 4.4: $g_{*,j}$ before occlusion.

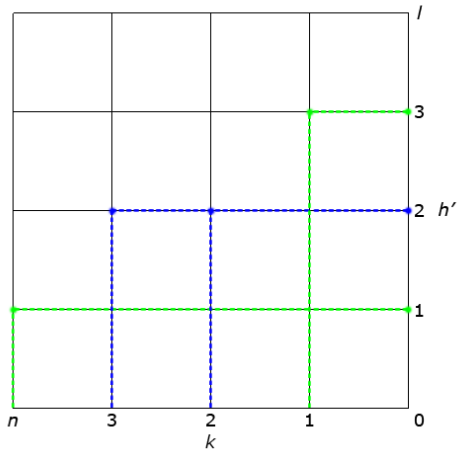


Figure 4.5: $g_{*,j}$ after occlusion.

The next logical step is to render c on screen at $P = (h', j)$ for each (c, h') in `col_occlude`. This can be done in parallel using Futharks `scatter`:

```
let init_col = replicate l lsc.sky_color
let (cs, hs) = unzip col_occluded
let screen_col = scatter init_col hs cs
```

Listing 4.9: Rendering in parallel c on screen at $P = (h', j)$ for each (c, h') in `col_occlude`.

The actions taken in **Listing 4.9** are twofold. First a `replicate` application is used to initialize the l-sized screen-column corresponding to `col_occluded`. The value $c_{\text{bg}} =$ `lsc.sky_color` used for initialization is the background color of the frame to be rendered. Next the enclosing `scatter` application is used to overwrite the screen column with c at those points $P = (h', j)$ for which there exists a (c, h') in `col_occluded`. In **Section 3.3** we mentioned that the result of a `scatter` application is unspecified if the array supplied to it as second argument contains duplicates. This is obviously very likely in this case as the color-height pairs in `col_occluded` have been generated via occlusion. The reason why the implementation works nevertheless is that the occlusion procedure ensures any two pairs (c_1, h_1) and (c_1, h_2) for which $h_1 = h_2$ must also have $c_1 = c_2$. This means arrays positions written to concurrently will have always end with the same value independent of how the concurrent writes are interleaved.

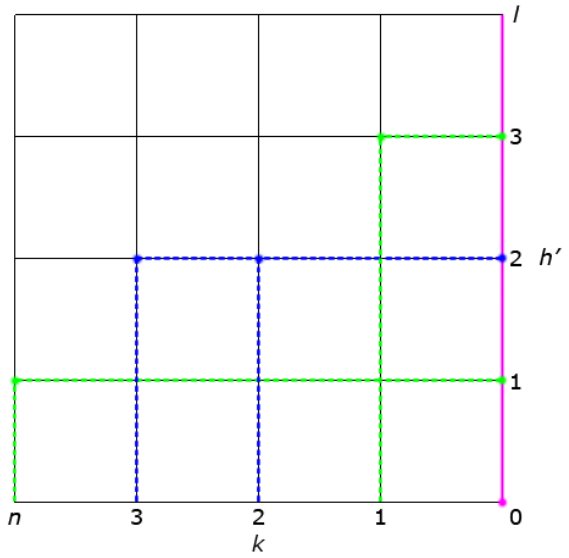


Figure 4.6: Rendering c on screen at $P = (h', j)$ for each (c, h') in **Figure 4.5** using the approach implemented in **Listing 4.9**. Note that the purple background color is drawn as a line (rather than a single point at the bottom of the screen) for illustrative purposes.

To complete the screen column we need to extend each c rendered on it with its associated vertical line $\overline{PL_c}$. Taking occlusion into account, it follows that each $\overline{PL_c}$ should cover all consecutive background color below its origin P but extend no further. Put another way, what we want to avoid is one vertical line $\overline{P_k L_c}$ occluding another vertical line $\overline{P_{k+\epsilon} L_c}$ with origin $P_{k+\epsilon}$ below P_k. The solution once again is to use `scan`, this time in conjunction with a `fill` operator:

```
let fill (c_1 : i32) (c_2 : i32) : i32 =
  if (c_2 == lsc.sky_color)
```

```
    then c_1
    else c_2
  ...
in scan (fill) lsc.sky_color screen_col
```

Listing 4.10: Using `scan` in conjunction with a `fill` operator to fill in the missing colors in `screen_col`.

The `scan` application in **Listing 4.10** folds the `fill` operator over each (c_k, c_{k+1}) in `screen_col` starting with (c_{bg}, c_0) at the top. Because the `fill` operator replaces c_{k+1} with c_k if and only if $c_{k+1} = c_{\text{bg}}$, it follows that each c_k in effect will extend to cover all consecutive background color below it. The value returned by the `scan` application must then necessarily be the fully rendered column at $(*, j)$ on screen. **Figure 4.7** provides an illustration.

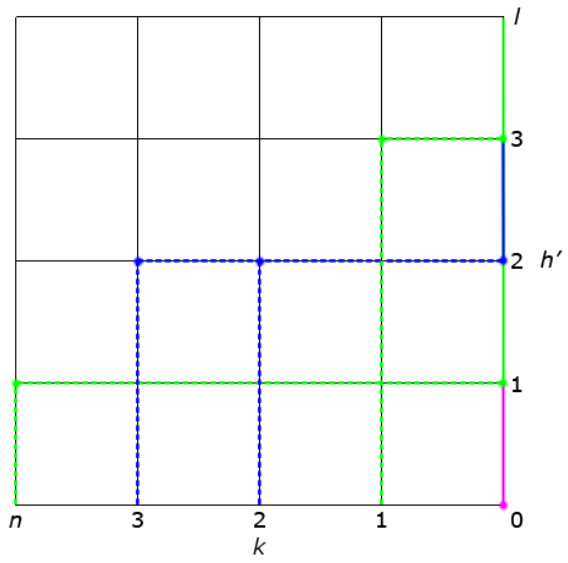


Figure 4.7: Completing the screen column in **Figure 4.6**. Lines are again drawn only for illustrative purposes (As there are no background-colored points to cover).

Consequently, the outer `map` will return the fully rendered frame on screen. The final step in our translation is to transpose the rendered frame from its current m × l shape to its intended l × m shape:

```
  ...
in transpose rendered_frame
```

Listing 4.11: Transposing the rendered frame from its initial $m \times l$ shape to its intended $l \times m$ shape.

Chapter 5

Implementation Analysis

5.1 Establishing a Cost Model

In Chapter 4 we covered in-depth a parallelizable implementation of the Voxel Space render algorithm written in Futhark. However we did not provide any actual run-time analysis. The first question is which framework should be used for such an analysis. One option would be to measure run-time in terms of instruction cycles given some abstract machine model such as the PRAM. As alluded to in Section 3.1, this approach is complicated because abstract machine models usually neglect important low-level details such as concurrency issues. One could of course incorporate these low-level details into ones model but this would not be a very portable solution. An alternative is to abandon machine representation altogether and instead use a language-based cost model, which defines run-time directly in terms of higher-level programming language constructs. With this approach programmers are presented with an interface of cost guarantees which the language implementor is responsible for ensuring across different machines. Hence performance reasoning does not need to be "ported" when targeting new machines.

5.2 Work and Span

For our analysis we will be using a variant of the language-based cost model developed by Guy Blelloch in connection with his work on NESL [Ble96]. Central to this cost model are the concepts of ***work*** and ***span***[1]. Work is defined as the total number of operations executed by a computation while span is defined as the longest chain of sequential dependencies in the computation. One way of visualizing the work and span of a computation is by using ***Dependency DAGs (Directed Acyclic Graphs)***. `inc_seq()` from Listing 3.1 for example has the following dependency DAG:

[1] 1Sometimes also called *depth*

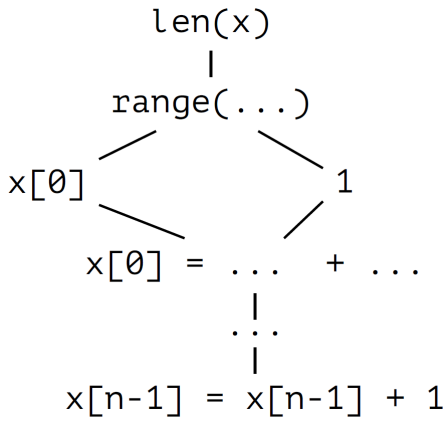


Figure 5.1: Dependency DAG for `inc_seq()` from **Listing 3.1**. Source: "Cost models and advanced Futhark programming" [Hen19a, s. 5].

Figure 5.1 is to be read from the top down. Each node corresponds to an operation q executed by `inc_seq()`. Nodes representing independent operations are placed on the same level. The children of a node q_i are to be interpreted as those operations immediately dependent on q_i. Given this setup, the work $W(p)$ of the program p implemented by `inc_seq()` is equal to the total number of nodes while its span $S(p)$ is equal to the longest path from root to a leaf. We note that $W(p) = O(n)$ and $S(p) = O(n)$. This tells us the total number of operations and longest chain of sequential dependencies in p both grow linearly with its input size. Put another way, p will always have to execute $O(n)$ operations in sequence, even if run on more than one core. This makes intuitive sense as its use of a for-loop inherently makes `inc_seq()` a sequential implementation. Now lets look at the dependency DAG for `inc_par()` from **Listing 3.1**:

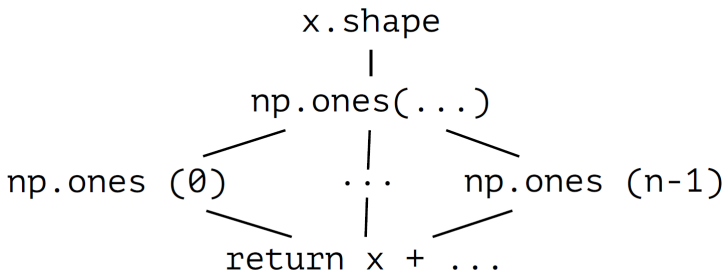


Figure 5.2: Dependency DAG for `inc_par()` from **Listing 3.1**. Source: "Cost models and advanced Futhark programming" [Hen19a, s. 6].

In this case $W(p)$ is still $O(n)$ however $S(p)$ is reduced to $O(1)$ because the individual initialization steps of `np.ones()` as well as the subsequent addition operations are internally independent. Each of these independent operations could, at least in principle, be executed in parallel if p were to be run on sufficiently many cores. With this in mind we would expect the runtime of p to be (more or less) constant as long as the number of cores it is run on scales appropriately with its input size.

Generally speaking, work and span can be seen as measures of the runtime of a computation at two extremes: the runtime T_1 when executing sequentially on a single core, and the runtime T_∞ when executing in parallel on an unlimited number of cores. In practice we want to know the runtime for some fixed number of cores. ***Brent's Theorem*** [Ble96, p. 89], which we will not prove here [2], shows that if a computation has work T_1 and span T_∞ then its runtime T_i when executing on i cores is:

$$\frac{T_1}{i} \leq T_i \leq \frac{T_1}{i} + T_\infty \tag{5.1}$$

Equation 5.1 is important because it tells us that the upper bound on T_i is proportional to T_∞ with an overhead of $\frac{T_1}{i}$ corresponding to the amount of missing hardware parallelism. Hence it makes sense to use span as a runtime measurement for programs running on massively parallel hardware such as GPUs (where i is very large). This, however, does not mean work is irrelevant in terms of runtime analysis. For one we cannot assume a parallelizable implementation will always be able to run on enough cores to make $\frac{T_1}{i}$ insignificant. Moreover, hidden factors related to communication may, even in the best case, prevent full utilization of its parallel potential. Considering this, it is not surprising that a parallelizable implementation with great span might, under certain circumstances, be outperformed by a sequential implementation with better work. This of course is undesirable. What we want ideally are implementations such as `inc_par()` which improve upon the span of their sequential equivalent while preserving the same work (within a constant factor). Such implementations are said to be ***work-efficient***.

5.3 A Cost Model for Futhark

So far we have analyzed programs in terms of work and span by tallying the levels of their dependency DAGs. The language-based cost model of Blelloch formalizes this procedure by associating specific work and span costs with primitive operations and specifying rules for combining these across program expressions [Ble96, p. 87-88]. Not surprisingly, the general rule is that the total work, when evaluating a set of expressions in parallel, equals the sum of the work of the expressions, while the total span equals the maximum of the span of the expressions. The cost model we will be using adopts a similar approach. Its (partially) complete specification is provided by Elsman et al. in *Parallel Programming in Futhark* [EHO20, p. 67-68] and on the official Futhark website [20b]. We will, for the sake of brevity, explain only the rules most relevant to the implementation from **Chapter 4**.

We start with the base rule which states that literals have constant work and span:

$$W(v) = S(v) = O(1) \tag{5.2}$$

This should not need further explaining. We next note that the work and span of an operator application is linear in the work and span of its operands with an overhead corresponding to the cost of the actual operation performed:

$$W(e_1 \oplus e_2) = W(e_1) + W(e_2) + O(1), \tag{5.3}$$

[2] The theorem also makes several assumptions about communication costs, i.e. latency and bandwidth. We note that latency can be accounted for by modifying **Equation 5.1** while bandwidth is less of a problem on newer machines [Ble96, p. 89-90]. However we wont go into further detail.

$$S(e_1 \oplus e_2) = S(e_1) \ + S(e_2) \ + O(1) \tag{5.4}$$

Equation 5.4 is important because it tells us that operator application is not fully parallelizable in Futhark. Now let $[\![e]\!]$ denote the result of evaluating e and let $e'[[\![e]\!]/x]$ denote e' after substituting each occurrence of x with $[\![e]\!]$. We then have the following rules with respect to let-expressions:

$$W\big(\texttt{let}\ x = e\ \texttt{in}\ e'\big) = W(e) + W\big(e'[[\![e]\!]/x]\big) + O(1), \tag{5.5}$$

$$S\big(\texttt{let}\ x = e\ \texttt{in}\ e'\big) = S(e) \ + S\big(e'[[\![e]\!]/x]\big) \ + O(1) \tag{5.6}$$

We can infer from **Equation 5.6** that let-expressions are not fully parallelizable either. This makes intuitive sense as the body of a let-expression is usually dependent on the variable that the let-expression assigns a value to initially. Function application is analogous because the evaluation of its body usually depends on the value of its arguments. The evaluation of these arguments is itself not fully parallelizable as Futhark treats them similar to operands. Assuming f is a function `\`$x_1 \ldots x_n$ `->` e applied to arguments e_1 through e_n, we arrive at the following rules for its work and span:

$$W(f\ e_1 \ldots e_n) = W(e_1) + \ldots + W(e_n) + W(e[[\![e_1]\!]/x_1, \ldots, [\![e_n]\!]/x_n]) \tag{5.7}$$

$$S(f\ e_1 \ldots e_n) = S(e_1) \ + \ldots + S(e_n) \ + S(e[[\![e_1]\!]/x_1, \ldots, [\![e_n]\!]/x_n]) \tag{5.8}$$

The constructs that, in general, are most parallelizable in Futhark are array combinators. The first-order variety includes `iota` which allows for parallel array initialization. Its work and span are:

$$W(\texttt{iota}\ e) = W(e) + O([\![e]\!]) \tag{5.9}$$

$$S(\texttt{iota}\ e) = S(e) \ + O(1) \tag{5.10}$$

The constant factor in **Equation 5.10** is a direct indication that each element in the array produced by `iota` can, in fact, be constructed in parallel. We assume the same holds true for `replicate`, as it performs array initialization in a semantically similar manner. We cannot, however, verify this as the specification does not actually list its work and span. The specification, in fact, does not list the work and span of any first-order array combinator besides `iota`. What we can do is derive these rules ourselves based on the implementation-model presented by Henriksen in his Futhark blog [Hen19c]. We will not explain the derivation here[3], but note that the rules relating to `replicate`, `unzip` and `transpose` are:

$$W(\texttt{replicate}\ e_1\ e_2) = W(e_1) + W(e_2) \quad + O([\![e_1]\!]) \tag{5.11}$$

$$S(\texttt{replicate}\ e_1\ e_2) = S(e_1) \ + S(e_2) \quad + O(1) \tag{5.12}$$

$$W(\texttt{unzip}\ e) = W(e) \ + O(n), \quad [\![e]\!] = [(a_1, b_1), \ldots, (a_n, b_n)] \tag{5.13}$$

$$S(\texttt{unzip}\ e) = S(e) \ + O(1), \quad [\![e]\!] = [(a_1, b_1), \ldots, (a_n, b_n)] \tag{5.14}$$

$$W(\texttt{transpose}\ e) = W(e) \ + O(nm), \quad [\![e]\!] = [a_1, \ldots, a_n][b_1, \ldots, b_m] \tag{5.15}$$

$$S(\texttt{transpose}\ e) = S(e) \ + O(1), \quad [\![e]\!] = [a_1, \ldots, a_n][b_1, \ldots, b_m] \tag{5.16}$$

The main SOAC relevant for our analysis is `map`. We learned in **Section 3.3** that its function argument can, in principle, be applied to each element in its array argument in parallel. The following pair of rules confirm Futhark supports this type of execution:

$$W(\texttt{map}\ f\ e) = W(e) + W\big(e'[v_1/x]\big) + , \ldots, + W\big(e'[v_n/x]\big), \tag{5.17}$$

[3] see **Section B.1** for more information

$$
\begin{aligned}
&[\![f]\!] = \backslash x \text{ -> } e', \quad [\![e]\!] = [v_1, \ldots, v_n] \\
S(\texttt{map } f\ e) \ &= S(e) + \max\big(S\big(e'[v_1/x]\big)+, \ldots, +S\big(e'[v_n/x]\big)\big) + O(1), \qquad (5.18) \\
&[\![f]\!] = \backslash x \text{ -> } e', \quad [\![e]\!] = [v_1, \ldots, v_n]
\end{aligned}
$$

Equation 5.17 and **Equation 5.18** should remind the reader of the general rule explained earlier. The parallel expressions in this case are the applications of function f. If we ignore the costs associated with evaluating array e, then the total work equals the sum of the work of these function applications while the total span equals the maximum of their span (plus a constant factor accounting for additional overhead). The model used by Futhark for evaluating instances of `scatter` is similar in the sense that each of the array updates to be performed can be executed in parallel. Hence the work and span of `scatter` are:

$$
\begin{aligned}
W(\texttt{scatter } e_1\ e_2\ e_3) &= W(e_1) + W(e_2) + W(e_3) + O(n), \qquad (5.19) \\
&\quad [\![e_1]\!] = [a_1, \ldots, a_m], \quad [\![e_2]\!] = [i_1, \ldots, i_n], \\
&\quad [\![e_3]\!] = [v_1, \ldots, v_n] \\
S(\texttt{scatter } e_1\ e_2\ e_3) &= S(e_1) + S(e_2) + S(e_3) + O(1), \qquad (5.20) \\
&\quad [\![e_1]\!] = [a_1, \ldots, a_m], \quad [\![e_2]\!] = [i_1, \ldots, i_n], \\
&\quad [\![e_3]\!] = [v_1, \ldots, v_n]
\end{aligned}
$$

The work and span of `scan` require a more thorough explanation. We will base our reasoning on the work-efficient implementation developed by Blelloch in *Prefix Sums and Their Applications* [Ble90] as Futhark's implementation guarantees the same work and span. It is, however, important to note that the two implementations are not necessarily identical.

Before the work-efficient implementation of `scan` can be explained we first need to understand how to parallelize `reduce`. Lets assume we want to reduce n values with the $+$ operator. Lets also assume that the values are the leaves in some balanced binary tree. We can then construct the nodes in the level above by summing each pair of leaves in parallel. By repeating the procedure at each level we end with a value at the root which is the total sum of the values in the leaves:

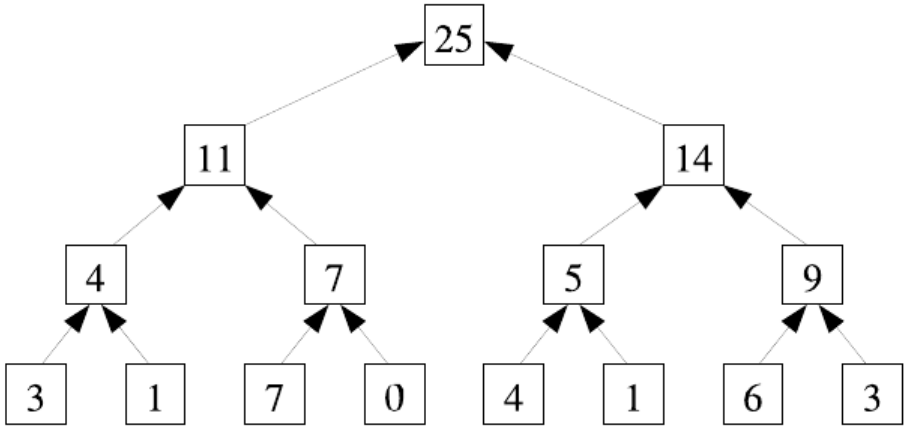


Figure 5.3: Summing n values in a semi-parallel manner using a balanced binary tree. Source: *Prefix Sums and Their Applications* [Ble90, p. 40].

The final tree, as illustrated by **Figure 5.3**, contains $O(n)$ nodes and has a height $h = O(\log(n))$. If we think of this tree as a dependency DAG, it follows that the implementation method has work and span equal to $O(n)$ and $O(\log(n))$ respectively. The correctness of the method is predicated on $+$ being an associative operator. The reason for this is that the method performs operator application in non-sequential order. Generally speaking, the method can be used whenever `reduce` is applied to an operator $\oplus$ which, along with its neutral element $0_\oplus$, constitute a monoid. With this in mind, it is not surprising that the work and span of `reduce` in Futhark are:

$$W(\texttt{reduce}\ \oplus\ e_1\ e_2) = W(e_1) + W(e_2) + W(e[v_1/x_1, v_2/x_2]) \cdot n + O(1) \tag{5.21}$$
$$[\![\oplus]\!] = \backslash x_1\ x_2\ \texttt{->}\ e, \quad [\![e_1]\!] = 0_\oplus,$$
$$[\![e_2]\!] = [v_1, \ldots, v_n]$$
$$S(\texttt{reduce}\ \oplus\ e_1\ e_2) = S(e_1) + S(e_2) + S(e[v_1/x_1, v_2/x_2]) \cdot \log(n) + O(1) \tag{5.22}$$
$$[\![\oplus]\!] = \backslash x_1\ x_2\ \texttt{->}\ e, \quad [\![e_1]\!] = 0_\oplus,$$
$$[\![e_2]\!] = [v_1, \ldots, v_n]$$

The tree in **Figure 5.3** contain many partial sums. It turns out that these partial sums can be used to generate all prefix sums of the n-sized array of input values. This requires performing another sweep of the binary tree this time starting at the root and going down to the leaves. Initially the neutral element 0 is inserted at the root. At each subsequent step the nodes at the current level passes to their left child their own value and passes to their right child $+$ applied to the value from their left child and their own value. The result is a binary tree in which the nth leaf (counting from left to right) equals the $n-1$th prefix sum of the initial array of values:

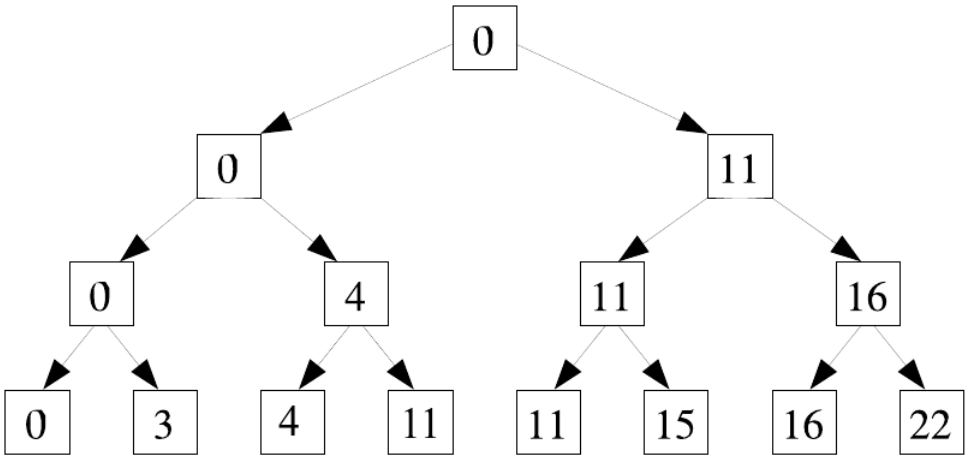


Figure 5.4: The binary tree produced by performing a downsweep of **Figure 5.3**. Source: Adapted from *Prefix Sums and Their Applications* [Ble90, p. 42].

The full array of prefix sums can then be obtained by extracting the leaves from **Figure 5.4**, appending the root from **Figure 5.3** and shifting one position to the right. A symmetrical procedure can be used for any other monoid. To understand why this works in general we will be relying on the following theorem:

THEOREM 5.1

After a complete down-sweep, each vertex of the tree contains the sum of all the leaf values that precede it.

Since the leaf values that precede any leaf are the values to the left of it in the input array, it follows from **THEOREM 5.1** that the leaves after a downsweep must constitute the full prescan of the input array. This prescan can then be extended to a scan using the method described above. We will not actually provide a proof of **THEOREM 5.1** but note that it can be found in *Prefix Sums and Their Applications* [Ble90, p. 41-43].

It should be obvious that nodes at the same level can be processed in parallel during a downsweep as they are only logically dependent on their own value and the value of their children. Because the binary tree produced by the downsweep contains the same number of nodes and has the same height as the binary tree produced by the initial upsweep, it stands to reason that the work and span of `scan` must be identical (within a constant factor) to that of `reduce`. This, in fact, is the case in Futhark as the following rules confirm:

$$W(\texttt{scan}\ \oplus\ e_1\ e_2) = W(e_1) + W(e_2) + W(e[v_1/x_1, v_2/x_2]) \cdot n + O(1) \tag{5.23}$$
$$[\![\oplus]\!] = \backslash x_1\ x_2\ \texttt{->}\ e, \quad [\![e_1]\!] = 0_\oplus,$$
$$[\![e_2]\!] = [v_1, \ldots, v_n]$$

$$S(\texttt{scan}\ \oplus\ e_1\ e_2) = S(e_1) + S(e_2) + S(e[v_1/x_1, v_2/x_2]) \cdot \log(n) + O(1) \tag{5.24}$$
$$[\![\oplus]\!] = \backslash x_1\ x_2\ \texttt{->}\ e, \quad [\![e_1]\!] = 0_\oplus,$$
$$[\![e_2]\!] = [v_1, \ldots, v_n]$$

This concludes our review of the cost model of Futhark.

5.4 Intermezzo – Correctness of Implementation

Before we can proceed with a runtime analysis of our implementation from **Chapter 4** we need to prove it is correct. We will not concern ourselves with actual testing here, as the interactive application available at the *FutSpace* repository [Kri+20] should be sufficient to convince the reader that the implementation does indeed work as intended. Whats more interesting is proving formally that the implementation produces the same result when run sequentially and in parallel. As explained in **Section 5.3**, this essentially boils down to proving that all instantiations of `scan` are applied to monoids only.

The first instantiation of `scan` in our implementation is applied to $(\texttt{occlude}, (0, l))$. Proving $(\texttt{occlude}, (0, l))$ is a monoid entails proving $(0, l)$ is the neutral element of `occlude`, and that `occlude` is an associative operator. First let $G_+ = G \cup \{(0, l)\}$. Then `occlude` can formally be defined as a mapping from $G_+ \times G_+$ to G_+:

$$\texttt{occlude} : G_+ \times G_+ \to G_+ \tag{5.25}$$

Because all generated heights h' in G are less than l it follows that:

$$\forall (c, h') \in G_+ : \\ (0, l)\ \texttt{occlude}\ (c, h') = (c, h') = (c, h')\ \texttt{occlude}\ (0, l) \tag{5.26}$$

Hence $(0, l)$ must be the neutral-element of `occlude`[4].

For `occlude` to be an associative operator it must satisfy:

$$\begin{aligned} &\forall\big((c_1, h_1'), (c_2, h_2'), (c_3, h_3')\big) \in G_+ \times G_+ \times G_+ : \\ &\big((c_1, h_1') \texttt{ occlude } (c_2, h_2')\big) \texttt{ occlude } (c_3, h_3') \\ &\qquad = \\ &(c_1, h_1') \texttt{ occlude } \big((c_2, h_2') \texttt{ occlude } (c_3, h_3')\big) \end{aligned} \tag{5.27}$$

As **Figure 5.5** illustrates, there are six distinct classes of values that can be taken by h_1', h_2' and h_3':

$$\begin{aligned} &1 : h_1' \le h_2' \le h_3', \qquad 2 : h_1' \le h_3' < h_2', \qquad 3 : h_2' < h_1' \le h_3', \\ &4 : h_2' \le h_3 < h_1', \qquad 5 : h_3' < h_1' \le h_2', \qquad 6 : h_3' < h_2' < h_1' \end{aligned} \tag{5.28}$$

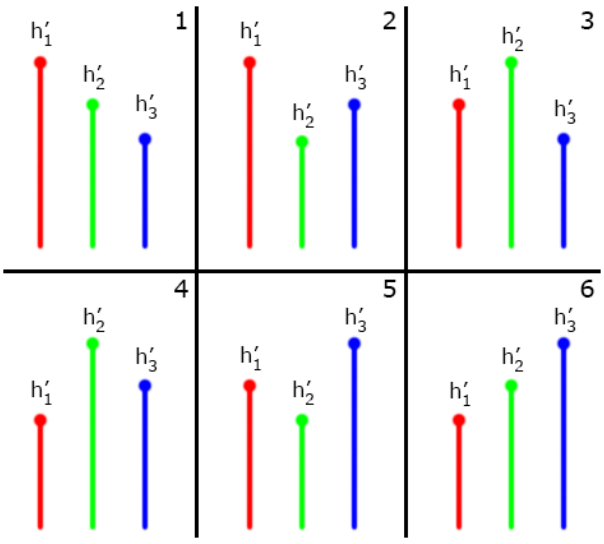


Figure 5.5: Visualization of the six classes of values that can be taken by h_1', h_2' and h_3'.

We need to prove **Equation 5.27** holds for each of these cases. This can be done by straightforward evaluation. For example, for the first case we have:

$$\begin{aligned} \big((c_1, h_1') \texttt{ occlude } (c_2, h_2')\big) \texttt{ occlude } (c_3, h_3') &= (c_1, h_1') \texttt{ occlude } (c_3, h_3') \\ &= (c_1, h_1')\,, \\ (c_1, h_1') \texttt{ occlude } \big((c_2, h_2') \texttt{ occlude } (c_3, h_3')\big) &= (c_1, h_1') \texttt{ occlude } (c_2, h_2') \\ &= (c_1, h_1') \end{aligned}$$

[4] A somewhat subtle point is that **Equation 5.26** must hold for $(0, l)$ itself. It should be obvious that this is the case because $(0, l)$ `occlude` $(0, l) = (0, l)$.

Similarly, for the second case it holds that

$$\begin{aligned}\big((c_1, h_1')\ \texttt{occlude}\ (c_2, h_2')\big)\ \texttt{occlude}\ (c_3, h_3') &= (c_1, h_1')\ \texttt{occlude}\ (c_3, h_3') \\ &= (c_1, h_1')\ , \\ (c_1, h_1')\ \texttt{occlude}\ \big((c_2, h_2')\ \texttt{occlude}\ (c_3, h_3')\big) &= (c_1, h_1')\ \texttt{occlude}\ (c_3, h_3') \\ &= (c_1, h_1')\end{aligned}$$

We will not go into further detail with the remaining cases as their proofs are symmetrical[5].

We next need to prove $(c_{\text{bg}}, \texttt{fill})$ is a monoid as well. First let $C_+ = C \cup \{c_{\text{bg}}\}$. Then `fill` is formally a function mapping from $C_+ \times C_+$ to C_+:

$$\texttt{fill} : C_+ \times C_+ \to C_+ \tag{5.29}$$

Because `fill` returns its second argument when it is different from c_{bg} and its first argument otherwise, it follows that:

$$\forall c \in C_+ : c\ \texttt{fill}\ c_{\text{bg}} = c = c_{\text{bg}}\ \texttt{fill}\ c \tag{5.30}$$

Therefore c_{bg} must be the neutral element of `fill`.

For `fill` to be an associative operator it must satisfy:

$$\begin{gathered}\forall (c_1, c_2, c_3) \in C_+ \times C_+ \times C_+ : \\ (c_1\ \texttt{fill}\ c_2)\ \texttt{fill}\ c_3 = c_1\ \texttt{fill}\ (c_2\ \texttt{fill}\ c_3)\end{gathered} \tag{5.31}$$

To prove **Equation 5.31** we will be using the same approach we took when proving **Equation 5.27**. Noting that the result of `occlude` depends only on its second argument it follows that there are exactly four cases to cover:

$$\begin{aligned}1 &: c_2 \neq c_{\text{bg}} \wedge c_3 \neq c_{\text{bg}} \quad 2 : c_2 = c_{\text{bg}} \wedge c_3 \neq c_{\text{bg}} \\ 3 &: c_2 \neq c_{\text{bg}} \wedge c_3 = c_{\text{bg}} \quad 4 : c_2 = c_{\text{bg}} \wedge c_3 = c_{\text{bg}}\end{aligned} \tag{5.32}$$

For the first case we have:

$$\begin{aligned}(c_1\ \texttt{fill}\ c_2)\ \texttt{fill}\ c_3 = c_2\ \texttt{fill}\ c_3 = c_3\ , \\ c_1\ \texttt{fill}\ (c_2\ \texttt{fill}\ c_3) = c_1\ \texttt{fill}\ c_3 = c_3\end{aligned}$$

Similarly for the second we have:

$$\begin{aligned}(c_1\ \texttt{fill}\ c_2)\ \texttt{fill}\ c_3 = c_1\ \texttt{fill}\ c_3 = c_3\ , \\ c_1\ \texttt{fill}\ (c_2\ \texttt{fill}\ c_3) = c_1\ \texttt{fill}\ c_3 = c_3\end{aligned}$$

Once again we will not go into further detail with the remaining cases as their proofs are symmetrical.[6]

[5] See **Figure B.1** for the full associativity proof
[6] See **Figure B.2** for the full associativity proof

5.5 Applying the Cost Model

We are now ready to apply the cost model introduced in Section 5.3 to our implementation from Chapter 4. We will once again use an incremental approach, covering each logical step in the implementation in turn.

The first logical step is determining the depth-slices which need to be iterated through. This is done with the `get_zs` function which performs a sequence of constant-time arithmetic operations followed by two `map` applications. The first `map` application is supplied a $(1 \ldots n)$ range and a constant-time function implementing integer conversion. The resulting array is supplied to the second `map` application along with another constant-time function implementing the arithmetic sequence formula. We can infer that the total work and span resulting from applying `get_zs` must be:

$$W(\texttt{get\_zs}\ \Delta\ d\ z_0) = O(n)$$
$$S(\texttt{get\_zs}\ \Delta\ d\ z_0) = O(1)$$

The next logical step is determining color-height pairs at each point in G. This is done with a double-nested `map` which uses the n-sized array of depth-slices returned by `get_zs` as outer range and `(iota m)` as inner range. The double-nested `map` performs a sequence of constant-time arithmetic operations and array indexing intermixed with calls to `get_h_line` and `get_segment_point`. These functions similarly perform a sequence of constant-time arithmetic operations. Hence it follows that the total work and span of the double-nested `map` is:

$$W(\texttt{color\_height\_pairs}) = O(nm)^{7}$$
$$S(\texttt{color\_height\_pairs}) = O(1)$$

The double-nested `map` returns a two-dimensional array of color-height pairs. The implementation renders a frame on screen by transposing this array and supplying the result to a `map`-application along with a function f_2 which, given a column $s_{*,j}$ in G renders the corresponding column on screen. Because the shape of the two-dimensional array is $n \times m$, the work and span of its transposition is:

$$W(\texttt{transpose color\_height\_pairs}) = O(nm)$$
$$S(\texttt{transpose color\_height\_pairs}) = O(1)$$

f_2 first applies `scan` to the (`occlude`, $(0, l)$) monoid and $s_{*,j}$. Because the `occlude` operator is a constant-time function and the size of $s_{*,j}$ is n, it follows that the work and span of the `scan` application is:

$$W\big(\texttt{scan (occlude) } (0,l)\ s_{*,j}\big) = O(n)$$
$$S\big(\texttt{scan (occlude) } (0,l)\ s_{*,j}\big) = O(\log(n))$$

The occluded column returned by this `scan`-application is then unzipped into an n-sized array containing color values and an n-sized array containing height values. The work and span of this operation is:

$$W(\texttt{unzip col\_occluded}) = O(n)$$

[6] `color_height_pairs` here is shorthand for **`let`** `color_height_pairs = ...`

$$S(\texttt{unzip col\_occluded}) = O(1)$$

The color and height arrays are used to render the column corresponding to $g_{*,j}$ on screen. This is done via a combination of `replicate` and `scatter` followed by a `scan`-application filling in missing colors. The `replicate` application takes l as it first argument, so its work and span is:

$$W\big(\texttt{replicate}\ l\ c_{\text{bg}}\big) = O(l)$$
$$S\big(\texttt{replicate}\ l\ c_{\text{bg}}\big) = O(1)$$

The `scatter`-application takes the n-sized color and height arrays as second and third argument respectively, so its work and span is:

$$W(\texttt{scatter init\_col hs cs}) = O(n)$$
$$S(\texttt{scatter init\_col hs cs}) = O(1)$$

The l-sized array returned by the `scatter` application is fed to the `scan`-application along with the $(\texttt{fill}, c_{bg})$ monoid. Because `fill` is a constant-time function, the work and span of the `scan` application is

$$W\big(\texttt{scan (fill)}\ c_{\text{bg}}\ \texttt{screen\_col}\big) = O(l)$$
$$S\big(\texttt{scan (fill)}\ c_{\text{bg}}\ \texttt{screen\_col}\big) = O(\log(l))$$

f_2 is applied to each of the m columns in G so the total work and span must be

$$\begin{aligned}
W(\texttt{rendered_frame}) &= O(nm) + O(m)(O(n) + O(n) + O(l) + O(n) + O(l)) \\
&= O(nm) + O(ml) \\
&= O(m(n+l)) \\
S(\texttt{rendered_frame}) &= O(1) + O(1)(O(\log(n)) + O(1) + O(1) + O(1) + O(\log(l))) \\
&= O(\log(n)) + O(\log(l)) \\
&= O(\log(nl))
\end{aligned}$$

The final step is transposing the rendered frame from $m \times l$-shape to $l \times m$-shape. The work and span of this operation is:

$$W(\texttt{transpose rendered\_frame}) = O(ml)$$
$$S(\texttt{transpose rendered\_frame}) = O(1)$$

Putting it all together, we end up with the following work and span for our implementation:

$$\begin{aligned}
W(\texttt{render cam lsc}\ l\ m) &= O(n) + O(nm) + O(m(n+l)) + O(ml) \\
&= O(m(n+l)) \qquad (5.33) \\
S(\texttt{render cam lsc}\ l\ m) &= O(1) + O(1) + O(\log(nl)) + O(1) \\
&= O(\log(nl)) \qquad (5.34)
\end{aligned}$$

Taking **Equation 4.2** into account, we have $n = O\left(\sqrt{d}\right)$. **Equation 5.33** and **Equation 5.34** can therefore be rewritten:

$$W(\texttt{render cam lsc}\ l\ m) = O(m(n+l))$$

$$= O\left(m\left(\sqrt{d}+l\right)\right) \tag{5.35}$$

$$S(\texttt{render cam lsc}\ l\ m) = O(\log(nl))$$

$$= O\left(\log\left(\sqrt{d}l\right)\right) \tag{5.36}$$

We can conclude that the work of our implementation is most dependent on the width m of the screen rendered on, while its span is completely independent of it. In both cases the render distance d is the least significant factor.

5.6 Benchmarking

In **Section 5.5** we derived **Equation 5.35** and **Equation 5.36** which characterize the sequential and parallel performance of the implementation from **Chapter 4** respectively. These equations may not, however, reflect reality. For one, they abstract away constant factors which may have a considerable impact on runtime. Moreover, work and span do not, as discussed in **Section 5.2** take low-level details such as communication costs and initialization overhead into account. To put **Equation 5.35** and **Equation 5.36** into perspective we are going to benchmark the sequential and parallel performance of the implementation. Given that **Equation 5.35** and **Equation 5.36** are functions of d, m and l, it makes sense benchmarking against each of these parameters in turn. What we will do specifically is run three different benchmark tests both in sequential and parallel mode. In each benchmark test we keep two parameters constant at integral value 10 and measure change in execution time as the last parameter is increased.

We use Futharks `dataset` utility with the `-b` option to generate the data needed for benchmarking in an efficient binary format. This is has the unfortunate side-effect of limiting us from supplying meaningful color and height maps as input. This, however, is not a great problem, as the implementation executes mostly the same steps regardless of the values in its height and color-map arguments. To perform the actual benchmarking we use Futharks `bench` utility with the `backend=c` option for sequential execution and with the `backend=opencl` option for parallel execution. We refer the reader to Listing A.4 for the full benchmarking code. The system we use for benchmarking is a desktop PC with an AMD Ryzen 5 3600 CPU and an NVIDIA GTX 1080 8GB GPU.

To visualize the results of each benchmark test we use separate graphs plotting both sequential and parallel runtimes as functions of input size. To quantify the relative speedup when executing in parallel, we also provide in each graph a third trendline which is simply the OpenCL runtimes divided by the C runtimes. The graphs are show in **Figure 5.6**, **Figure 5.7** and **Figure 5.8**.

These graphs confirm what **Equation 5.35** and **Equation 5.36** imply: that the implementation can be executed efficiently in parallel, but to varying degree depending on the value of parameters d, m, and l. This is especially evident when comparing **Figure 5.6**, where the implementation is benchmarked as a function of d, with **Figure 5.7**, where it is benchmarked as a function of m. To clearly show the difference we provide in **Figure 5.9** and **Figure 5.10** graphs which plot against each other the sequential and parallel runtimes from **Figure 5.6**, **Figure 5.7** and **Figure 5.8** . It is perhaps not surprising that the sequential runtime of the implementation scales best with d and worst with m as they are respectively the least and most significant factor in **Equation 5.35**. The reason why the parallel runtime of the implementation scales similarly is not as obvious, given that

Equation 5.36 is completely independent of m. The most likely explanation is that the GPU of the benchmarking system is not sufficiently hardware parallel to approximate **Equation 5.36**. In that case it is possible that the span and parallel runtime of the implementation may correspond better on a system with a newer GPU. Regardless, setting m (or l) above 5000 does not really make sense, as the dimensions of most screens rarely exceed 3840×2160. Hence we may assume that the implementation scales very well in practice.

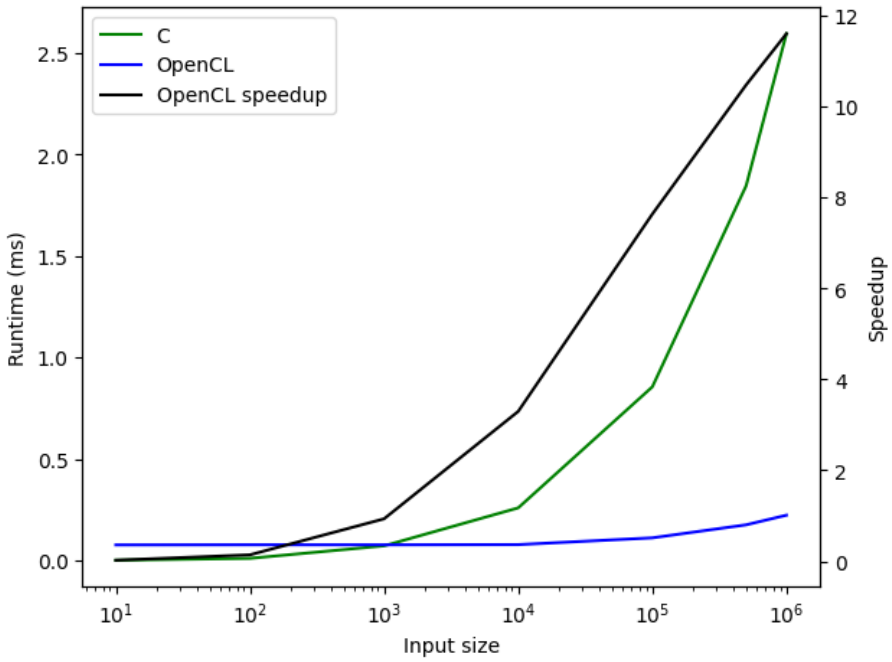


Figure 5.6: Runtime as a function of d when $m = l = 10$.

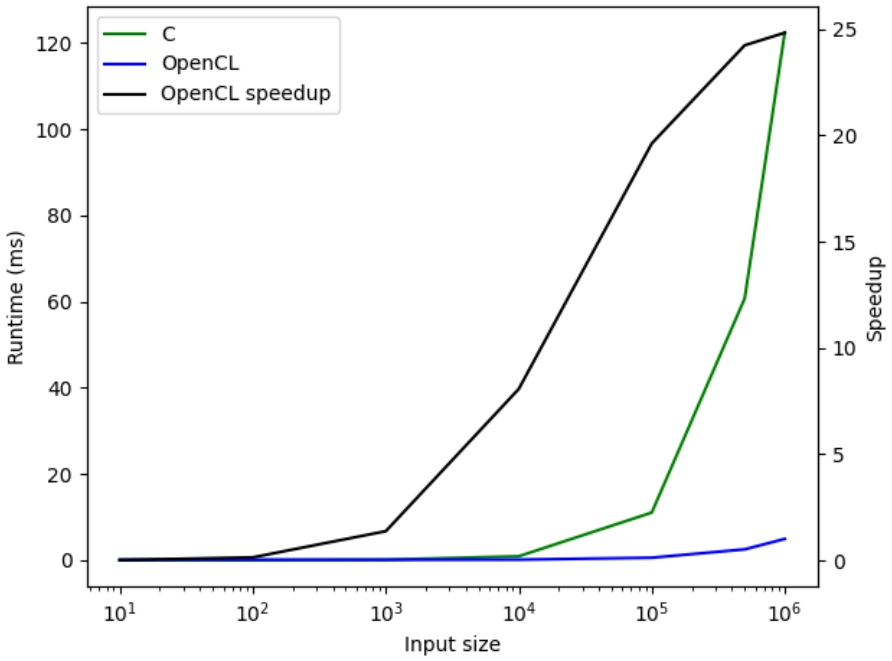


Figure 5.7: Runtime as a function of m when $d = l = 10$.

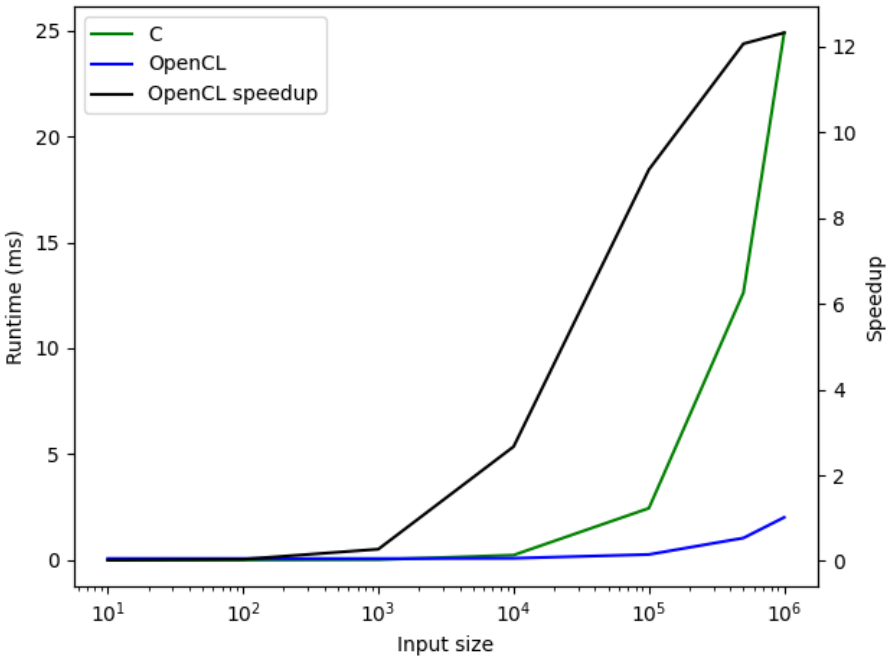


Figure 5.8: Runtime as a function of l when $d = m = 10$.

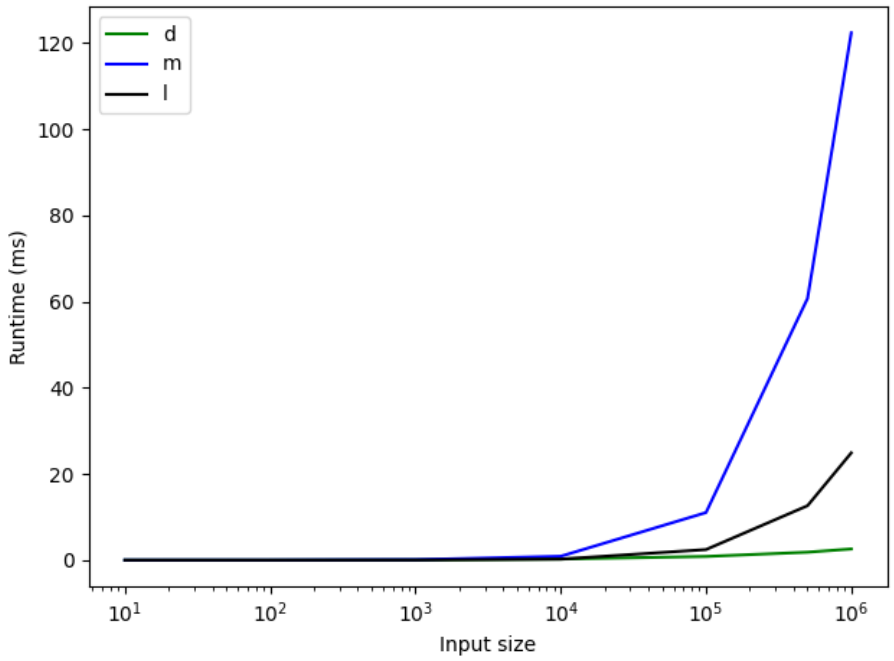


Figure 5.9: Comparing runtime as a function of d, m and l respectively when executing the sequential C binary.

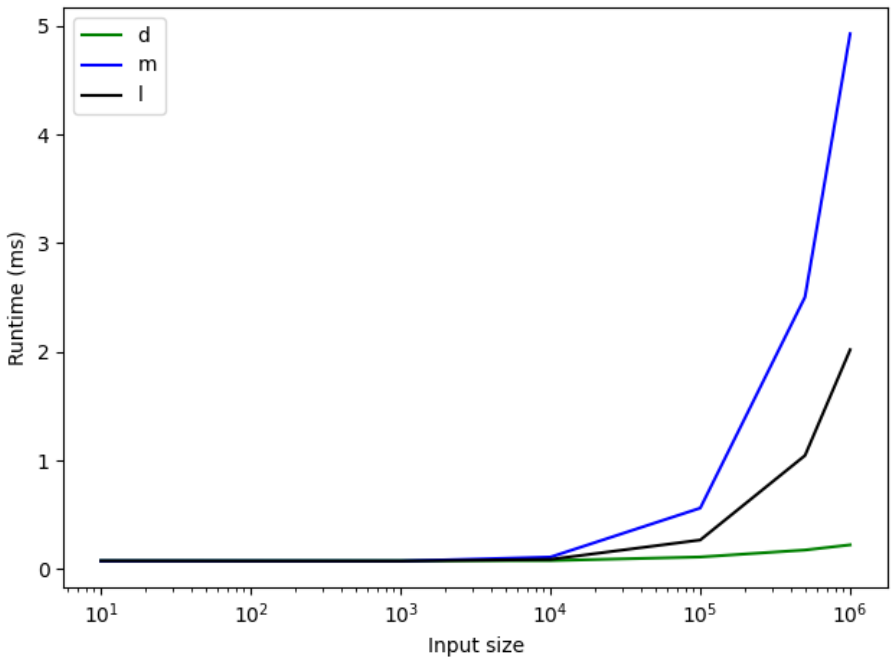


Figure 5.10: Comparing runtime as a function of d, m and l respectively when executing the parallel OpenCL binary.

Chapter 6

Improvements

Having implemented a parallel version of the Voxel Space algorithm and having access to computers that by far, in terms of computational power, exceed those that the original Voxel Space algorithm was designed for, we now look for additional features and rendering techniques to spend our available computational power on. First of all, in addition to the core rendering algorithm, we have replicated the interactive aspect of VoxelSpace. As such, it is possible to move around in the rendered environments in real-time, switch between environments and adjust certain rendering parameters on the fly. The interactive application makes use of *Lys*[Cop20b], an SDL wrapper for Futhark that makes it possible to both handle keyboard and mouse input and blit a framebuffer from inside a Futhark program to an OpenGL window context. The additions we have implemented both apply to the core rendering algorithm and the interactive application.

6.1 Generalising the algorithm

As explained in the previous chapters, (C, H) are mappings $\mathbb{N}^2 \to \mathbb{N}$. However, why limit ourselves to $\mathbb{N}^2$? Conceptually, there is nothing about our rendering approach that keeps us in $\mathbb{N}^2$, as any value in $\mathbb{R}^2$ can be rounded such that we end up back in $\mathbb{N}^2$. In fact, rounding to whole numbers is only necessary when we are dealing with indices into (C, H). Therefore we should be able to present any function $g \in \mathbb{R}^2$ as a valid input to the rendering algorithm, such that our mapping becomes $\mathbb{R}^2 \to \mathbb{N}$ for all inputs. In fact, the only thing stopping us from doing this, is the presumption made in the code about input data being two dimensional integer arrays.

```
1    let seg_point_color = lsc.color[y%q,x%r]
2    let seg_point_height = lsc.altitude[y%q,x%r]
```

Listing 6.1: Retrieving height and color values from arrays

Concretely, if we express the array indexing seen above as a function of (x, y), it naturally becomes possible to render any other function of (x, y) as well. We have implemented this by adding two additional parameters to the rendering algorithm, namely `color_fun:f32->f32->i32` and `height_fun:f32->f32->i32`. Then the rendering of (C, H) suddenly becomes a special application of the rendering algorithm instead of its sole use. The specific handling of (C, H)

has been implemented in two new functions `png_color:[][]i32->f32->f32->i32` and `png_height:[][]i32->f32->f32->f32`, which can then be partially applied with (C, H) in order to get two new functions of respectively type `f32->f32->i32` and `f32->f32->f32`. These functions can then be passed as arguments to the rendering algorithm.

```
let render (c: camera) (color_fun : f32 -> f32 -> i32) (height_fun : f32 ->
    f32 -> f32) (h : i32) (w: i32) (t : smoothing) : [][]i32 =
...
let height_color_map =
    map (\z ->
        ...
        map (\i ->
            ...
            let (interp_color, map_height) = (color_fun x y, height_fun x y)
            ...
            ) (iota w)
) zs
```

Listing 6.2: Core renderer function signature and usage of height and coloring functions

```
let png_height [h][w] (heights : [h][w]i32) (x : f32) (y : f32) : f32 =
    f32.i32 heights[(i32.f32 y)%h, (i32.f32 x)%w]
let png_color [h][w] (colors : [h][w]i32) (x : f32) (y : f32) : f32 =
    colors[(i32.f32 y)%h, (i32.f32 x)%w]
```

Listing 6.3: Handling the special case of image rendering

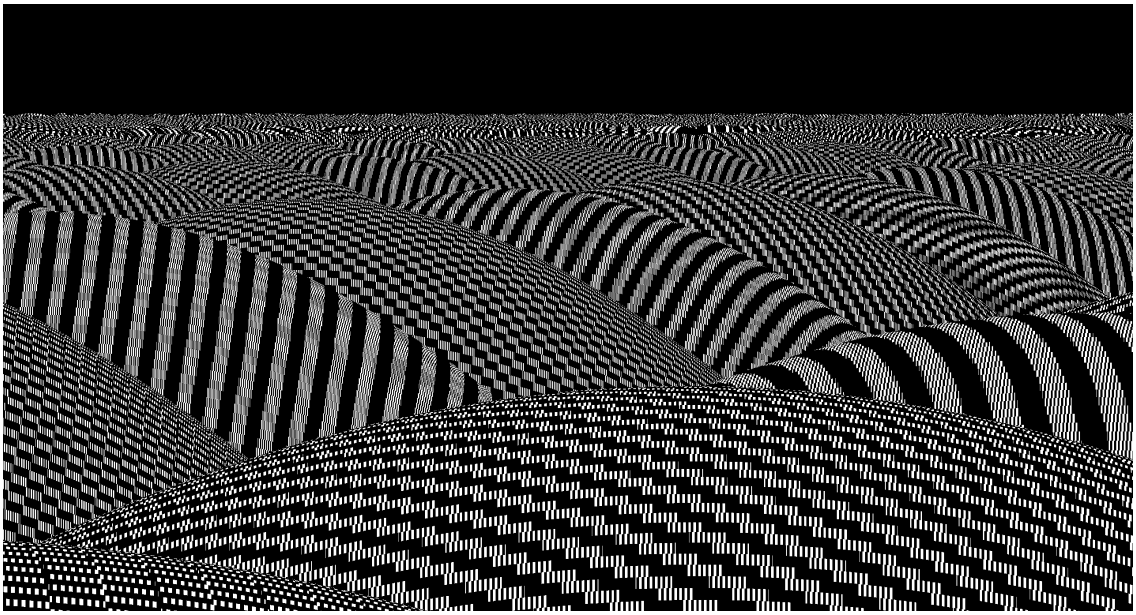


Figure 6.1: A render of $\cos(x) * \sin(y)$ with low field of view

6.2 Shadow rendering

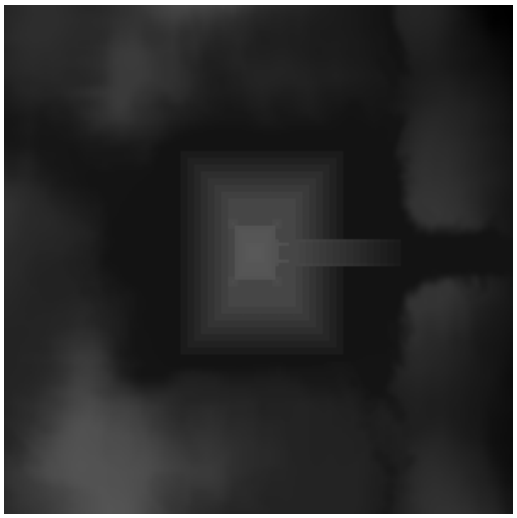


Figure 6.2: Part of height map included with project (C1W.png)

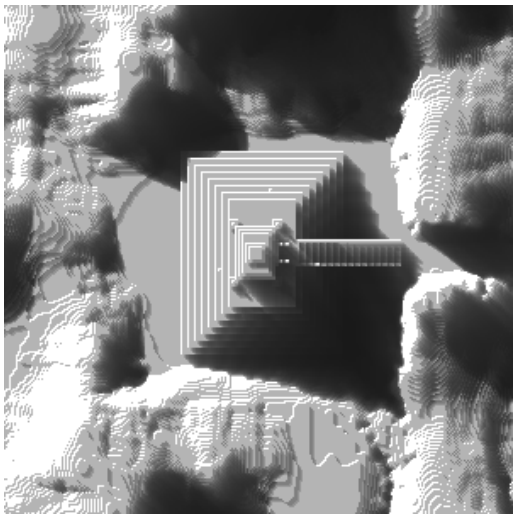


Figure 6.3: Shadowmap generated from heightmap on the left

A major limitation to the Voxel Space algorithm is that shadows are not drawn dynamically. This means that shadows have to be precomputed and blended into the color map, which is the case for the included Comanche maps. This starts becoming a headache when rendering (C, H) pairs that do not contain any prebaked shadows, as the final image lacks any of the depth that the Comanche maps have. However, we wish to stay within the framework of ray casting, and we therefore need a way to generate the shading for each $c_{ij} \in C$, such that the final image for any arbitrary (C, H) appears to contain shadows cast by the sun. First of all, how do we represent light and shadows in terms of (C, H)? Viewing $h_{ij} \in H$ as points, or voxels with the volume $h_{ij} \cdot 1 \cdot 1$, on a plane at coordinate (i, j) is the first step towards building this representation.

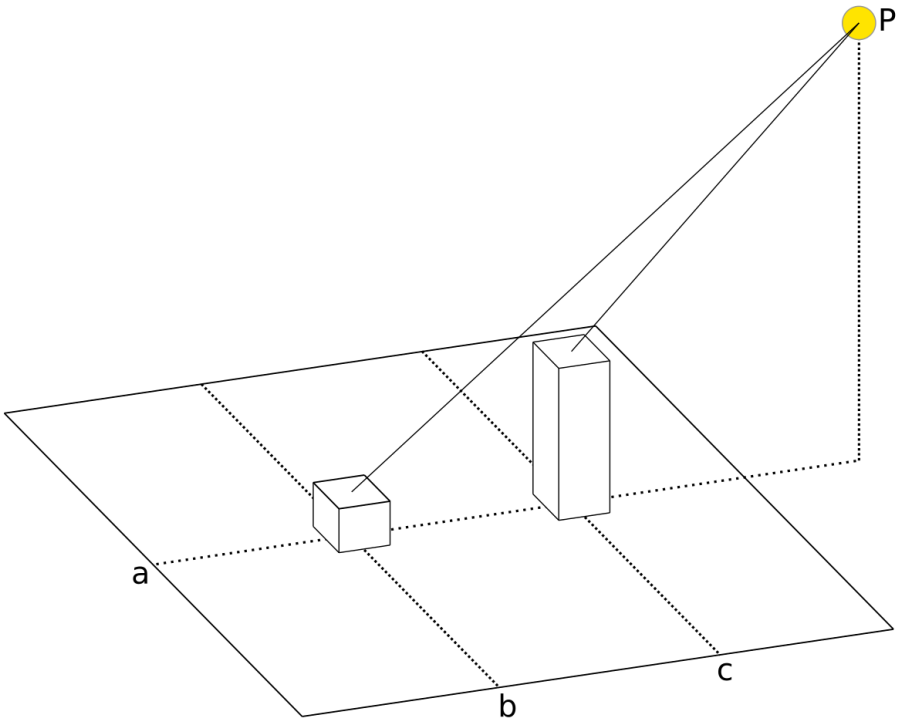


Figure 6.4: Scene with voxels h_{ab}, h_{ac} and light source P.

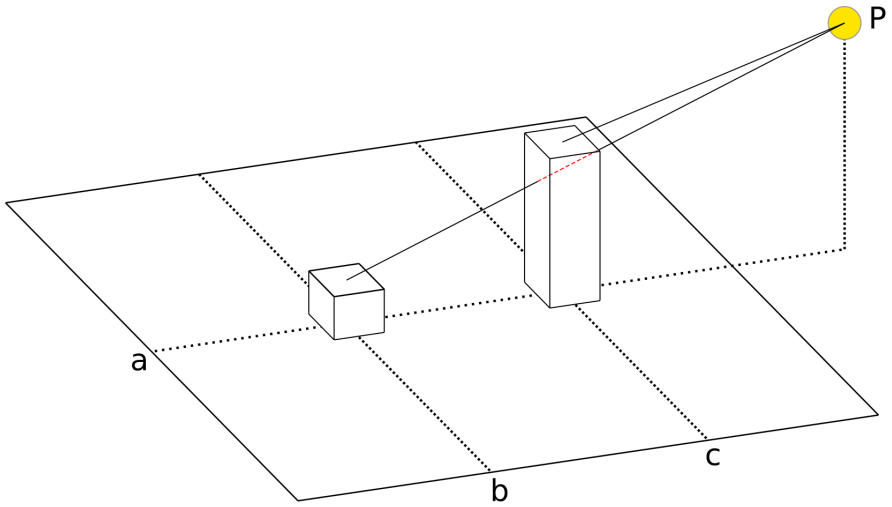


Figure 6.5: P is lowered, such that $\overline{h_{ab}P}$ intersects h_{ac}

We see in **Figure 6.4** that a light source P is shooting rays of light onto the voxels h_{ab} and h_{ac}. P, h_{ab} and h_{ac} are all colinear. If we then lower the height of the P, as in **Figure 6.5**, we see that the light cannot reach h_{ab} anymore, as h_{ac} now obscuring the line $\overline{h_{ab}P}$, the line between h_{ab} and P. If we now imagine that there is an arbitrary amount of voxels in the scene, it must still be the case that for any voxel h_{ij} in the scene there either is or is not some other voxel intersecting $\overline{h_{ij}P}$.

Expressed in terms of **Figure 6.5**, we can formulate this as finding a point on $\overline{h_{ab}P}$ that is inside the volume of h_{ac}. This is the segment of $\overline{h_{ab}P}$ in **Figure 6.5** that is colored red.

How can we find this point? From the perspective of h_{ab}, we have no knowledge of the exact position and volume of h_{ac} until we encounter h_{ac}. Therefore we need to traverse the line from h_{ab} to P until we reach h_{ac}. However, there are two additional details we have to consider. Since we want to model the sun, P should be much further away from the voxels. As the distance between the voxels and P increases, the lines $\overline{h_{ab}P}$, $\overline{h_{ac}P}$ become increasingly parallel to eachother. Secondly, if for some voxel h_{ij} there is nothing obscuring P, it is inconvenient and practically impossible to traverse the entire line $\overline{h_{ij}P}$ if P is basically infinitely distant.

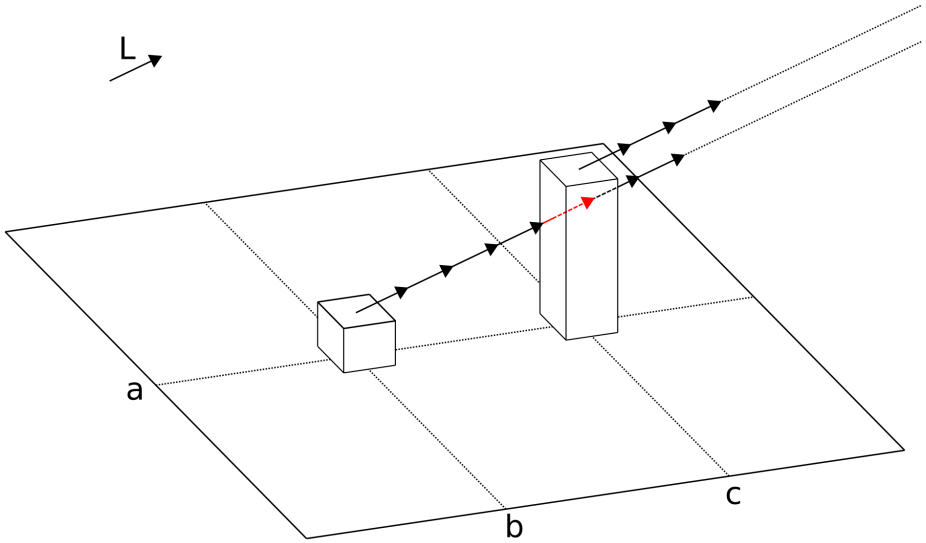


Figure 6.6: Final light representation and intersection model

Therefore, we wish to move away from the idea of lines and P and instead represent the light by a vector $\overrightarrow{L} = (x, y, z)^T$ pointing towards the light. Then, in order to discover a potential intersection at some distance d from h_{ij}, we can simply calculate $\overrightarrow{L'} = h_{ij} + \overrightarrow{L} \cdot d$ and test if $\overrightarrow{L'}$ is inside another voxel. Mathematically we define this intersection test as

$$\text{Intersection}(h_{i,j}, d) = \begin{cases} 1 & h_{i,j} + d \cdot \overrightarrow{L}.y - h_{i+d\cdot\overrightarrow{L}.x, j+d\cdot\overrightarrow{L}.z} < 0 \\ 0 & h_{i,j} + d \cdot \overrightarrow{L}.y - h_{i+d\cdot\overrightarrow{L}.x, j+d\cdot\overrightarrow{L}.z} \geq 0 \end{cases}. \tag{6.1}$$

As we potentially still have to test many values of d for any voxel, we introduce a maximum value d_m for $\overrightarrow{L} \cdot d$. We also introduce a step size d_s and step count d_c such that $d_s = \frac{d_m}{d_c}$ and $d = k \cdot d_s, k \in \{0, 1, 2, ..., d_c\}$.

Having explained the shadow mapping on a conceptual level and defined the necessary intersection test in **Equation 6.1**, we can now move on to the actual implementation. Since **Equation 6.1** for a

voxel h_{ij} does not depend on the solution of **Equation 6.1** for any other voxel, we can do the intersection testing in parallel for all voxels. Furthermore, since we have defined Intersection($h_{i,j}, d$) along with d_m and d_s, we can parallelize the ray traversal as well. We have implemented this as a triple nested `map` SOAC, where the outer two `map` SOACs work on all $h_{i,j}$ and the innermost `map` SOAC is used to perform the ray traversal in parallel. The innermost `map` then produces a set of values A of dimension d_c where $a_k \in A$ is a value indicating whether an intersection occured at distance $k \cdot d_s$. These values are then summed to a single real value $s_{i,j}$ using `reduce (+)`, which is then used as a weight in a color blending with $c_{i,j} \in C$, for some C. The reason why we choose to use the sum of A, instead of simply finding out whether a single intersection has occurred, is that we have found it to be more aesthetically pleasing when doing the color blending. It gives the shadow maps more depth, as crevaces and low points in the terrain end up becoming increasingly shaded as $\overrightarrow{L}$ becomes increasingly parallel to the plane.

```
let generate_shadowmap_accumulated (q : i32) (r : i32) (color_fun : f32 ->
    f32 -> i32) (height_fun : f32 -> f32 -> f32) (sun : [3]f32) : [][]i32 =
let max_dist = 1024
let steps_per_ray = 256
let step_size = f32.i32 (max_dist / steps_per_ray)
in
map (\y ->
    map (\x ->
        let x = f32.i32 x
        let y = f32.i32 y
        let height = height_fun x y
        let intersections =
            map (\dist ->
                let dist = f32.i32 dist
                let height = height + dist * step_size * sun[1]
                let test_height = height_fun
                                  (x + dist * step_size * sun[0])
                                  (y + dist * step_size * sun[2])
                in
                if height - test_height < 0 then 1 else 0
                ) (1..<steps_per_ray)
        let amount = f32.i32 (reduce (+) 0 intersections)
        in (argb.mix (amount) argb.black 1.0 (color_fun x y))
    )(0..<r)
) (0..<q)
```

Listing 6.4: The shadowmap generation implementation

As seen in **Listing 6.4**, the implementation follows the explanation to the letter. The outer two map SOACs iterate over both dimensions of some H, and the inner map SOAC does the intersection testing. As we want black shadows, we use s_{ij} as a weight against c_{ij} in `argb.mix`, a color mixing procedure imported from the futhark library *Matte*[Hen20]. Furthermore, while not shown inside the shadow mapping function, the direction of `sun` is changed through the use of a matrix multiplication library *Linalg*[Cop20a].

Finally, we need to look at the asymptotic behavior of the algorithm along with some benchmarks to see how the algorithm actually performs. Based on the work and span definitions in **Section**

5.2, our triple nested `map`, along with `reduce (+)` leads to a work of $O(q \cdot r \cdot (d_c + d_c)) = O(q \cdot r \cdot d_c)$. Our span, on the other hand, becomes $O(\log(d_c))$. As seen in Figure 6.9, the runtime of the algorithm scales with d_c, which fits with the asymptotic analysis. Casting shadows therefore becomes prohibitively expensive to perform in real-time when d_c is too big.

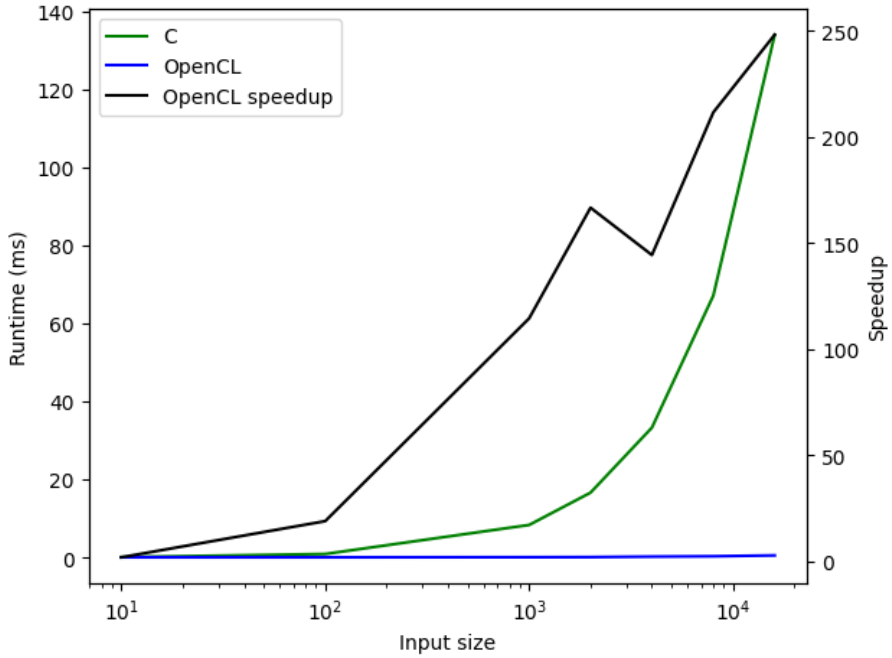


Figure 6.7: Runtime of shadow mapping as a function of q when $d_c = 32$ and $r = 128$

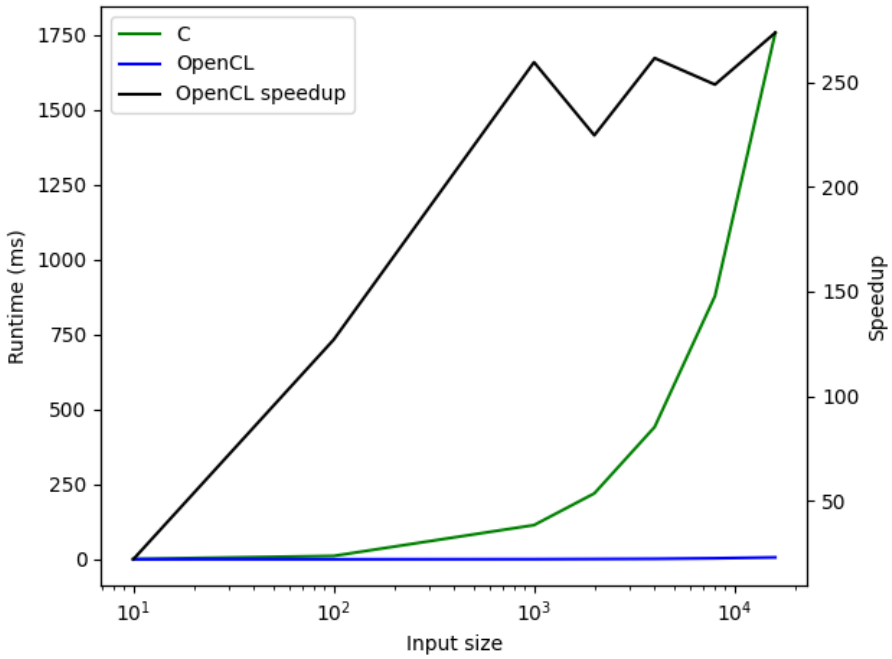


Figure 6.8: Runtime of shadow mapping as a function of d_c when $q = r = 128$

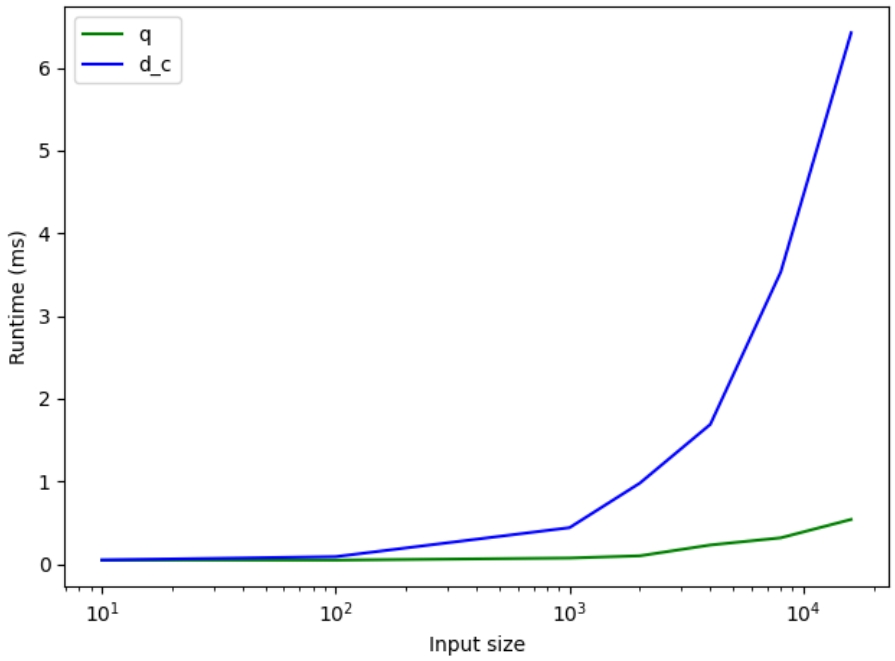


Figure 6.9: Comparing runtime of q and d_c respectively when executing the parallel OpenCL binary

6.3 Bilinear Filtering

A defining visual feature of the height/color map rendering is the sharp, well-defined look of the rendered voxel columns, however we want to increase the overall smoothness of the final image. The sharp, blocky look is especially apparent in the foreground of the image, which is due to the fact that the foreground is sampled more times per $h_{i,j} \in H$ compared to the background.

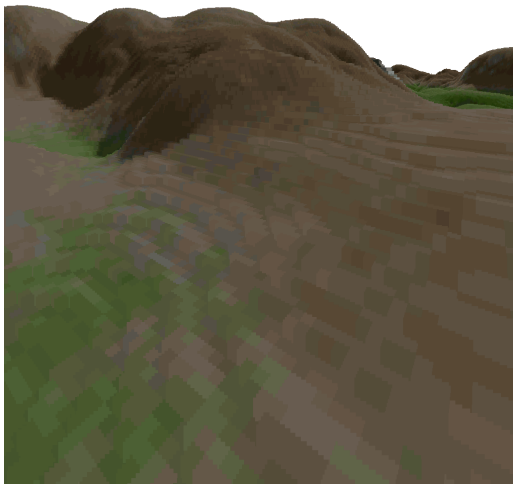


Figure 6.10: Notice how the hill in the foreground is perceptually lower resolution than the hills in the background

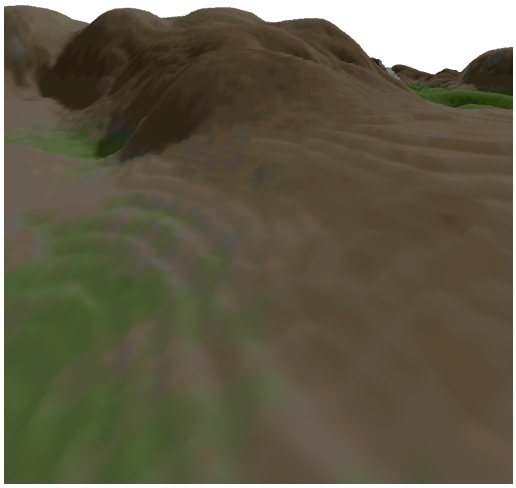


Figure 6.11: Same image and rendering settings, but with height and color map interpolation in use

As mentioned in **Section 6.1**, we do a mapping $\mathbb{R}^2 \to \mathbb{N}^2$ when dealing with (C, H) pairs. We do this using a function $p(x, y)$ defined in **Equation 2.9**, where $(x, y) \in (\mathbb{R}, \mathbb{R})$ become indices to some (C, H) pair. However, we can introduce a new function $p'(x, y)$ that uses the respective fractional parts of (x, y) to perform interpolation between the nearest indices around (x, y). A suitable candidate mathematical method for interpolation in $\mathbb{R}^2$ is bilinear interpolation. Bilinear interpolation is the $\mathbb{R}^2$ equivalent of linear interpolation, which is defined as

$$f(x) \approx (x_2 - x_1) f(x_1) + (x_2 - x_1) f(x_2), \quad x_1 \leq x \leq x_2 \tag{6.2}$$

Therefore if we have a function f, the function values of f at (x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2), and we want to find $f(x, y)$ where $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$, we can approximate $f(x, y)$ by linearly interpolating first in the x-direction and then in the y-direction:

$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(x_1, y_1) + \frac{x - x_1}{x_2 - x_1} f(x_2, y_1) \tag{6.3}$$

$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(x_1, y_2) + \frac{x - x_1}{x_2 - x_1} f(x_2, y_2) \tag{6.4}$$

$$f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \tag{6.5}$$

In our case we apply this to neighboring points in (C, H), such that the distance $x_2 - x_1 = 1$ and $y_2 - y_1 = 1$. Therefore we can simplify the previous equations:

$$f(x, y_1) \approx (x_2 - x) f(x_1, y_1) + (x - x_1) f(x_2, y_1) \tag{6.6}$$

$$f(x, y_2) \approx (x_2 - x) f(x_1, y_2) + (x - x_1) f(x_2, y_2) \tag{6.7}$$

$$f(x, y) \approx (y_2 - y) f(x, y_1) + (y - y_1) f(x, y_2) \tag{6.8}$$

And **Equation 6.6**, **Equation 6.7**, **Equation 6.8** in terms of $p'(x, y)$ become

$$p'(x, \lfloor y \rfloor) = (\lceil x \rceil - x)p(\lfloor x \rfloor, \lfloor y \rfloor) + (x - \lfloor x \rfloor)p(\lceil x \rceil, \lfloor y \rfloor) \tag{6.9}$$
$$p'(x, \lceil y \rceil) = (\lceil x \rceil - x)p(\lfloor x \rfloor, \lceil y \rceil) + (x - \lfloor x \rfloor)p(\lceil x \rceil, \lceil y \rceil) \tag{6.10}$$

$$p'(x, y) = (\lceil y \rceil - y)p'(x, \lfloor y \rfloor) + (y - y_1)p'(x, \lceil y \rceil) \tag{6.11}$$

The actual implementation of the filtering is spread across two functions, as the filtering needs to happen per color channel for $c \in C$.

Chapter 7

Conclusion

By breaking down and analyzing VoxelSpace, and the underlying Voxel Space algorithm, we have been succesful in translating the rendering stages of the algorithm such that they become highly parallel. The translation itself makes use of second order array combinators (SOACs), as they are proven to be easily parallelized. The analysis and translation has resulted in a working implementation of the VoxelSpace algorithm in Futhark, a pure functional programming language that makes use of the parallelizable nature of SOACs to generate CUDA or OpenCL programs, such that our implementation makes use of parallelism on the GPU. Besides the translation of the core rendering algorithm, we have also implemented an interactive application that makes it possible to move around a rendered terrain in real time with keyboard inputs. The interactive application makes use of a Futhark SDL binding, lys, which makes it possible to easily create an OpenGL window context and display a rendered frame from the core renderer on screen.

Secondly, we have looked into ways in which the compute power of modern GPUs can be used to improve the rendering capabilities of the Voxel Space algorithm. This resulted in a parallelized ray tracing-type shadow mapping algorithm, the introduction of bilinear filtering in the ray casting, along with the ability to dynamically render terrain defined by mathematical functions instead of images.

In terms of ideas for future improvements to our translation of the core renderer, we find it most relevant to consider finding a way to avoid the use of `scan` in the height map occlusion step, as `scan` has a span of $S(\texttt{scan}) = \log(n)$, which is asymptotically the most taxing step of the rendering process. In a similar vein, the shadow mapping implementation also makes use of `scan`, which leads to poor performance when both casting shadows on large terrains or with long rays. Perhaps some acceleration structure can be applied, so that the ray traversal step becomes less gruesome. The translation also assumes that only a terrain should be rendered, so another thing that would be interesting to research is a way to represent other types of objects on the terrain, such as sprites, so that they can be rendered as a part of the core renderer as well.

Bibliography

[20a] *Futhark Documentation*. Release 0.80. Feb. 2020. URL: https://futhark.readthedocs.io.

[20b] *Futhark Library Documentation – SOACs*. Apr. 2020. URL: https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html (visited on 05/18/2020).

[20c] *NumPy User Guide*. Release 1.18.1. NumPy Community. Feb. 2020. URL: https://numpy.org/doc/1.18/numpy-user.pdf.

[96] "NovaLogic". In: *Next Generation* 20 (1996), pp. 62–63. URL: https://archive.org/details/nextgen-issue-020/page/n63/mode/2up.

[Ble90] Guy E. Blelloch. *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990. DOI: 10.1184/R1/6608579.v1.

[Ble96] Guy E. Blelloch. "Programming Parallel Algorithms". In: *Communications of the ACM* 39.3 (1996), pp. 85–97. DOI: 10.1145/227234.227246.

[Cop20a] DIKU – University of Copenhagen. *Linalg*. June 2020. URL: https://github.com/diku-dk/linalg (visited on 06/07/2020).

[Cop20b] DIKU – University of Copenhagen. *Lys*. June 2020. URL: https://github.com/diku-dk/lys (visited on 06/07/2020).

[EG88] David Eppstein and Zvi Galil. "Parallel Algorithmic Techniques For Combinational Computation". In: *Annual Review of Computer Science* 3.1 (1988), pp. 233–283. DOI: 10.1146/annurev.cs.03.060188.001313.

[EHO20] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. 2020. URL: https://futhark-book.readthedocs.io.

[Fly72] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[Fre00] Kyle G. Freeman. "System and method for realistic terrain simulation". Patent US6020893A (US). Feb. 2000. URL: https://patents.google.com/patent/US6020893.

[Fre96] Kyle G. Freeman. "Technique and system for the real-time generation of perspective images". Patent US5550959A (US). Aug. 1996. URL: https://patents.google.com/patent/US5550959A.

[Hen19a] Troels Henriksen. “Cost models and advanced Futhark programming”. University Lecture. Nov. 2019. URL: https://github.com/diku-dk/pfp-e2019-pub/blob/master/slides/L2-advanced-futhark-cost-models.pdf.

[Hen19b] Troels Henriksen. “Deterministic parallel programming and data parallelism”. University Lecture. Nov. 2019. URL: https://github.com/diku-dk/pfp-e2019-pub/blob/master/slides/L1-determ-prog.pdf.

[Hen19c] Troels Henriksen. *What is the minimal basis for Futhark?* Apr. 2019. URL: https://futhark-lang.org/blog/2019-04-10-what-is-the-minimal-basis-for-futhark.html (visited on 05/20/2020).

[Hen20] Troels Henriksen. *Matte*. June 2020. URL: https://github.com/athas/matte (visited on 06/07/2020).

[Kae+15] David R. Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous Computing with OpenCL 2.0*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015. ISBN: 0128014148.

[Kri+20] Jonas Kristensen, Mathias Rasmussen, Jens Sørensen, and Christian Troelsen. *FutSpace*. June 2020. URL: https://github.com/MrHootJones/futspace (visited on 06/07/2020).

[Mac20] Sebastian Macke. *VoxelSpace*. Jan. 2020. URL: https://github.com/s-macke/VoxelSpace (visited on 05/18/2020).

[Nvi] Nvidia. *Nvidia History - A Timeline of Innovation*. URL: https://www.nvidia.com/en-us/about-nvidia/corporate-timeline/ (visited on 06/05/2020).

[Sta00] FlightSim.Com Staff. *NovaLogic Awarded Patent For Voxel Space Graphics Engine*. Feb. 2000. URL: https://www.flightsim.com/vbfs/content.php?2994-NovaLogic-Awarded-Patent-For-Voxel-Space-Graphics-Engine (visited on 05/29/2020).

Appendices

Appendix A

Code

Listing A.1: JavaScript implementation of the Voxel Space rendering algorithm (and associated interactive application.

```
"use strict";

// ---------------------------------------------
// Viewer information

var camera =
{
    x: 512., // x position on the map
    y: 800., // y position on the map
    height: 78., // height of the camera
    angle: 0., // direction of the camera
    horizon: 100., // horizon position (look up and down)
    distance: 800   // distance of map
};

// ---------------------------------------------
// Landscape data

var map =
{
    width: 1024,
    height: 1024,
    shift: 10,  // power of two: 2^10 = 1024
    // 1024 * 1024 byte array with height information
    altitude: new Uint8Array(1024 * 1024),
    // 1024 * 1024 int array with RGB colors
    color: new Uint32Array(1024 * 1024)
};

// ---------------------------------------------
// Screen data

var screendata =
{
```

```
35      canvas: null,
36      context: null,
37      imagedata: null,
38
39      bufarray: null, // color data
40      buf8: null, // the same array but with bytes
41      buf32: null, // the same array but with 32-Bit words
42
43      backgroundcolor: 0xFFE09090
44  };
45
46  var updaterunning = false;
47
48  var time = new Date().getTime();
49
50
51  // for fps display
52  var timelastframe = new Date().getTime();
53  var frames = 0;
54
55  // Update the camera for next frame. Dependent on keypresses
56  function UpdateCamera() {
57      var current = new Date().getTime();
58
59      input.keypressed = false;
60      if (input.leftright != 0) {
61          camera.angle +=
62              input.leftright * 0.1
63              * (current - time) * 0.03;
64          input.keypressed = true;
65      }
66      if (input.forwardbackward != 0) {
67          camera.x -=
68              input.forwardbackward * Math.sin(camera.angle)
69              * (current - time) * 0.03;
70          camera.y -=
71              input.forwardbackward * Math.cos(camera.angle)
72              * (current - time) * 0.03;
73          input.keypressed = true;
74      }
75      if (input.updown != 0) {
76          camera.height +=
77              input.updown * (current - time) * 0.03;
78          input.keypressed = true;
79      }
80      if (input.lookup) {
81          camera.horizon +=
82              2 * (current - time) * 0.03;
83          input.keypressed = true;
84      }
85      if (input.lookdown) {
86          camera.horizon -=
87              2 * (current - time) * 0.03;
88          input.keypressed = true;
```

```
89      }
90
91      // Collision detection. Don't fly below the surface.
92      var mapoffset =
93          ((Math.floor(camera.y) & (map.width - 1)) << map.shift)
94          + (Math.floor(camera.x) & (map.height - 1)) | 0;
95      if ((map.altitude[mapoffset] + 10) > camera.height)
96          camera.height = map.altitude[mapoffset] + 10;
97
98      time = current;
99
100 }
101
102 // --------------------------------------------
103 // Keyboard and mouse event handlers
104 // --------------------------------------------
105 // Keyboard and mouse event handlers
106
107 function GetMousePosition(e) {
108     // fix for Chrome
109     if (e.type.startsWith('touch')) {
110         return [e.targetTouches[0].pageX, e.targetTouches[0].pageY];
111     } else {
112         return [e.pageX, e.pageY];
113     }
114 }
115
116
117 function DetectMouseDown(e) {
118     input.forwardbackward = 3.;
119     input.mouseposition = GetMousePosition(e);
120     time = new Date().getTime();
121
122     if (!updaterunning) Draw();
123     return;
124 }
125
126 function DetectMouseUp() {
127     input.mouseposition = null;
128     input.forwardbackward = 0;
129     input.leftright = 0;
130     input.updown = 0;
131     return;
132 }
133
134 function DetectMouseMove(e) {
135     e.preventDefault();
136     if (input.mouseposition == null) return;
137     if (input.forwardbackward == 0) return;
138
139     var currentMousePosition = GetMousePosition(e);
140
141     input.leftright =
142         (input.mouseposition[0] - currentMousePosition[0])
```

```
        / window.innerWidth * 2;
    camera.horizon =
        100 + (input.mouseposition[1] - currentMousePosition[1])
        / window.innerHeight * 500;
    input.updown =
        (input.mouseposition[1] - currentMousePosition[1])
        / window.innerHeight * 10;
}


function DetectKeysDown(e) {
    switch (e.keyCode) {
        case 37:    // left cursor
        case 65:    // a
            input.leftright = +1.;
            break;
        case 39:    // right cursor
        case 68:    // d
            input.leftright = -1.;
            break;
        case 38:    // cursor up
        case 87:    // w
            input.forwardbackward = 3.;
            break;
        case 40:    // cursor down
        case 83:    // s
            input.forwardbackward = -3.;
            break;
        case 82:    // r
            input.updown = +2.;
            break;
        case 70:    // f
            input.updown = -2.;
            break;
        case 69:    // e
            input.lookup = true;
            break;
        case 81:    //q
            input.lookdown = true;
            break;
        default:
            return;
            break;
    }

    if (!updaterunning) {
        time = new Date().getTime();
        Draw();
    }
    return false;
}

function DetectKeysUp(e) {
    switch (e.keyCode) {
```

```
197         case 37:    // left cursor
198         case 65:    // a
199             input.leftright = 0;
200             break;
201         case 39:    // right cursor
202         case 68:    // d
203             input.leftright = 0;
204             break;
205         case 38:    // cursor up
206         case 87:    // w
207             input.forwardbackward = 0;
208             break;
209         case 40:    // cursor down
210         case 83:    // s
211             input.forwardbackward = 0;
212             break;
213         case 82:    // r
214             input.updown = 0;
215             break;
216         case 70:    // f
217             input.updown = 0;
218             break;
219         case 69:    // e
220             input.lookup = false;
221             break;
222         case 81:    //q
223             input.lookdown = false;
224             break;
225         default:
226             return;
227             break;
228     }
229     return false;
230 }
231
232 // ------------------------------------------
233 // Fast way to draw vertical lines
234
235 function DrawVerticalLine(x, ytop, ybottom, col) {
236     x = x | 0;
237     ytop = ytop | 0;
238     ybottom = ybottom | 0;
239     col = col | 0;
240     var buf32 = screendata.buf32;
241     var screenwidth = screendata.canvas.width | 0;
242     if (ytop < 0) ytop = 0;
243     if (ytop > ybottom) return;
244
245     // get offset on screen for the vertical line
246     var offset = ((ytop * screenwidth) + x) | 0;
247     for (var k = ytop | 0; k < ybottom | 0; k = k + 1 | 0) {
248         buf32[offset | 0] = col | 0;
249         offset = offset + screenwidth | 0;
250     }
```

```
}

// ---------------------------------------------
// Basic screen handling

function DrawBackground() {
    var buf32 = screendata.buf32;
    var color = screendata.backgroundcolor | 0;
    for (var i = 0; i < buf32.length; i++)
        buf32[i] = color | 0;
}

// Show the back buffer on screen
function Flip() {
    screendata.imagedata.data.set(screendata.buf8);
    screendata.context.putImageData(screendata.imagedata, 0, 0);
}

// ---------------------------------------------
// The main render routine

function Render() {
    var mapwidthperiod = map.width - 1;
    var mapheightperiod = map.height - 1;

    var screenwidth = screendata.canvas.width | 0;
    var sinang = Math.sin(camera.angle);
    var cosang = Math.cos(camera.angle);

    var hiddeny = new Int32Array(screenwidth);
    for (var i = 0; i < screendata.canvas.width | 0; i = i + 1 | 0)
        hiddeny[i] = screendata.canvas.height;

    var deltaz = 1.;

    // Draw from front to back
    for (var z = 1; z < camera.distance; z += deltaz) {
        // 90 degree field of view
        var plx = -cosang * z - sinang * z;
        var ply = sinang * z - cosang * z;
        var prx = cosang * z - sinang * z;
        var pry = -sinang * z - cosang * z;

        var dx = (prx - plx) / screenwidth;
        var dy = (pry - ply) / screenwidth;
        plx += camera.x;
        ply += camera.y;
        var invz = 1. / z * 240.;

        for (var i = 0; i < screenwidth | 0; i = i + 1 | 0) {
            var mapoffset =
                ((Math.floor(ply) & mapwidthperiod) << map.shift)
                + (Math.floor(plx) & mapheightperiod) | 0;
            var heightonscreen =
```

```
                (camera.height - map.altitude[mapoffset])
                * invz + camera.horizon | 0;
            DrawVerticalLine(
                i, heightonscreen | 0,
                hiddeny[i],
                map.color[mapoffset]
            );
            if (heightonscreen < hiddeny[i])
                hiddeny[i] = heightonscreen;
            plx += dx;
            ply += dy;
        }

        deltaz += 0.005;
    }
}



// --------------------------------------------
// Draw the next frame

function Draw() {
    updaterunning = true;
    UpdateCamera();
    DrawBackground();
    Render();
    Flip();
    frames++;

    if (!input.keypressed) {
        updaterunning = false;
    } else {
        window.requestAnimationFrame(Draw, 0);
    }
}

// --------------------------------------------
// Init routines

// Util class for downloading the png
function DownloadImagesAsync(urls) {
    return new Promise(function (resolve, reject) {

        var pending = urls.length;
        var result = [];
        if (pending === 0) {
            resolve([]);
            return;
        }
        urls.forEach(function (url, i) {
            var image = new Image();
            //image.addEventListener("load", function() {
            image.onload = function () {
```

```
                var tempcanvas =
                    document.createElement("canvas");
                var tempcontext =
                    tempcanvas.getContext("2d");
                tempcanvas.width = map.width;
                tempcanvas.height = map.height;
                tempcontext.drawImage(
                    image, 0, 0,
                    map.width, map.height
                );
                result[i] =
                    tempcontext.getImageData(
                        0, 0, map.width,
                        map.height
                    ).data;
                pending--;
                if (pending === 0) {
                    resolve(result);
                }
            };
            image.src = url;
        });
    });
}

function LoadMap(filenames) {
    var files = filenames.split(";");
    DownloadImagesAsync(
        [
            "maps/" + files[0] + ".png",
            "maps/" + files[1] + ".png"
        ]
    ).then(OnLoadedImages);
}

function OnLoadedImages(result) {
    var datac = result[0];
    var datah = result[1];
    for (var i = 0; i < map.width * map.height; i++) {
        map.color[i] =
            0xFF000000
            | (datac[(i << 2) + 2] << 16)
            | (datac[(i << 2) + 1] << 8)
            | datac[(i << 2) + 0];
        map.altitude[i] = datah[i << 2];
    }
    Draw();
}

function OnResizeWindow() {
    screendata.canvas =
        document.getElementById('fullscreenCanvas');

    var aspect =
```

```
413         window.innerWidth / window.innerHeight;
414
415     screendata.canvas.width =
416         window.innerWidth < 800 ? window.innerWidth :
417             800;
418     screendata.canvas.height =
419         screendata.canvas.width / aspect;
420
421     if (screendata.canvas.getContext) {
422         screendata.context =
423             screendata.canvas.getContext('2d');
424         screendata.imagedata =
425             screendata.context.createImageData(
426                 screendata.canvas.width,
427                 screendata.canvas.height
428             );
429     }
430
431     screendata.bufarray =
432         new ArrayBuffer(
433             screendata.imagedata.width
434             * screendata.imagedata.height * 4
435         );
436     screendata.buf8 =
437         new Uint8Array(screendata.bufarray);
438     screendata.buf32 =
439         new Uint32Array(screendata.bufarray);
440     Draw();
441 }
442
443 function Init() {
444     for (var i = 0; i < map.width * map.height; i++) {
445         map.color[i] = 0xFF007050;
446         map.altitude[i] = 0;
447     }
448
449     LoadMap("C1W;D1");
450     OnResizeWindow();
451
452     // set event handlers for keyboard, mouse, touchscreen and window resize
453     var canvas =
454         document.getElementById("fullscreenCanvas");
455     window.onkeydown = DetectKeysDown;
456     window.onkeyup = DetectKeysUp;
457     canvas.onmousedown = DetectMouseDown;
458     canvas.onmouseup = DetectMouseUp;
459     canvas.onmousemove = DetectMouseMove;
460     canvas.ontouchstart = DetectMouseDown;
461     canvas.ontouchend = DetectMouseUp;
462     canvas.ontouchmove = DetectMouseMove;
463
464     window.onresize = OnResizeWindow;
465
466     window.setInterval(function () {
```

```
        var current = new Date().getTime();
        document.getElementById('fps').innerText =
            (frames / (current - timelastframe) * 1000).toFixed(1)
            + " fps";
        frames = 0;
        timelastframe = current;
    }, 2000);

}

Init();
```

Listing A.2: Futhark implementation of the Voxel Space rendering algorithm

```
1
2  -- benchmark program as a function of d, m and l respectively
3  -- ==
4  -- input @ d_10
5  -- input @ d_100
6  -- input @ d_1000
7  -- input @ d_10000
8  -- input @ d_100000
9  -- input @ d_500000
10 -- input @ d_1000000
11 -- input @ m_10
12 -- input @ m_100
13 -- input @ m_1000
14 -- input @ m_10000
15 -- input @ m_100000
16 -- input @ m_500000
17 -- input @ m_1000000
18 -- input @ l_10
19 -- input @ l_100
20 -- input @ l_1000
21 -- input @ l_10000
22 -- input @ l_100000
23 -- input @ l_500000
24 -- input @ l_1000000
25 type camera =
26   { x : f32,
27     y : f32,
28     height : f32,
29     angle : f32,
30     horizon : f32,
31     distance : f32,
32     fov : f32 }
33
34 type landscape [q][r] =
35   { width : i32,
36     height : i32,
37     color : [q][r]i32,
38     altitude : [q][r]i32,
39     sky_color : i32 }
40
```

```
type line =
  { x_0 : f32,
    y_0 : f32,
    dx : f32,
    dy : f32 }

let get_zs (delta: f32)(d : f32) (z_0 : f32) : []f32 =
  let sqrt =  f32.sqrt ((delta-2*z_0)**2 + 8*delta*d)
  let num = sqrt - 2*z_0 + delta
  let div = 2*delta
  let n = i32.f32 (num / div)
  let is = map (\i -> f32.i32 i) (1...n)
  in map (\i -> (i/2) * (2*z_0 + (i-1) * delta)) is

let get_h_line (z : f32) (cam : camera) (m : i32) : line =
  let sin_ang = f32.sin cam.angle
  let cos_ang = f32.cos cam.angle
  let view = cam.fov
  let left_x = (- cos_ang  - sin_ang * view )*z
  let left_y = (sin_ang  - cos_ang * view)*z
  let right_x = (cos_ang  - sin_ang * view)*z
  let right_y = (- sin_ang  - cos_ang * view)*z

  let seg_dim_x = (right_x - left_x) / (f32.i32 m)
  let seg_dim_y = (right_y - left_y) / (f32.i32 m)

  let left_x = left_x + cam.x
  let left_y = left_y + cam.y

  in
  { x_0 = left_x,
    y_0 = left_y,
    dx = seg_dim_x,
    dy = seg_dim_y }

let get_segment_point (l : line) (j : i32) : (i32, i32) =
  let x = i32.f32 (l.x_0 + (f32.i32 j) * l.dx)
  let y = i32.f32 (l.y_0 + (f32.i32 j) * l.dy)
  in (x, y)

let render [q][r] (cam: camera) (lsc : landscape [q][r])
                  (l : i32) (m: i32) : [l][m]i32 =
  #[unsafe]
  let z_0 = 1.0
  let delta = 0.005
  let zs = get_zs delta cam.distance z_0

  let color_height_pairs =
    map (\z ->
         let h_line = get_h_line z cam m
         let inv_z = (1.0 / z) * 240.0
         in
         map (\j ->
             let (x, y) = get_segment_point h_line j
```

```
95              let seg_point_color = lsc.color[y%q,x%r]
96              let seg_point_height = lsc.altitude[y%q,x%r]
97              let height_diff = cam.height - (f32.i32 seg_point_height)
98              let relative_height = height_diff * inv_z + cam.horizon
99              let bounded_height = i32.min (l-1) (i32.max 0 (i32.f32
    relative_height))
100             in (seg_point_color, nonneg_height)
101             ) (iota m)
102         ) zs
103
104   let occlude (c_1 : i32, h_1 : i32 )
105               (c_2 : i32, h_2 : i32 ) : (i32, i32) =
106     if (h_1 <= h_2)
107     then (c_1, h_1)
108     else (c_2, h_2)
109
110   let fill (c_1 : i32) (c_2 : i32) : i32 =
111     if (c_2 == lsc.sky_color)
112     then c_1
113     else c_2
114
115   let rendered_frame =
116     map (\j ->
117          let col_occluded = scan (occlude) (0, l) j
118          let (cs, hs) = unzip col_occluded
119          let init__col = replicate l lsc.sky_color
120          let screen_col = scatter init__col hs cs
121          in scan (fill) lsc.sky_color screen_col
122         ) (transpose color_height_pairs)
123
124   in transpose rendered_frame
125
126 let main [q][r] (color_map : [q][r]i32)
127                 (height_map: [q][r]i32)
128                 (l:i32) (m: i32) (d: f32) : [][]i32 =
129
130   let init_camera =
131     { x = 512f32,
132       y = 800f32,
133       height = 78f32,
134       angle = 0f32,
135       horizon = 100f32,
136       distance = d,
137       fov = 1f32 }
138
139   let init_landscape =
140     { width = q,
141       height = r,
142       color =  color_map,
143       altitude = height_map,
144       sky_color = 0xFF9090e0 }
145
146   in render init_camera init_landscape l m
```

Listing A.3: Custom implementations of the first-order array combinators used in **Listing A.2**

```
let replicate 't (n: i32) (x: t) : *[n]t =
  map (\i -> x) (iota n)

let zip [n] 'a 'b (as: [n]a) (bs: [n]b) : [n](a,b) =
  map2 (\x1 x2 -> (x1,x2)) as bs

let unzip [n] 'a 'b (xs: [n](a,b)): ([n]a, [n]b) =
  let as = map (\i -> xs[i].0) (iota n)
  let bs = map (\i -> xs[i].1) (iota n)
  in (as, bs)

let transpose [n][m] 't (xss: [n][m]t) : [m][n]t =
  map (\j ->
            map (\i -> xss[i, j]) (iota n)
            ) (iota m)
```

Listing A.4: Auxilliary file used for benchmarking.

```
#core_renderer benchmark

d_%:
	futhark dataset -b --i32-bounds=0:16777215 -g [1024][1024]i32 -g
	↪  [1024][1024]i32 -g 10i32 -g 10i32 -g $*f32 > $@
m_%:
	futhark dataset -b --i32-bounds=0:16777215 -g [1024][1024]i32 -g
	↪  [1024][1024]i32 -g 10i32 -g $*i32 -g 10f32 > $@
l_%:
	futhark dataset -b --i32-bounds=0:16777215 -g [1024][1024]i32 -g
	↪  [1024][1024]i32 -g $*i32 -g 10i32 -g 10f32 > $@

SIZES1 = 10 100 1000 10000 100000 500000 1000000

bench1: $(SIZES1:%=d_%) $(SIZES1:%=m_%) $(SIZES1:%=l_%) renderer_core.fut
	futhark bench --backend=c --json bench_c.json renderer_core.fut
	futhark bench --backend=opencl --json bench_opencl.json
	↪  renderer_core.fut

plot_bench:
	python3 plot_bench.py

clean1:
	rm renderer_core
	rm renderer_core.c
	for i in $(SIZES1); do \
		rm d_$$i; \
		rm m_$$i; \
		rm l_$$i; \
	done

#shadow_mapping benchmark

q_%:
	futhark dataset -b --i32-bounds=0:16777215 -g [$*][128]i32 -g
	↪  [128][128]i32 -g 10i32 > $@
d_%:
	futhark dataset -b --i32-bounds=0:16777215 -g [128][128]i32 -g
	↪  [128][128]i32 -g $*i32 > $@

SIZES2 = 10 100 1000 2000 4000 8000 16000

bench2: $(SIZES2:%=q_%) $(SIZES2:%=d_%) shadow_mapping.fut
	futhark bench --backend=c --json bench_c.json shadow_mapping.fut
	futhark bench --backend=opencl --json bench_opencl.json
	↪  shadow_mapping.fut

clean2:
	rm shadow_mapping
	rm shadow_mapping.c
	for i in $(SIZES2); do \
		rm q_$$i; \
		rm d_$$i; \
	done

shadows: bench2 clean2 plot_bench

all: bench1 clean1 bench2 clean2 plot_bench
```

Appendix B

Proofs

B.1 Work and Span of First-order Combinators

In Section 5.3, we defined the work and span of `replicate`, `unzip` and `transpose`. We cannot formally prove these definitions are correct, as we do not have access to the actual implementation of `replicate`, `unzip` and `transpose` In Futhark. In Listing A.3 are listed custom implementations of these first-order operators. Assuming these implementations are representative, we can infer that the work and span of `replicate`. `unzip` and `transpose` must be :

$$
\begin{aligned}
W(\texttt{replicate}\ n) &= O(n) + O(n) = O(n),\\
S(\texttt{replicate}\ n) &= O(1) + O(1) = O(1),
\end{aligned}
$$

$$
\begin{aligned}
W(\texttt{unzip}\ e) &= 2O(n) + 2O(n) = O(n),\\
[\![e]\!] &= [(a_1, b_1), \ldots, (a_n, b_n)]\\
S(\texttt{unzip}\ e) &= 2O(1) + 2O(1) = O(1),\\
[\![e]\!] &= [(a_1, b_1), \ldots, (a_n, b_n)]
\end{aligned}
$$

$$
\begin{aligned}
W(\texttt{transpose}\ e) &= O(m) \cdot O(n) = O(nm),\\
[\![e]\!] &= [a_1, \ldots, a_n][b_1, \ldots, b_m]\\
S(\texttt{transpose}\ e) &= O(1) \cdot O(1) = O(1),\\
[\![e]\!] &= [a_1, \ldots, a_n][b_1, \ldots, b_m]
\end{aligned}
$$

B.2 Associativity Proofs

Figure B.1: Full proof of **Equation 5.27**

$$
\begin{aligned}
1: \ \big((c_1,h_1')\ \texttt{occlude}\ (c_2,h_2')\big)\ \texttt{occlude}\ (c_3,h_3') &= (c_1,h_1')\ \texttt{occlude}\ (c_3,h_3') \\
&= (c_1,h_1')\ , \\
(c_1,h_1')\ \texttt{occlude}\ \big((c_2,h_2')\ \texttt{occlude}\ (c_3,h_3')\big) &= (c_1,h_1')\ \texttt{occlude}\ (c_2,h_2') \\
&= (c_1,h_1'), \\
2: \ \big((c_1,h_1')\ \texttt{occlude}\ (c_2,h_2')\big)\ \texttt{occlude}\ (c_3,h_3') &= (c_1,h_1')\ \texttt{occlude}\ (c_3,h_3') \\
&= (c_1,h_1')\ , \\
(c_1,h_1')\ \texttt{occlude}\ \big((c_2,h_2')\ \texttt{occlude}\ (c_3,h_3')\big) &= (c_1,h_1')\ \texttt{occlude}\ (c_3,h_3') \\
&= (c_1,h_1') \\
3: \ \big((c_1,h_1')\ \texttt{occlude}\ (c_2,h_2')\big)\ \texttt{occlude}\ (c_3,h_3') &= (c_2,h_2')\ \texttt{occlude}\ (c_3,h_3') \\
&= (c_2,h_2')\ , \\
(c_1,h_1')\ \texttt{occlude}\ \big((c_2,h_2')\ \texttt{occlude}\ (c_3,h_3')\big) &= (c_1,h_1')\ \texttt{occlude}\ (c_2,h_2') \\
&= (c_2,h_2') \\
4: \ \big((c_1,h_1')\ \texttt{occlude}\ (c_2,h_2')\big)\ \texttt{occlude}\ (c_3,h_3') &= (c_2,h_2')\ \texttt{occlude}\ (c_3,h_3') \\
&= (c_2,h_2')\ , \\
(c_1,h_1')\ \texttt{occlude}\ \big((c_2,h_2')\ \texttt{occlude}\ (c_3,h_3')\big) &= (c_1,h_1')\ \texttt{occlude}\ (c_2,h_2') \\
&= (c_2,h_2') \\
5: \ \big((c_1,h_1')\ \texttt{occlude}\ (c_2,h_2')\big)\ \texttt{occlude}\ (c_3,h_3') &= (c_1,h_1')\ \texttt{occlude}\ (c_3,h_3') \\
&= (c_3,h_3')\ , \\
(c_1,h_1')\ \texttt{occlude}\ \big((c_2,h_2')\ \texttt{occlude}\ (c_3,h_3')\big) &= (c_1,h_1')\ \texttt{occlude}\ (c_3,h_3') \\
&= (c_3,h_3') \\
6: \ \big((c_1,h_1')\ \texttt{occlude}\ (c_2,h_2')\big)\ \texttt{occlude}\ (c_3,h_3') &= (c_2,h_2')\ \texttt{occlude}\ (c_3,h_3') \\
&= (c_3,h_3')\ , \\
(c_1,h_1')\ \texttt{occlude}\ \big((c_2,h_2')\ \texttt{occlude}\ (c_3,h_3')\big) &= (c_1,h_1')\ \texttt{occlude}\ (c_3,h_3') \\
&= (c_3,h_3')
\end{aligned}
$$

Figure B.2: Full proof of **Equation 5.31**

$$
\begin{aligned}
1 : &(c_1 \texttt{ fill } c_2) \texttt{ fill } c_3 = c_2 \texttt{ fill } c_3 = c_3 \,, \\
&c_1 \texttt{ fill } (c_2 \texttt{ fill } c_3) = c_1 \texttt{ fill } c_3 = c_3 \\
2 : &(c_1 \texttt{ fill } c_2) \texttt{ fill } c_3 = c_1 \texttt{ fill } c_3 = c_3 \,, \\
&c_1 \texttt{ fill } (c_2 \texttt{ fill } c_3) = c_1 \texttt{ fill } c_3 = c_3 \\
3 : &(c_1 \texttt{ fill } c_2) \texttt{ fill } c_3 = c_2 \texttt{ fill } c_3 = c_2 \,, \\
&c_1 \texttt{ fill } (c_2 \texttt{ fill } c_3) = c_1 \texttt{ fill } c_2 = c_2 \\
4 : &(c_1 \texttt{ fill } c_2) \texttt{ fill } c_3 = c_1 \texttt{ fill } c_3 = c_1 \,, \\
&c_1 \texttt{ fill } (c_2 \texttt{ fill } c_3) = c_1 \texttt{ fill } c_2 = c_1
\end{aligned}
$$

Appendix C

Simplification of rotation matrix

We can simplify Equation 2.5 and Equation 2.6 to a more computationally friendly definition by setting $\psi = \frac{1}{2}\pi$.

Equation 2.5 can be simplified:

$$\begin{aligned}\overrightarrow{OA_k} &= z_k \cdot \begin{pmatrix}\cos(\theta)\cdot\cos(\psi/2) - \sin(\theta)\cdot\sin(\psi/2)\\ \cos(\theta)\cdot\sin(\psi/2) + \sin(\theta)\cdot\cos(\psi/2)\end{pmatrix}\\ &= z_k \cdot \begin{pmatrix}\cos(\theta)\cdot\cos\left(\frac{1}{4}\pi\right) - \sin(\theta)\cdot\sin\left(\frac{1}{4}\pi\right)\\ \cos(\theta)\cdot\sin\left(\frac{1}{4}\pi\right) + \sin(\theta)\cdot\cos\left(\frac{1}{4}\pi\right)\end{pmatrix}\\ &= z_k \cdot \begin{pmatrix}\cos(\theta)\cdot\frac{\sqrt{2}}{2} - \sin(\theta)\cdot\frac{\sqrt{2}}{2}\\ \cos(\theta)\cdot\frac{\sqrt{2}}{2} + \sin(\theta)\cdot\frac{\sqrt{2}}{2}\end{pmatrix}\end{aligned} \tag{C.1}$$

Multiplying Equation 2.5 by $\sqrt{2}$ results in

$$\begin{aligned}\overrightarrow{OA_k} &= z_k \cdot \begin{pmatrix}\cos(\theta)\cdot\frac{\sqrt{2}}{2} - \sin(\theta)\cdot\frac{\sqrt{2}}{2}\\ \cos(\theta)\cdot\frac{\sqrt{2}}{2} + \sin(\theta)\cdot\frac{\sqrt{2}}{2}\end{pmatrix}\cdot\sqrt{2}\\ &= z_k \cdot \begin{pmatrix}\cos(\theta) - \sin(\theta)\\ \cos(\theta) + \sin(\theta)\end{pmatrix}\end{aligned} \tag{C.2}$$

This simplication only has the side effect, that the length of $\overrightarrow{OA_k}$ becomes $\sqrt{2}$ instead of 1, which has no noticable impact on the algorithm.

A symmetrical derivation shows that:

$$\overrightarrow{OB_k} = z_k \cdot \begin{pmatrix}\cos(\theta) + \sin(\theta)\\ -\cos(\theta) + \sin(\theta)\end{pmatrix} \tag{C.3}$$

It then follows from Equation 2.1 that:

$$A_k = \begin{pmatrix}\cos(\theta) - \sin(\theta)\\ \cos(\theta) + \sin(\theta)\end{pmatrix} + \begin{pmatrix}x_0\\ y_0\end{pmatrix} \tag{C.4}$$

$$B_k = \begin{pmatrix} \cos(\theta) + \sin(\theta) \\ -\cos(\theta) + \sin(\theta) \end{pmatrix} + \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \tag{C.5}$$

The attentive reader will realize that the definition of (`plx`, `ply`) and (`prx`, `pry`) in **Listing 2.5** differs somewhat from **Equation C.4** and **Equation C.5**. We may infer that the implementation actually renders inverted frames. This, however, is not perceptible in practice.