Computer Science C, Degree Project, 15 Credits

Baking And Compression For Dynamic Lighting Data

Tom Olsson
Computer Engineering Programme, 180 Credits
Örebro University, Sweden, Spring 2015

Examiner:
Martin Magnusson
Supervisor:
Daniel Canelhas

Industry supervisors:
Daniel Johansson
Mikael Uddholm
Torbjörn Söderman

Örebro Universitet
Institutionen för
naturvetenskap och teknik
701 82 Örebro

Örebro University
School of Science and
Technology
SE-701 82 Örebro, SWEDEN

Abstract

This report describes the development and prototype implementation of a method for baking and compression of lightmaps, in an environment with dynamic lights. The method described can achieve more than 95 % compression efficiency, and can be easily tuned with only two parameters. Even without specific tuning, the prototype consistently achieves signal-to-noise ratios above 30 dB, reaching 60 dB in some scenes.

Compression is achieved in four steps, first by using image segmentation and function approximation to reduce the data-size and then using a predictive quantizer approach based on the PNG-filters together with an open-source compression algorithm. Both compression and decompression can be adapted for asynchronous and multi-threaded execution.

Sammanfattning

Denna report beskriver utvecklingen av en metod för bakning och komprimering av dynamiska ljuskartor, i renderingar med dynamiska ljuskällor. Metoden uppnår mer än 95 % storleksreduktion, och kan enkelt anpassas för olika ljuskartor med två variabler. Även utan specifika anpassningar uppnås en signal-to-noise nivå över 30 dB, och närmare 60 dB i vissa scener.

Komprimering sker i fyra steg, först genom bildsegmentering och linjär funktionsapproximation, följt av predictive quantization och en vanlig komprimeringsalgorithm. Både komprimering och dekomprimering kan anpassas för asynkron och flertrådig exekvering.

Acknowledgement

This report is original and unpublished by work exclusively performed by the author, T. Olsson.

A big thanks to **DICE** for the internship, and to all fantastic colleagues who made my ten weeks fantastic. Special mentions also for my supervisors Daniel Johansson, Mikael Uddholm, and Torbjörn Söderman at **DICE** for their great encouragement and enthusiasm, and pushing me towards excellence. A special thanks also to Jan Schmidt for his input and domain knowledge, without which this would have been much harder.

Finally, a great thanks to my university supervisor Daniel Canelhas, for his great knowledge, enthusiasm and interest during this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Many modern games use a multistage rendering pipeline where rendering is done on both the graphics processing unit (GPU) and the central processing unit (CPU) simultaneously. In rendering technology the GPU is often referred to as a server, while the CPU is referred to as a client[I]. With careful planning this separation allows simultaneous and partially asynchronous execution, as long as both client and server finish at the same time. If either takes too long, there will be freezes in the game resulting in a lower framerate. Modern console games are often optimised to run at either *30* or *60* FPS (frames per second), which gives slots that are either 33 ms or 17 ms long. This time is then broken down into separate parts that need to occur during execution, e g animation, simulation, rendering, or loading.

[I] when using OpenCL and CUDA the server is called *device* and the client is called *host*

These time constraints can be met by baking parts of the data during production to minimise processing needs, which comes at the cost of reduced dynamics in the rendering or large game sizes. In modern games, more and more game elements such as lights, buildings, and props (*scene objects*) are expected to be dynamic. Destructible environments for example, have gone from being a gimmick or very limited feature in older games to being a core gameplay element of modern first-person shooters. Because of this the possibilities for baking decrease as the number of scene permutations increase. However, the cost of simulating this dynamic game content eventually reaches the point where the content is limited by execution time. This breakpoint is easier to control on consoles as every user has the same hardware, while computers are more diverse.

However, some objects have a very deterministic dynamic behaviour, or are subject to very slow changes. This type of object can be dynamic over the course of a gaming session or an hour, but nearly identical for many consecutive frames or even several hundred frames. This property means that many calculations are done needlessly while it is hard to optimise performance based on ‘this code will sometimes execute’.

One example of this is lighting data, in so called physically based rendering engines. These engines strive to achieve realistic results by basing lighting simulations on empirical formulae. This means that an in-game sun may seem static from one minute to the next, but over the course of an in-game hour it will have visibly repositioned itself. Moreover, this rate of change is likely to be globally large in the morning and evening when the sun is low, and globally small during midday and midnight.

1.2 Project

The purpose of this project was to solve the problem presented in section 1.1 on the preceding page for the lighting data mentioned. The general goal was to analyse the data and available compression methods, and then propose a baking method that allows both good quality and dynamic content. The challenge is to create data small enough to be distributed in a finished product, such as on a disc or via digital distribution, as well as used when rendering in the game.

There is also a second part to the project of investigating how the compression tool can fit with the existing systems and pipelines used to create and run the game.

This also made the goals hard to define, as both “good quality and dynamic content” and “small enough” were not only relative to the original uncompressed data but *also* also to the game as a whole. An integral part of the project was therefore development of the goals.

The project was defined as three different stages: defining the data, research about compression techniques optimal for the data, and finally implementation. The stages between these points then became natural points for updating the goals and the direction of the project.

The first stage defining the data can be found in chapter 4 on page 13, which presents the data to be compressed and analyses it in various ways. This chapter ends with a definition of the requirements for the compression pipeline.

The second stage is found in chapters 5-6, which describe compression algorithms as well data manipulation and structures.

The first part of the research was focused on compression algorithms found in chapter 5 on page 21 and the results of this research are shown in section 7.1 on page 33. This was proposed on a meeting with the industry supervisors, but it was decided that more effective compression methods were needed.

The second part of the research took place because of this decision, and is shown in chapter 6 on page 27. This part focused on the data manipulation and data structures. The results from this part are then shown in section 7.2 on page 35. This proposal was shown on a meeting as before, and it was decided to continue with implementation of the pipeline.

The implementation of the compression is described after the two proposals in chapter 7 on page 33, and describes the implementation, mathematics, as well as tuning of the different compression stages.

The results of the implementation are described in chapter 8 on page 49. It includes measurements of compression efficiency and execution time. It also contains visual and quantitative measurements of the loss from the compression.

Finally, chapter 9 on page 55 discusses the results in relation to the goals, as well as the evolution of the project.

1.3 Goals

This project had two goals based on the problem presented above.

The first goal was to *investigate*, *compare* and *propose* suitable compression methods based on data *structure* and *size*, as well as *performance* and *quality requirements.*

This goal was tracked and updated in regular meetings with the industry supervisors, and fulfilled when a satisfactory compression method was found. The preliminary deadline was at the midpoint of the thesis project, but was delayed with the addition of the second research period.

The second goal was to *show* how this can be implemented in the existing pipeline, as well as to produce a *prototype* to demonstrate the compression capabilities.

This goal was fulfilled by the end of the thesis project but was reduced in scope in proportion to the increased scope of the search.

1.3.1 Requirements

At the end of the thesis project the following was shown:

- The available compression techniques and their benefits and drawbacks

- Possible compression techniques specific for this data
- A working prototype tool

Chapter 2

Background

This chapter gives a short background both to lighting in games and to compression methods. The lighting technology will only be briefly mentioned here, while a much deeper description of compression technology can be found in chapter 5.

This chapter also includes a list of common abbreviations and domain-specific expressions at the end.

2.1 Lighting in games

A very important part of the visual aesthetics in a game is the *lighting*. Though serving a very practical purpose of adding depth to a rendering, it has also been noted that the lighting in a game environment even can influence the users' performance [1]. It is therefore important as a developer or artist to take care when designing lighting in a game.

There are two distinct types of lighting used in games referred to as *direct* and *indirect* lighting. These are most clearly defined as each others opposite: direct lighting has **not** undergone diffuse reflection, while indirect lighting has done so. A diffuse (also called matte) surface has the property of dispersing parallel incoming light into different and unrelated directions. Sometimes, global illumination is used in place of indirect lighting but can also mean a combination of direct and indirect lighting. In modern games both indirect and direct lighting is used used to create realistic environments.

The classic example of lighting simulation is the *rendering equation*, a physically based rendering model which which was based on the conservation of energy [2, 3]. This equation is still the basis for realistic rendering models. However, it is also very expensive to compute and it is therefore simplified in various ways to make it possible to calculate in real-time.

The first rendering algorithm used commonly in games was the the *Phong shading model*, which was later modified to create the much faster *Blinn-Phong model* [4, 5]. This model defines three parts of lighting: *specular*, *diffuse*, and *ambient*. The specular component is the reflection directly from light source to eye via the surface, the diffuse term is direct illumination from light to surface, and the ambient term approximates surface-to-surface reflection. This model completely ignores both energy and bouncing light, making it much faster than the rendering equation.

A more modern approach could instead combine a Phong-Blinn shading model with *ray tracing*, *sampled lights*, or *cubemapping* to simulate actual lighting that travels from sources and between surfaces, to calculate the illumination for a point. These approaches are more directly linked to the rendering equation, and often use *irradiance* as a measurement of illumination strength. Irradiance is a measure of the incoming energy over an area, and the emitted energy is called radiance and may be used to define realistic light sources.

While the traditional lighting can be computed with relatively low cost on a GPU (graphics processing unit), the indirect lighting is often based on mathematical models not easily integrated into a rendering pipeline [6]. On a CPU on the other hand they are easy to implement but instead end up being expensive to do in real-time. They are therefore further simplified to use partial solutions, integration or baking [7].

One baking approach is called *lightmapping*. It is a technique used for storing lighting data for objects, and allows a developer to precompute illumination data before execution in order to save computation cycles when running the program. This technique became widely known when used by id Software's Quake. The general drawback of lightmaps however is that they are static, and hence cannot be used to illuminate dynamic objects nor does classic lightmapping allow moving lights. Though the word *mapping* implies the use of textures, there is no strict definition: generally any technique for precomputing lighting data can be referred to as lightmapping.

All the techniques discussed above however fail or become very expensive in one situation: dynamic objects. One solution to this problem is the usage of lighting probes, which were originally developed for illuminating static objects [8]. This approach uses spherical harmonics to sample incoming light at preset points, and then interpolating between these for intermediate positions to illuminate moving objects. These probes can be precomputed similar to how illumination data is precomputed for static objects, and can save a lot of time in real-time execution [9] .

2.2 Compression

Compression is an operation that can be performed for data in order to reduce the storage size. The algorithms for compression can be separated as being either *lossy* or *lossless*. This denotes if the original data can be reconstructed perfectly from the compressed file (without loss, i.e. lossless), or if some information will be lost [10]. As a general rule, a lossy algorithm will be able to reduce the size of a file more than a computationally similar lossless algorithm. For video and audio, a compression algorithm is often referred to as a *codec*.

Lossy algorithms are domain specific and uses research related to that domain in order to minimise the *perceived* loss [10]. In the case of audio compression this might be removing inaudible parts, while a video approach might downsample every other frame and upsample those in real-time instead, for example by doing trilinear interpolation.

Lossless algorithms are more general, and the data-specific differences between algorithms are based on whether an algorithm operates on the bit- or byte-level. The lossless algorithms are grouped primarily into two categories, *entropy coding* and *dictionary coding*.

Entropy coding algorithms attempt to reduce the bit-size of common uncompressed symbols[I] by using probabilities for occurrence. Then the algorithm assigns short bit-sequences to symbols with high occurrence, and opposite for those with low probability [11]. These algorithms commonly use *prefix-free* coding, which means that no encoded sequence is the beginning of another sequence. For example, the alphabet [0, 10, 110, 111] allows a bit sequence to unambiguously identify a compressed symbol, under the limitation of being at most 3 bits long, or ending with a 0.

[I] a piece of data, such as one or several bytes

Dictionary coding algorithms compress by attempting to create *sequences* of repeated symbols which are referenced as needed when parsing the file. These words are stored in a *dictionary*, and when a match is made a reference to the dictionary is stored instead [12, 13]. This allows a commonly occurring sequence to be represented by only a few bits when compressed. Common implementations use static dictionaries, sampled dictionaries, sliding, or dynamic dictionaries depending on the type of data and size.

There are also a few other algorithms that do not fit in these categories that use differential, sorting or counting algorithms to compress the data [14].

2.3 Definitions

Unless otherwise noted or obviously inferred from context, these are common terms and their definitions in this report.

Atlas	A collection of textures
Bake	Compute and save before execution
BPC	Bits Per Character
Chart	A texture inside an atlas
Client side	Execution on the CPU
CPU	Central Processing Unit
DOD	Data-oriented design
FPS	Frames Per Second
GPU	Graphics Processing Unit
Irradiance	Incoming light intensity, energy
LSB	Least significant bit
MSB	Most significant bit
OOD	Object-oriented design
Radiance	Outgoing light intensity, energy
Rasteriser	Software and/or hardware that transforms game objects from world-space to screen-space
Renderer	Software and/or hardware that makes game objects appear on the screen
Server side	Execution on the GPU
Symbol	An abstraction of the input data: for example a character or pixel (compare to **word**)
Word	One or more bytes, in the context of data

Chapter 3

Methodology and tools

This chapter describes the methodology and tools used during the thesis project, as well as other resources of importance.

3.1 Methods

This thesis project was conducted in two main phases, which can be seen below.

Part 1: Research

- Define the characteristics of the lighting data
- Research available compression techniques
- Research possible methods for data-specific compression

Part 2: Software design

- Research integration requirements to fit the prototype into the existing development environment
- Define and implement the model
- Implementation of the prototype

The whole project followed a SCRUM-methodology with daily scrum meetings, two-week sprints and a continously updated backlog and sprint-log.

The research was conducted using a literature review approach to get a good overview of the two areas involved. The literature review was conducted in three steps, with each step narrowing the search subject towards more relevant information.

Originally, the algorithm selection was intended to be done using a decision-matrix approach. However, many of the algorithm have such minute differences that they are hard to weigh against each other. Instead, the algorithm selection was done using an elimination method to narrow the choices. The final choice was then made from this smaller set of algorithms using a logical reasoning method.

The implementation was done using a SCRUM-programming method, by iteratively adding features in a controlled manner. Where it was applicable unit tests were also added to make sure the functionality was not broken in any step.

Finally, the results were measured both using visual inspection of a difference image, as well as using the signal-to-noise ratio and RMSE-measurements.

3.1.1 Literature review: Compression

The literature review was conducted in three general steps of searching. The first step was the *general* step. This meant finding very general sources to establish a knowledge base for further search as well as finding more keywords.

Common keywords in this step were: *overview, review, introduction, compression, lossless, lossy, algorithm.*

The second step was the *weeding* step. The purpose of this step was to take all the new keywords and information from the first step and use it to find current and accurate research, as well as to find compression algorithms and approaches that were relevant for this project.

Common keywords in this step were: *Lempel-Ziv, LZ78, LZ77, entropy, dictionary, encoding, video compression, efficient compression, fast compression, fast decompression, Huffman, arithmetic, DEFLATE, maximum compression, comparison.*

The third step was the *detailing* step. The purpose of this step was to find primary sources and implementation details for the results from the previous step, and forms the bulk of the review. Some time was also spent trying to find modern work based on the well-documented algorithms. Most of the keywords in this step were the same as in the previous step, but a few more were added and the search terms were formulated to be more precise.

Added keywords in this step were: *benchmark, numerical system, adapted, improved, faster, better, harder, hardware-accelerated.*

Searching was done using four different search engines, in order of importance: *Scopus, IEEE Xplore, Google Scholar, and Google.* Though the first two were used primarily, I employed a strategy of chaining these together by starting with Google to find references to articles, techniques and algorithms. Then these were traced backwards through the chain of importance until a reliable and/or original source was found.

In the cases where it was relevant wild-card searches were used with the stem of words, such as searching for “adapt” instead of “adapted”. As many keywords can also work as both as an adjective, verb or noun these forms were also used for searches.

3.1.2 Literature review: Image segmentation and data structures

The extended literature review was based on the methodology as the compression literature review, with the same three general steps. As a lot of detail were already described in the first part, there was a more restrictive selection of topics and sources.

The first step was used to find information about general data structures and compressor optimization, as well as data description and transformation. Common keywords were: *voxelisation, vectorisation, data transformation, data optimisation, sparse coding, file structure, data structure.*

The second step was focused on techniques for data identification and data separation as the structuring was deemed a by-product from the technique used. Common keywords used were: *k-d tree, spanning trees, octree, quadtree, image graphs, image segmentation, foreground segmentation, object identification, blocking, patching.*

The third step was used to combine the information from the first and second step to both define data structure and storage, as well as methods to generate the same structure. No more keywords were used in this step, but keywords both from this extended review and the previous review were used to search for precise results.

3.2 Tools

Most of the work was done inside the existing development environment at **DICE** using their proprietary tools and pipelines. The primary language for development was originally **C++**, with some **C#**.

The final prototype for the compression was created using Python to allow faster iterations. Various non-canonical Python packages were used such as **VTK**, **wxPython**, and **Mayavi**.

3.3 Other resources

3.3.1 Software

ParaView 4.3.1 was used to visualise some of the data

Chapter 4

Data and system

This section describes the system in which the prototype is supposed to exist, as well as the data that shall be compressed.

4.1 System

The purpose of the research was to find a practical compression method for a set of precomputed lightmaps. The flow-chart in fig. 4.1 shows how the related data is created by the editor and later used in the game. The two steps labeled *compress* and *decompress* will be the primary focus, but obviously they need to conform to the greater system. It is important to note that this is not a constant flow as the left part (until *File*) is executed during development while the right part is executed when a scene is loaded in the game.

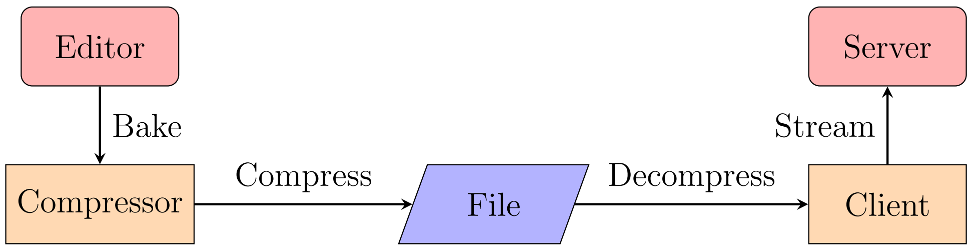

Figure 4.1: Data flow from editor to runtime

This allows the compression to be arbitrarily complex while the decompression must occur in near real-time. It also means that the compressed file-size needs to be small enough to be distributed on a physical media or via downloads. Furthermore, the decompressed data shall be of similar quality to the original file.

There are two restrictions defined for the allowed distortion in the decompressed data. Firstly it must not needlessly reduce spikes in illumination such as when a surface normal is parallel to the incoming light and having the highest incoming energy. Secondly it must not blend between neighbouring elements as the irradiance textures are very low resolution: one pixel of irradiance data can represent several square meters.

4.1.1 Client capabilities

The client side has an existing framework for streaming linear media such as video and animation data.

4.1.2 Server capabilities

As the data will be used on a GPU it may be possible to use 3D-textures or texture arrays to store the data, as the baked data uses a format that can be used directly as a texture.

4.2 Data structure

The baked output data from the editor contains three different parts that describe the lighting in a scene. The first two parts are similar to traditional lightmaps in the sense that they are textures, and contain irradiance magnitude and direction [15], while the last part contains light probe data and is used to illuminate dynamic objects [8]. The total size of this data, across the whole game equals roughly 200 MB per timepoint. This is split across 42 *lighting systems* each being roughly 5 MB big. As the lighting systems are loaded dynamically as the player moves in the game, they need to be compressed dynamically.

As noted in the previous section, one pixel of irradiance data can represent several square metres. This is an optimisation based on the assumption that nearby areas will receive similar amounts of indirect lighting, and can be very realistic with just linear interpolation. This small texture size brings an overhead of pointers and unnecessary context switches on texture units[I] when used in a renderer. In order to alleviate this issue the smaller textures are stored in a *texture atlas* which combines many smaller textures. Each part inside this atlas is then called a *chart*.

[I] graphics hardware dedicated to textures

4.2.1 Irradiance

The irradiance map contains a texture of values describing the incident irradiance in a point. This map can use either the OpenGL format RGB9E5[II] for a size of 32 bits per pixel, or one *IEEE 754* half-float

[II] 9 bits of mantissa per element, and a shared 5-bit exponent

(16 bits) per channel for a total size of 64 bits per element.

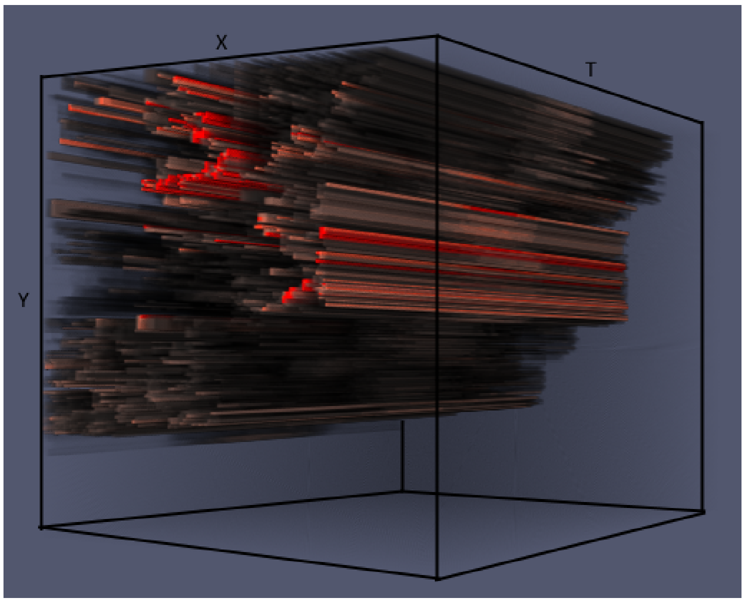

Figure 4.2: 384 frames of temporal irradiance data rendered with z as the time-axis, and red channel plus alpha scaled by intensity.

An example of data for one system can be seen in fig. 4.2. Both the red and alpha channels are based on the intensity in the point. As can be seen there are many points that are very low intensity throughout most of the day only being illuminated briefly at certain points, as well as many points that are permanently illuminated with almost constant intensity throughout the day. Though hard to see, there are also areas that are briefly illuminated at various points during the day and therefore fade in and out. As can be seen, a large part is constantly empty as well.

To get a clearer picture of the data structure fig. 4.2 was flattened to produce fig. 4.3 on the following page. Though there may be points that are actually used but never illuminated it gives a more defined view of the data. Noteworthy is the very inefficient atlas which utilises between half and two thirds of the texture. This is an execution optimisation since texture units are more effective on texture dimensions that are a power of two, but makes it harder to fill the atlas as each time a chart overflows the atlas it needs to be at least doubled in size[I].

[I] the logic behind this behaviour is the same as fanout fill-rate in a B+-tree

4.2.2 Irradiance direction

As a complement to the irradiance the data also contains textures which can either show the aggregated direction for incoming “white” irradiance or per-channel direction for incoming red, green or blue irradiance. The data represents the direction as one vector $[X, Y, Z, W]$ per element, or

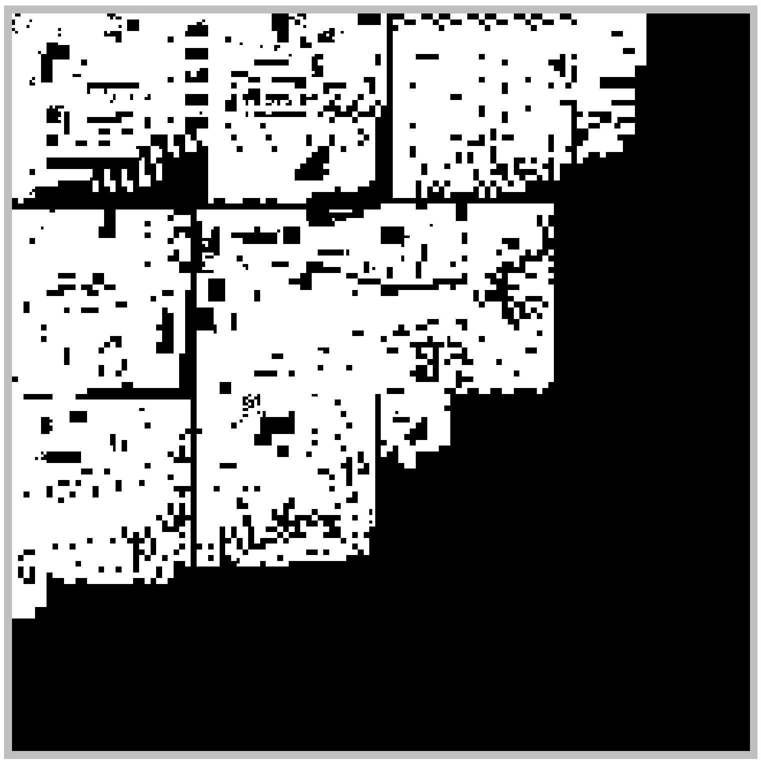

Figure 4.3: Flattened view of system illumination where white pixels represent an element that is non-zero at least once, and black pixels are always empty. The grey background was added to emphasise the borders and is not part of the texture.

three vectors per element if stored per color channel. This makes the data either 32 bits or 96 bits per element stored in one or three 2D-textures.

In fig. 4.4 on page 19 an example of these directional textures can be seen. The base scene is the same as the one used in fig. 4.2 on the preceding page though visualisation is a lot different. In the scene used here the direction seems to be mostly the same, but detailed inspection shows small differences between the channels that could be caused by nearby coloured surfaces or coloured light-sources.

4.2.3 Probe data

The probe data contains sampled irradiance in a set of points, represented using Spherical Harmonics coefficients. These may be stored either as first-order (L1) or second-order (L2) harmonics. In both cases, the data is separated per color channel. This data does not have a spatial representation like the other parts, instead being just a list. It is also much smaller, accounting for on average 3 % of the total data-size in the data-set used.

Spherical harmonics can be calculated and stored in many ways, but the general one used for lighting is as a series of *[function, coefficient]* pairs, which extends the **Fourier series** to three dimensions. The Fourier series allows approximation of a function f using as a series of function such that $\hat{f} = C_1 \times b_1 \ldots C_n \times b_n$ where C is a coefficient and b is a base function. The exact same approach can be applied in three dimensions. This allows very compact representation of illumination in an area that can be computed using a simple dot product.

In the case of L1 coefficients there are four floats per channel. These represent an ambient term and the three cardinal basis functions: x, y, z, for a total size of 48 bytes [8].

In the L2 case there are 9 flats per channel for a total of 108 bytes per probe. These represent the same bases as the L1 spherical harmonics and then five more quadratic bases: xy, yz, xz, z^2, $x^2 - z^2$ [8].

4.2.4 Data summary

Part	Dimensions	Format/element	Size
Irradiance	X-Y-T	RGB9E5/FP16 (RGBA)	32 or 64 bits
Direction	X-Y-T	RGBA8	32 or 96 bits
Probe data	Index-T	FP32	48 (L1) or 108 (L2) bits

Table 4.1: Summary of the input data that shall be compressed. The dimensions shown relate to the full data-set, though each element may also have a specific format

4.3 The data in detail

Remembering the data from section 4.2 on page 14 it may be important to reiterate that large amounts of data is empty, as was shown in both fig. 4.2 on page 15 and fig. 4.3 on the preceding page. It is however hard to quantify exactly how large this emptiness is purely from looking at the images. Another approach for visualising this empty space can be seen in fig. 4.5 on page 20.

This image shows us an interesting phenomenon. Despite the irregularities shown in fig. 4.2 on page 15 the empty space is constant overtime as each line in both diagrams has a constant coloring with no shifts. Though the diagram does not answer the question of **where** the emptiness is, it is not an extreme assumption that it is the **same** elements that are empty constantly[I]. Furthermore, the pictures show very clearly that there are borders that will be good for segmentation, shown as ridges in the textures. For example, in both images there is high potential around row/column 60, 120 and 190.

[I] Consider the opposite: if elements are only sometimes occupied, the figure implies that there are always as many elements that turn on as that turn off. This is highly unlikely.

As the emptiness is time-invariant, further visualisation can be done for a single slice which can be seen in fig. 4.6 on page 20. The spatial distribution is the exact match of a column above, and the cumulative distribution shows two important things. Firstly is that almost exactly half of the texture space is empty. Secondly is that the spikes shown are very prominent, and surrounded by areas with mostly constant emptiness. This was also shown though not as clearly both above and in fig. 4.2 on page 15. Since these form a plateau-like pattern it may prove itself very suitable for segmentation. As a very basic example, consider cutting away the 20 % that is constantly empty throughout the whole atlas, and then reiterating the process. Just the first step, in this case, could remove 36 % of the area.

4.4 Compression requirements

Dimensionality:	2D/3D
Compression type:	lossless primarily[I]
Complexity:	
Compression:	arbitrary
Decompression:	close to real-time
Architecture:	software and hardware

Adaptivity:

- Data structure may change
- Resolution will change
- Interframe changes will roughly be symmetric [II]
- Frame density will not be symmetric[III]

[I] see section 4.1 on page 13

[II] The changes between frame 1 and 2 will be as large as between frame 2 and 3

[III] The frame density will be higher where the lighting is changing fast

4.4.1 Target compression ratio

The target compression ratio is a minimum size reduction of 85 % based on the redundancy and size requirements. This is slightly better than what general compression tools can achieve. While the ratios vary between data-sets, LZMA achieves at least 85 % compression ratio, DEFLATE achieved a minimum of 80 %, bzip2 achieves 75 % and normal Windows ZIP (a DEFLATE-algorithm) achieves 65 %.

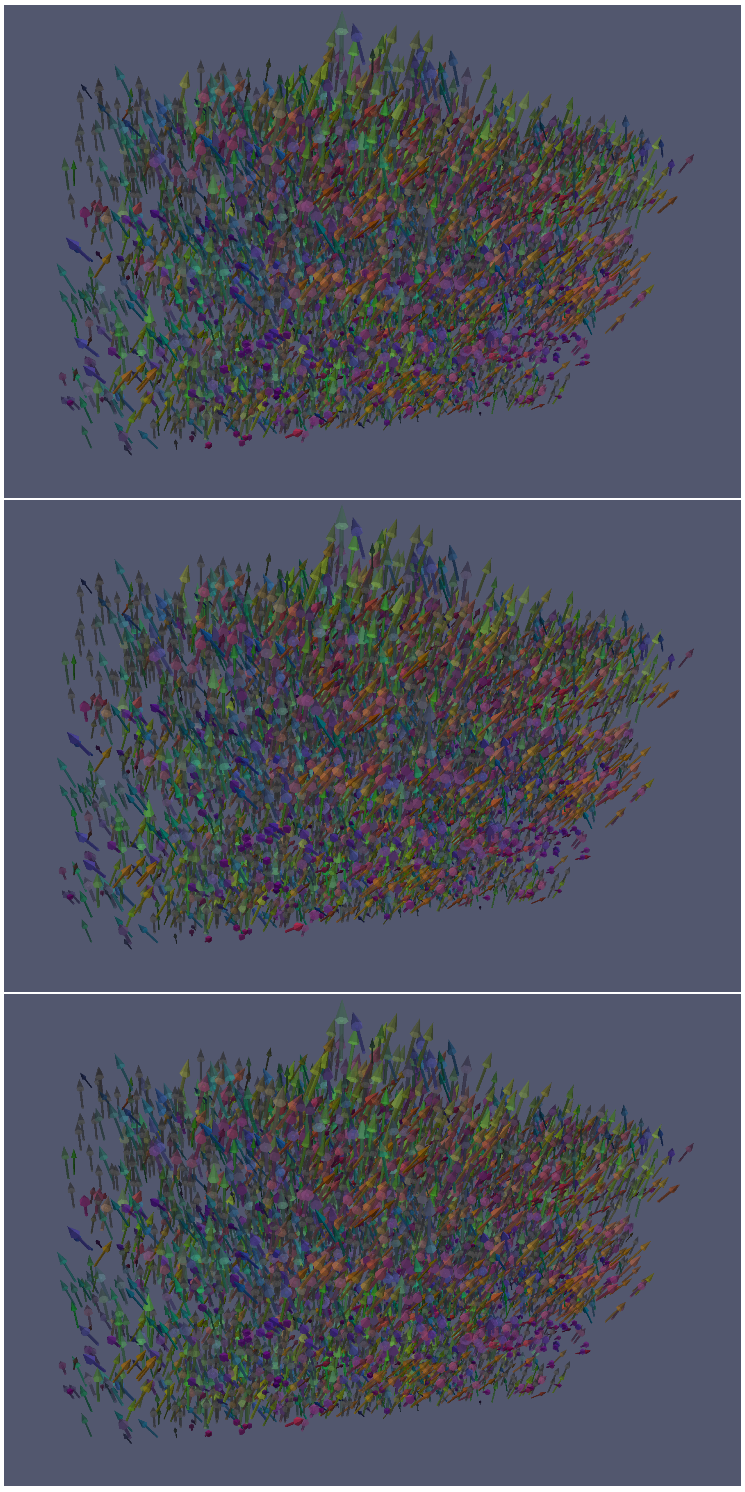

Figure 4.4: 384 frames of temporal irradiance direction data separeted per channel with z as the time-axis, and each arrow being colored by direction where $(x, y, z)\hat{=}(r, g, b)$. Unlike in fig. 4.2 on page 15 the unused texture space was removed here.

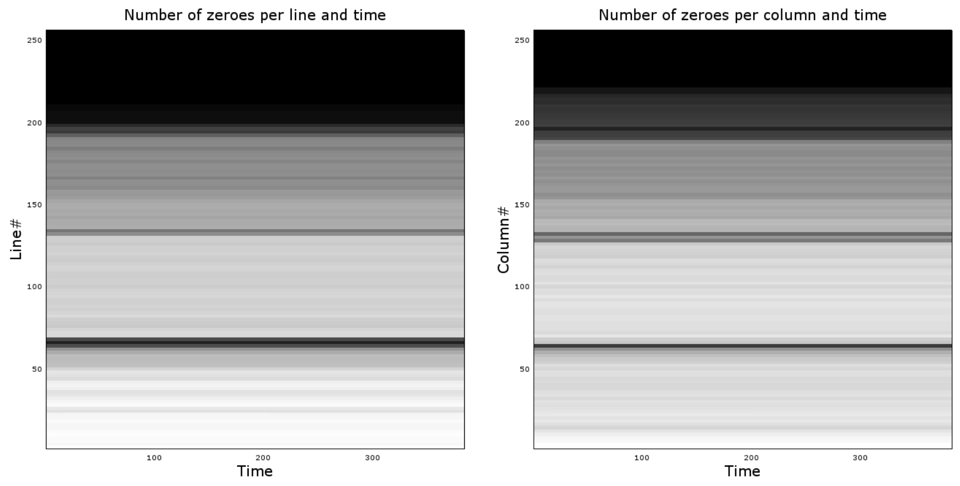

Figure 4.5: Spatial and temporal distribution of empty elements

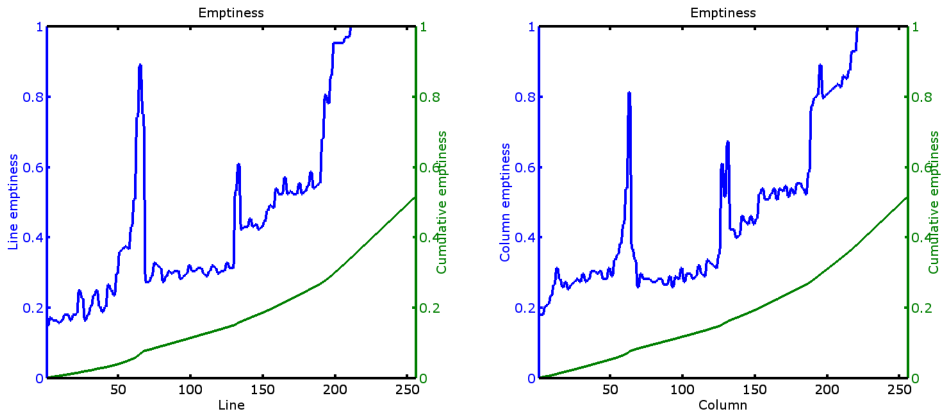

Figure 4.6: Spatial distribution of emptiness and total cumulative emptiness.

Chapter 5

Literature review: Compression algorithms

This section contains a review of common compression algorithms, as well as their strengths and weaknesses.

5.1 Overview

As shown in section 2.2 compression of general data is a thoroughly researched area, with more and more specialised algorithms being created for new types of digital data [16]. Many of these however are based on previous research or show a lot of similarities, and so algorithms are often grouped both based on family and on type. To further complicate matters some compression algorithms are used primarily to prepare data for other algorithms to make them more effective, and some do not perform any compression on their own [10].

A general grouping of algorithms is as *lossy* or *lossless* [10]. A lossy algorithm is domain-specific and attempts to reduce data-size by removing less important content. This may for example be inaudible parts of an audio signal such as very high or low frequencies, which means that the decompressed data file will not be equal to the source file. A lossless algorithm on the other hand uses more general approaches in order to reduce redundancy in files while preserving all the content, so that the original data may be reconstructed completely. It is important to note that this grouping is not forced – a lossy algorithm may be general and a lossless algorithm may be domain-specific but it is harder to achieve.

When studying lossless algorithms there are two different groups: *entropy coding* algorithms and *dictionary coding* algorithms. Entropy coding uses probabilities to assign few bits to common *symbols* and more bits to uncommon symbols in order to achieve a shorter average length [17]. Dictionary coding reduces redundancy in a file by finding repeated sequences of symbols and replacing them with dictionary references [12, 13].

The main difference between the two groups is that a dictionary compression uses knowledge about the domain and may compress based on words, pixels, and other domain entities, while entropy encoding is domain-agnostic and operates on bits and bytes [10].

There are also a few other algorithms that transform and polish the data in various ways, and they will be discussed more in section 5.1.3 on page 25.

As this is temporal image data it also makes sense to look at various video and image compression methods to reduce data size. Video compression algorithms use a combination of intra-frame compression techniques adapted from image-compression and inter-frame compression which attempt to reduce the redundancies between frames, as well as general techniques that are not domain-dependent [16].

5.1.1 Entropy encoding

Entropy, when talking about compression, is a measure of the randomness of a message. Formally it was defined by Shannon as

$$H(x) = -\sum_{i=1}^{N} p_i \log_2 p_i$$

with p_i being the probability of symbol i and gives the average number of bits needed to encode a symbol in x. This number is also the theoretical minimum BPC (bits per character), and algorithms are often compared based on how close to this measure they get [10]. The measure for an actual encoding is given by

$$\hat{H}(x) = \sum_{i=1}^{N} b_i \times p_i$$

where b_i is the number of bits used to encode symbol i. Because of this statistical limit it is often more relevant to discuss entropy coding algorithms in terms of balance between speed/performance and efficiency rather than only pure efficiency.

Huffman coding works by encoding each symbol using an integer number of bits which is the optimal encoding if each symbol is encoded separately [17]. It is however easy to prove that it is only globally optimal if all probabilities are on the form $2^{-k}, k \in \mathbb{N}$, e.g. 50 %, 25 %, 12.5% and so on, which would correspond to the $[0, 10, 110, \ldots]$ alphabet. A set of symbols with probabilities $P = [.9, 0.05, 0.05]$ however would at best be encoded with $[0, 10, 11]$ which has an average BPC of 1.1 compared to 0.57 BPC which is the optimal case.

Arithmetic coding is a generalisation of Huffman coding which codes based on sequences of symbols instead of encoding each symbol separately. This creates a more compact representation at the cost of a more expensive algorithm, which allows BPC-levels close to the theoretical limit [18]. Another approach to arithmetic coding called *range encoding* operates on bytes instead of bits and therefore gains a slight increase in speed with lower efficiency, though the approach is the same.

The general idea of the arithmetic coding is the range $[0, 1)$ is allocated according to the probability for all symbol to occur, so that a symbol with 50 % chance has for example the range $[0, 0.5)$ [18]. When a symbol is read from the stream, that symbols range is subdivided in the same manner, so that the symbol from earlier would occupy the range $[0, 0.25)$. This continues for a certain depth before a *tag* from center of the last encoded range is emitted, followed by a codeword indicating end of sequence.

Consider the alphabet $[1, 2, 3]$ with the probabilities $\left[\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right]$, and encode the sequence 1123. The workflow and nesting of ranges can be seen in fig. 5.1. In the example, the final tag would be $\frac{\frac{7}{36}+\frac{5}{24}}{2} = 0.2013889$.

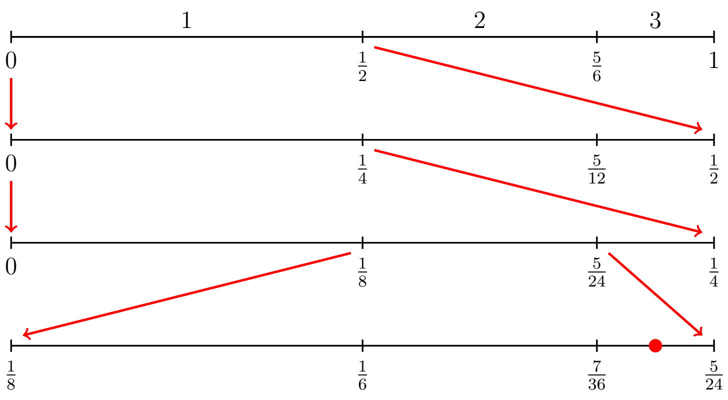

Figure 5.1: Visualisation of arithmetic encoding using nested ranges.

Several variations of the arithmetic coding algorithm exist that attempt to improve performance. The main approaches used in these focus on reducing the computation complexity of the original implementation by using lookup-tables, shifts and additions instead of multiplications and divisions as they require fewer instructions. However, none of these has been able to reach the speed of Huffman coding though beating it in compression [19].

A further improvement upon the arithmetic coding called *asymmetric numeral systems* was proposed in 2009, and claims to combine (or even beat) the speed of Huffman coding with the compression rates of arithmetic coding [20]. It borrows both from arithmetic coding and Huffman coding to create a related set of algorithms that allow a trade-off to be made between speed and efficiency [21]. The original paper however has not received much scientific interest though it has been discussed in compression communities and several implementations have been proposed. Neither of these can verify the claim of **beating** Huffman coding speeds, though several report equal speed and higher efficiency [22, 23, 24, 25, 26].

5.1.2 Dictionary coding

The most common type of dictionary coding is the *Lempel-Ziv* family of algorithms, based on two algorithms by Abraham Lempel and Jacob Ziv in 1977 and 1978 [10]. Sometimes a distinction is made between these two as separate families, as they differ in the way they use dictionaries [12, 13]. The basic operation of any of these algorithms is to iterate over an input stream and attempting to replace incoming data with references to data somewhere else in order to reduce size.

The algorithm proposed in 1977, called LZ77 or LZ1, uses a *sliding window* with a character buffer extending both backwards *(history)* and forward *(preview)* to match incoming characters to previous characters [12]. If a match is found between the preview and the history, the algorithm outputs how far back the match is found *(offset)*, and how many characters can be matched from that point *(length)*. By definition of the algorithm, the length can be larger than the offset, which means that the sequence is repeated fully or partially. This means that the algorithm encodes based on the local context but has no global knowledge.

By contrast, the algorithm proposed in 1978, called LZ78 or LZ2, uses a global dictionary of *sequences*. The algorithm begins with an empty dictionary and an empty sequence, and then iterates over each character in the stream [13]. If the character plus the previous sequence can be found in the dictionary, the character is appended to the sequence and a new character is retrieved. If it cannot be found in the dictionary, a dictionary reference to the sequence is sent to the output[I] together with the character. The concatenated string is then added to the dictionary, and the sequence is emptied.

[I] as the sequence must have been in the dictionary to reach this stage

Three very succesful derivatives from the LZ-family are the *LZMA*, *DEFLATE* and LZP variants. LZMA is a combination of arithmetic coding from 5.1.1 on page 22, Markov Chains and LZ77, making very efficient use of the forward and backward references. This requires a large dictionary and memory usage but it compares well to many other algorithms especially when accounting for performance [27, 28, 29]. A similar composed algorithm is DEFLATE which uses LZ77 and Huffman coding to achieve slightly less compression at higher speeds [28, 29]. LZP on the other hand is a simple algorithm which aims to reduce the length of the offsets in LZ77 by using hash-tables instead of references back into the stream [30]. It has overall poor compression performance on its own but can be used as a preprocessor for other algorithms to improve their performance by more than 10% [31].

Another noteworthy derivative from LZ77 is the *LZ4* codec which has decompression speeds an order of magnitude faster than LZ77, but also less effective compression [28, 29, 32]. It improves efficiency by using variable length coding, restrictive referencing and lookup tables to improve speed at the cost of compression rate [33].

Sequitur or *byte pair* coding is a dictionary coding which uses recursive substitution to remove repetitions in a file. It analyses the input sequence for either matches against previous substitutions, else against the current encoded string. The algorithm follows two basic rules for coding:

Rule I no set of neighbouring symbols shall appear more than once in the stream

Rule II each grammatical rule must apply more than once [34]

In order to achieve compression comparable to other algorithms the grammar is not generally included in the encoded file. Each original instance of a rule is instead left unchanged, and the second time it occurs a rule descriptor is written detailing where a previous occurence can be found. The third time it occurs a reference to the rule is stored. This scheme is used as otherwise any gains from the compression will be consumed by overhead from terminators and the grammar.

5.1.3 Other algorithms

Burrows-Wheeler transform is a data-manipulation algorithm which can be used to sort data so that it is more compressible by algorithms that rely on proximity for compression. It works by rotating and sorting data multiple times, creating a quasi-lexical ordering in the data [35]. This algorithm uses the same logic as second-order language modeling, which is that certain characters are likely to be in sequence most of the time [36]. An example is that a “he” string will likely be preceded by a ‘t’, so by sorting on the “he” all the ‘t’s will group together on the other end of the rotation.

Run-length encoding is a very simple encoding algorithm which encodes a source into *runs*, which contain a symbol and how many of that symbol that occurred in sequence [37]. For example, the word **bookkeeping** could be encoded to b(**o,2**)(**k,2**)(**e,2**)ping. It is therefore most suited to sources with long runs, for example in quantized images or fax-messages. It can also be used for coding data sources which are mostly empty.

5.1.4 Video and image compression algorithms

As mentioned above video and image compression algorithms are generally lossy and in order to achieve a good quality this is unwanted. However, there are some methods that are lossless, or can be both.

One such algorithm is the *discrete cosine transform* (DCT) which encodes a source into a frequency domain, which by itself does not compress. It is however a common step in a chain with other algorithms [38]. It is used for example in JPEG-compression together with quantization and rounding to discard high-frequency content. A related algorithm is the wavelet transform, which transforms signals into a time-scale domain instead of a frequency domain [39]. The DCT is used by normal JPEG coompression, while a wavelet-based method is used in lossless JPEG.

There are also various different algorithms that encode symbols based on their differences. Two common algorithms are the *differential pulse code modulation* and *predictive coding*. The basic idea behind either of these algorithms is to use a heuristic function to calculate a predicted value for the current symbol from the previously encoded symbols, and then encodes only the difference from that predicted value [40, 41]. This can both reduce the impact of noise in the decoded signal and make compression more efficient. Variants of this algorithm is used in lossless JPG and PNG.

Moving on to video compression, section 4.1.2 on page 14 the decompression will happen on GPUs with access to 3D-textures. A large part of the video specific compression is related to reducing redundancy between frames. Often these algorithms use various blocking or prediction techniques to account for moving objects, but in the case of this data all datapoints will be static and change slowly. If the keyframes can be selected to allow linear interpolation between them, a very cheap implementation of inter-frame compression can be constructed using the GPU, *if the linearity can be guaranteed.*

This linearity can be guaranteed by using *piecewise linear approximation* such as the approach by Hamann et al for either a given number of points or a given error tolerance [42]. Another similar approach was also used by Jones to compress distance fields [43] to cull data with a forward predictor. Both of these approaches compress by removing redundancy that can be approximated from other data-points.

Chapter 6

Literature review: Image segmentation and data structures

This chapter describes image segmentation algorithms as well as data structures that are relevant for storing the data.

6.1 Image segmentation

Image segmentation is the act of separating parts of images by some heuristic in order to make it easier to interpret. A major application for image segmentation is as a processing step in computer vision, such as in robots and cameras where foreground and background need to be separated. It is also commonly used in image and video editing programs and used for example as “magic lasso” tools and similar. As with the compression algorithms there are a few basic families of algorithms, and many algorithms are actually combinations of the basic algorithms.

Graph usage for image segmentation is a well-studied area, with two main types of algorithm. The common idea is to treat some or all pixels in the grid as nodes in a graph, and then use graph-theory or other methods to segment the image.

The classic example of graph-based segmentation is the *max-flow/min-cut* method, though modern variations on this method are called *minimum energy segmentation*. The algorithm starts off with atleast two points that represent the foreground and background called *source* and *sink*, and all nodes in the picture are marked with the probability of being either foreground or background [44]. Then all edges are weighted based on the difference between the related nodes, such that edges between nodes with different belonging are ‘weak’ or ‘low energy’, and nodes with the same alignment have 'strong' or 'high-energy' edges. Lastly, the algorithm finds the cut in the graph which minimises the energy, which will also be the cut which separates the most foreground pixels from background pixels.

A merging approach can also be used for segmentation, where the algorithm starts with each pixel being a node. The edges between pixels are then assigned a weight based on the dissimilarities of the nodes it connects. The algorithm will then iterate over the edges and merge nodes into segments wherever the edge heuristic meets the requirements for merging [45]. There are many variations on the edge traversal and heuristics, for example by considering more than the two current nodes, so that more optimal solutions are found.

Two other techniques based on clustering for segmentation uses machine learning to segment images, by grouping them based on similarity only. Two major algorithms of this type exist, *k-means* and *mean shift*. *K-means* is an iterative algorithm that starts by placing k centroids in the feature-space[I] and assigning all elements to the nearest centroid [46]. The centroids are then moved to the centre of mass of all associated elements, and then all assignments are updated. The algorithm is finished when no elements change centroid during an iteration. Each cluster is then a segment.

[I] e g in the RGB colorspace

Mean-shift is similar to k-means, but does not need pre-placed centroids [47]. The algorithm iterates over all points, and for each point it finds the center of mass for the local area[II] instead of the closest centroid. The search area is moved towards this center of mass, and a new center of mass is calculated. This process is repeated until the center of mass does not change. At the end, all points that end their search in the same location are said to belong to the same cluster.

[II] in feature-space

Edge detection algorithms are used in image-processing to find edges and changes in gradients. Most edge detection models are based on first and second order derivatives, quantifying the rate of change at a given position, though some are more algorithmic in nature [48]. Often these models are applied using as a convolution filter over the image, or as a multiplication in the frequency domain. A well-known first-order kernel is the **Sobel** operator, and a well-known second-order kernel is the **discrete laplacian** operator.

A very naïve form of segmentation can be done using thresholds. Such an approach might label all elements below a certain threshold[III] as belonging to one layer and all other pixels to another, or split the range of intensities into multiple thresholds to create several segments. Other approaches use a more local approach by considering the local neighbourhood when selecting layers, so that local variations are not completely lost [49].

[III] for example, the average intensity

6.2 Data structures

When optimising execution for high-performance applications a very common pitfall is the usage of an object-oriented design (OOD) pattern for everything. On modern computers there is a large difference in access speeds for data in the processor cache, which can be several hundred processor cycles [50]. Having to repeatedly fetch data from RAM into the processor cache will introduce large performance drains to transfer data, and it is therefore imperative to optimize the data structures to reduce this.

This general goal can be split into two rules:

Rule I Keep necessary data in memory for as long as possible

Rule II Avoid loading unnecessary data whenever possible

These rules are formulated as an alternative to OOD called *data-oriented design* (DOD). The basic paradigm for data-oriented design is to rotate data-structures by 90 %. This is based on the object-oriented paradigm of storing self-contained objects in arrays, while a data-oriented approach would merge the data from several objects into a multi-object container. The first type is often called *array of structures* while the data-oriented approach is called *structure of arrays*.

This change is most easily seen in code listings 6.1-6.2 and fig. 6.1 on the following page. The purpose of this is to streamline access to parts that may be used together more often. In the example, the data in a Monster would likely be modified per-object in the client but when preparing data for the server such as by creating matrices, it may be more efficient to do so per-type rather than per-object.

Listing 6.1. Object-oriented design

```
struct Monster
{
  vec3 m_position;
  vec3 m_rotation;
  bool m_isAngry;
};
Monster monsters[10];
```

Listing 6.2. Data-oriented design

```
struct Monsters
{
  vec3 m_position[10];
  vec3 m_rotation[10];
  bool m_isAngry[10];
};
Monsters monsterList;
```

As the purpose of this work is to improve performance, this is an important consideration when choosing data structure. In relation to the flowchart in section 4.1 on page 13 it is also important to note that the most important factors are search and insertion as those are the only

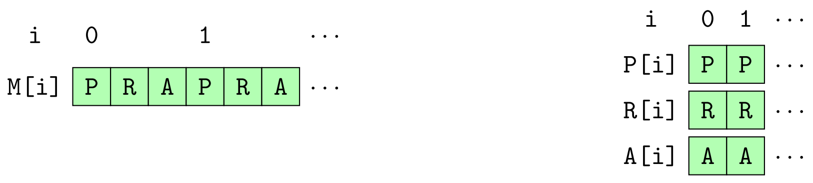

Figure 6.1: Memory-layout of object-oriented approach (left) and data-oriented approach (right). Note that the indexes and number of elements is constant, but the data for each monster has been "rotated".

operations during execution.

6.2.1 Tree structures

Tree structures are ubiquitous in computer science, and are useful for fast retrieval and insertion. Trees are defined either by their branching factor[I] and their dimensionality. For example, a binary tree in three dimensions always splits the volume in two partitions, along one of the cardinal axes. A ternary tree would divide the space into three partitions for each level, and so on. For higher branching factors it becomes possible to create many alignments of the partioning, such as partioning in either a cross-pattern or as four slices in a quaternary tree.

[I] how many children each node has

In games design three types of trees are commonly used for spatial partioning: *binary trees*, *quadtrees*, *octrees* and k-d-*trees*.[II]

[II] The quick reader may notice that the first three correspond to 2^1, 2^2, and 2^3.

A binary tree has a branching factor of two, and can be used for normal ordering, and is often applied for depth-sorting and ordering data that is only comparable in one dimension.

A *quadtree* is a specific type of quaternary tree splitting at the centerpoint [51]. This means that each layer has four times as many nodes as the previous layer, but they are all exactly a quarter of the parents size. The common use cases for quadtrees is for searching in a 2D-area such as a map or for image processing.

Octrees are three-dimensional data structures that are used for searching in 3D-areas such as in a volume, as well as for raycasting and density textures [52].

k-d-*trees* finally are an extension of the binary trees into k dimensions, by cycling through the dimensions when subdividing [53]. k-d-trees are commonly used for nearest-neighbour search as the canonical implementation subdivides through the median at each point, creating a balanced and very efficient search tree.

Tree structures are dependent on the data for complexity, but are usually

$\mathcal{O}(n \log n)$ for searching and insertion.

6.2.2 Arrays, lists and maps

The most basic data structures is the *array*. Arrays are linearly allocated data structures where elements are packed tightly without any gaps, such as the left example in fig. 6.1 on the facing page. Arrays have static indexing and assignment complexity, $\mathcal{O}(1)$, and linear complexity for all other operations, $\mathcal{O}(n)$. An array is generally of static size, but most languages have built-in support for dynamic arrays for example the `std::vector` in **C++** or `ArrayList` in **C#**. This allows dynamic growing, insertion and removal in linear time, which is not possible at all in normal arrays. The other arrays shown on the right side of fig. 6.1 on the preceding page are an example of *parallel* arrays, where data at the same index in different arrays has some form of relation.

A *hash map* or *hash table* is a data-structure optimized for fast lookup for data with a known distribution, by mapping input *keys* to output *values* to create key-value pairs [54]. The key is used as the input to a hashing function which converts it to a numeric index which points to the actual value. Hash maps are often used for lookups where searches are primarily done using non-numeric keys, such as strings. The complexity for searching in a hash map is $\mathcal{O}(1)$, as most common implementations uses an array for the actual data.

6.3 File structure

The file format is also important for optimisation, as unnecessary data in the file increases the file size, as well as increases complexity of loading and storing. There are two common paradigms for file structure: *header-based* and *chunk-based*.

Header-based file structures use a header which contains all the information needed to parse the data in the file. This header usually contains what type of file it is, metadata such as mime-type, what encodings it uses and so on. If the data has multiple parts, it will also contain information about where to find each part. This can make finding specific data in files very efficient as only the header and the relevant data needs to be loaded.

Chunk-based file structures on the other hand encapsulate each separate part alone with all the relevant info. This is a common format for extensible file-types, as a file-reader may simply skip unknown chunks and read whatever it can. It is used for example in PNG where custom chunks can be used to store metadata such as animations or unicode-text. Because of this it is especially useful if all the data will be loaded at once.

Chapter 7

Implementation

This section describes the implementation used in the final compression pipeline in the prototype.

7.1 Compression algorithm selection

Referencing back to section 4.4 on page 18 there are a few strong candidates from the literature review. First the choices in the 2D domain will be discussed and then the third domain will be discussed.

As noted in section 5.1.1 on page 22 the speed of Huffman coding is consistently better than arithmetic coding [19], while arithmetic coding has better compression. The basic compression tests made in section 4.4 on page 18 show LZMA (arithmetic) as top, and then DEFLATE (Huffman) as second not very far behind. As speed is very important the logical choice is therefore to use Huffman coding as a startpoint, and if more effective compression is needed **and** there is execution time available arithmetic coding can be investigated further. Though the asymmetric numeral systems seem to provide both good compression and good speed it seems too experimental still to use in this type of product.

There are a few more alternatives when looking at the dictionary coding alternatives. Sequitur coding, while interesting, has very few implementations and even on text files it does not perform well, with compressed sizes 3-4 times larger than other dictionary coding algorithms [31]. As such, the choice of algorithm then stands between a LZ77 and LZ78 type algorithm. Looking at the data-set shown in fig. 4.2 it is easy to see that the both local and global similarity is high. However, LZ78 requires a global dictionary with all sequences, and as the symbols in this data have varied size it may be hard to optimise without compressing each data separately. In the case of LZ77 and its derivatives there is a more logical blocking approach, and increased local referencing.

As such, the choice of standard compression algorithm becomes Huffman with LZ77 or one of its derivatives. Looking again at fig. 4.2 on page 15 it is also clear that there is a lot of redundancy in the temporal domain, which is unnecessary. In order to reduce this a video or image compression approach may be useful to reduce redundancy. Since the data is already in a compressed format that can be used natively, it may be good for performance to leave it in that format. It is also important to note that the transforms with trigonometric functions (DCT, wavelet, and so on) are computationally expensive. This leaves the predictive coding and Burrows-Wheeler transform. However, the Burrows-Wheeler transform is based on language models and requires structured data such as text. It is unlikely that this will be applicable to the lighting data, which leaves the predictive coding approach. However, as this is a performance-centred project it is important to not add parts unless they are required.

As mentioned in section 4.1.2 on page 14 the data will be used on a GPU, which has hardware accelerated interpolation and texture lookups. It will therefore be easy to interpolate between different frames in this context, and if frames can be interpolated within allowed error margins they can be removed altogether. The forward predictor or linear approximation methods mentioned in section 5.1.4 on page 26 can be used to cull some of the frames. As our goal is to interpolate however, the linear approximation method will be used. It may also be possible to use some intraframe segmentation technique to reduce the empty space that is stored, but at the cost of more expensive compression and decompression.

To recapitulate, the proposed compression algorithm will use a three-step algorithm with a **linear approximation**, one step with **LZ77** that can be expanded into one of its derivatives depending on the requirements, and **Huffman coding** to reduce entropy in the final source. This can be seen in fig. 7.1. There is also an unresearched optional step with image segmentation or similar between steps one and two that can be researched further if necessary.

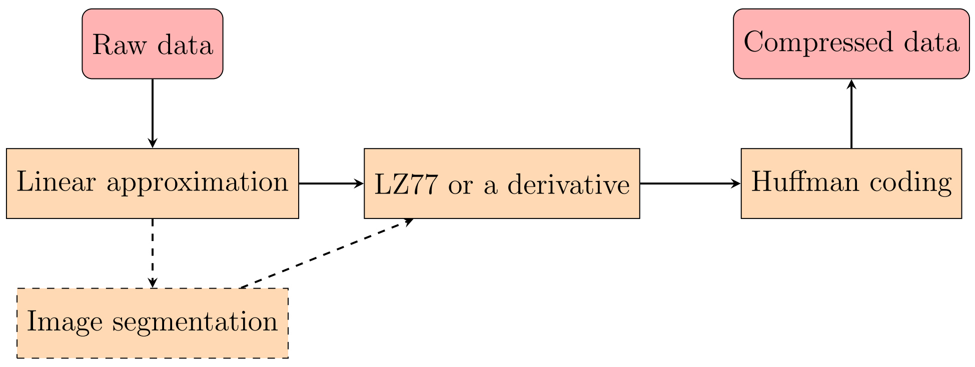

Figure 7.1: Original proposed compression pipeline

7.1.1 Existing implementations and relative gains

Based on the abundance of existing open source implementations of compression algorithms that are both well-documented and tested by many users, it was decided that it was better to use an available library instead of developing a new one. This will also allow more focus to be invested into the more advanced methods as opposed to repeating what others have already done. The proposal suggested a LZ77-type algorithm, and there are many available open source libraries for Python. The choice was made to use Python-LZ4 as it is well-documented, fast and requires only one line of code for both compression decompression. This makes it easy to replace with another library in the future.

It was also decided that the linear approximation method shall be a central part of the final compression pipeline, with the addition of image segmentation. The image segmentation part was deemed important as each atlas contains charts for many different objects. These objects may shade each other, and obviously have different facing. Therefore they behave differently, and by utilizing this fact when doing the approximation the errors will be lower if the image segmentation is well-aligned with the charts.

This segmentation approach will also make the predictive coding approach more effective, as the pixels in each segment should be similar to each other. It will therefore also be included to compress each segment.

7.2 Image segmentation algorithm and data structure selection

Starting with the choice of image segmentation algorithm, it is clear that neither of them meets the only explicit requirement of creating straight cuts. This is an implementation detail though and it is likely that most of the algorithms can be adapted for this. It may however not be possible to adapt *k-means* or *mean shift* as they work only in the feature-space, and hybridizing it to make straight cuts in the spatial domain may prove to be difficult if not impossible. Similarly the naïve thresholding approach while locally adaptive will likely also be hard to create cuts with, as well as being likely to behave badly around the edges due to the rapid shifts in intensity.

This narrows the choice to edge detection, as well as the graph-based choices. The edge detection approach combines well with the observations in section 4.3 on page 17 but edge detection filters suffer from discontinuity problems which requires reparation and merging to create a workable and straight edge. On the other hand, it is well aligned with the goal of reversing the atlas creation process, as this is the most likely area to have strong edges. Strong edges from an edge detection algorithm can also likely be found on the border to empty space.

The graph-based choices can also be seen as aligned with the goal, and in particular the min-cut max-flow algorithm. The merging approach though more sophisticated wants to create large areas, and this may not necessarily align well with the overall goals of reversing the atlasing.

A naïve distinction for the max-flow min-cut algorithm may be that everything empty is background, and everything non-empty is foreground. The explicit goal of the algorithm then becomes to cut along the longest border between empty and non-empty data. Combined with the above requirement, it can be further condensed to: *cut along the line or column with the most change between empty and non-empty space*. This can be done recursively or globally to segment the image. Defining it as a maximisation also makes it easy to combine with edge detection, as edges are strongest where the difference is high.

The choice of recursive or global segmentation is relevant for the choice of data-structure, or the reverse. However, an approach combining edge detection with a variation on the max-flow min-cut algorithm seems to be the most logical choice. Remembering the spikes and ridges in fig. 4.5 on page 20 and fig. 4.6 on page 20 it is clear that the approach may provide good results, but none of the spikes go all the way up. If a cut was made in these graphs and a recursive approach is used, the opposite graph would be reduced in height and increased in length[I]. As it may create a higher relative emptiness, it could increase the efficiency of the cuts.

[I] Consider that a cut in one graph is rotated 90°in the opposite graph. This means that cutting the horizontal graph will half the size, but double the length of the vertical graph if the two parts are visualised after each other

This leads towards a binary partitioning scheme, but this is not necessary. It could be possible to use the best column and best row as the centerpoint for a non-square quadtree. There is however one big drawback: consider the case where the best horizontal line is optimal on one side of the best vertical cut, but suboptimal on the other. Doing it in two steps would allow the best cut from both of them, and then the next cut could be made on each side and hopefully be better in total. The next decision then is between the binary tree and the k-d-tree.

The search characteristics of the trees is not very important for the purposes here, as the primary use for the tree is to aid the subdivision and for the data-structure. However, as the segmentation algorithm will partition in two dimensions, the idea fits more with the k-d-tree than the binary tree, and as such it will be used.

The temporal segmentation will then occur within each of the leafs, and a good data-structure needs to be chosen for this data. The data to be stored will consist of key-value pairs where the key will be the temporal index of the data, and the value will be the actual data. This lends itself to a hashmap, but hashmaps are not very memory efficient as each possible hash-result needs to have an associated slot in the map. As the temporal indices will be integers of low magnitude, it is likely that this will become problematic and introduce an unnecessary memory overhead. In the example data used there are 384 temporal segments originally, and even if the subset of used frames remains high, the relative overhead will quickly become large. A more reasonable solution would be to use either a parallel array approach, sorted by the key. As the keys will be used in sequence, the index can be stored and only increased when necessary.

The next step used will be the actual compression step, which will use the predictive coding from PNG, but substitute LZ4-compression for `DEFLATE` which is normally used in PNG. This is done per slice. If the final data size is too big the whole stack of slices can be compressed as well, at the cost of decompression time.

The last step is storing the actual data. There is no major reason not to choose either, but as the uncompressed data sets might still be quite large it would be logical to only load them as needed, which is easier to do in a header-based format.

To recapitulate, the algorithm shall consist of three steps as shown in fig. 7.2. The first step is the *image segmentation step*, which subdivides the spatial domain using a specialized max-flow min-cut algorithm. The output from this step is used to build a *k*-*d*-tree, to logically separate the data parts. Each leaf in the *k*-*d*-tree is then segmented in the temporal domain, using a *linear approximation* approach to find the important keyframes. Each of the keyframes is then first filtered with *predictive coding* before being compressed using *LZ4*. Lastly, all data is written to a file using a header-based file structure. There is also the optional step, shown with dashed lines, that can be used if needed but will come at a performance cost.

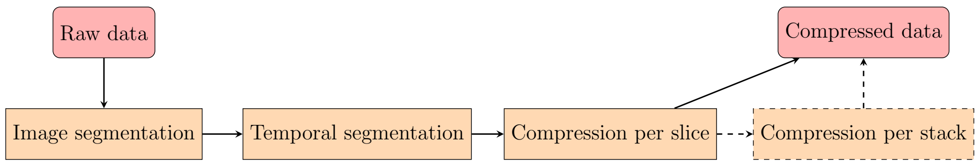

Figure 7.2: Updated proposed compression pipeline

It is also important to remember that the lightprobe data mentioned in section 4.2 on page 14 does not have a spatial representation. This also means that the spatial algorithms proposed here are not applicable. However, as the data-size is very small, this will not have a huge impact on the final file-size.

7.3 The full pipeline

The full algorithm that was implemented is shown in fig. 7.3, with annotations showing both step and algorithm. Though not shown, the decompression process is an exact reversal process, by first decompressing each slice individually, interpolating the temporal slices that are not stored, and then merging all segments to recreate the original texture frame.

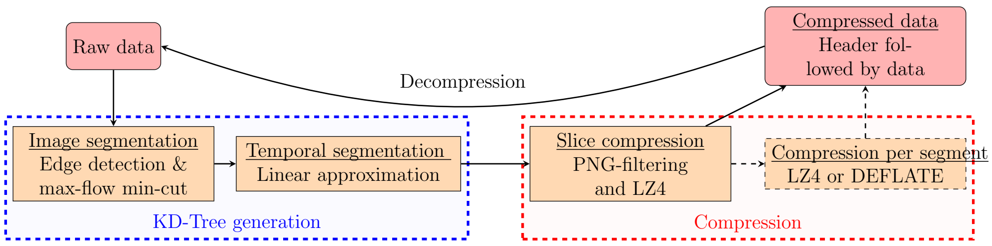

Figure 7.3: Final compression pipeline

7.4 K-D-tree

The k-d-tree suggested for usage is a modified version of the standard algorithm, as it is not generated using the normal rules of median-point-selection. It is instead recursively partitioned using the segmentation algorithms detailed below. This segmentation is done first for the spatial representation of the data, and then in the temporal domain.

7.4.1 Image segmentation

The image segmentation algorithm iterate over all rows and columns in the data-set, finding the max value of the heuristic function (7.2) on the next page, as shown in (7.1) on the facing page. There are two parts to the heuristics function. The first part is based on first-order edge detection, and the second part is naïve max-flow min-cut approach discussed in 7.2 on page 35. A cut is made along the dimension and at the position where the heuristics function is maximized. To limit the depth a lower threshold τ_s is also used and cuts are only made when the heuristic value is greater than this value.

Let:

C_R and C_C be the number of rows and columns
I_R and I_C be the sets of rows and columns

Then:

$$H_s(I)_{axis,index} = I_{\underset{D,x}{\arg\max}\{h_s(I_{D,x},\ I_{D,x+1})\ |\ x\ <\ C_D,\ D\ \in\ [R,C]\}} \tag{7.1}$$

$$h_s(I_{D,x},\ I_{D,x+1}) = \sum_{i \in I_{D,x},\ h_j \in I_{D,x+1}} \begin{cases} |i-j| & if \quad i \neq 0 \wedge j \neq 0 \\ 1 & if \quad i = 0 \oplus j = 0 \\ 0 & if \quad i = 0 \wedge j = 0 \end{cases} \tag{7.2}$$

After initial tests, it was noted that there were some issues with detecting borders where the majority of the points fall into the third case. Therefore, another part was added to the heuristics function, as shown in (7.3)- (7.4). The purpose of the function $e(\cdots)$ is to normalize the border length, so that lines that pass through mostly empty space are equally likely to be borders.

$$h_s(I_{D,x},\ I_{D,x+1}) = e(I_{D,x},\ I_{D,x+1})\ \cdot \sum \cdots \tag{7.3}$$

$$e(I_{D,x},\ I_{D,x+1}) = \frac{\texttt{card}(\{p\ |\ p\ \in\ (I_{D,x} + I_{D,x+1}),\ ||p|| = 0\})}{\texttt{card}(I_{D,x})} \tag{7.4}$$

Three further optimisations were added to improve the algorithm. The first optimisation was to limit the segmentation based on the size of segments, so that already small segments are not subdivided. A length of at least 20 pixels in both directions was deemed good enough. The second optimisation is to force cuts to be made a certain distance from the borders. This value was chosen to be 5 pixels. The third optimisation was to use the median cut-point, if multiple points have the same heuristics value. This is based on the assumption that the median point is most likely to give a centered cut. These measures were implemented as the algorithm generated a lot of very thin segments otherwise, as shown in fig. 7.4 on the next page, leading to very large trees.

The threshold for cutting was very important for generating good cuts. $\tau_s = 0.8$ provided a good trade-off between depth and accuracy of cuts. Running the final algorithm creates the tree shown in table 7.1 on the following page. The matching segmented image can be seen in fig. 7.5 on page 41.

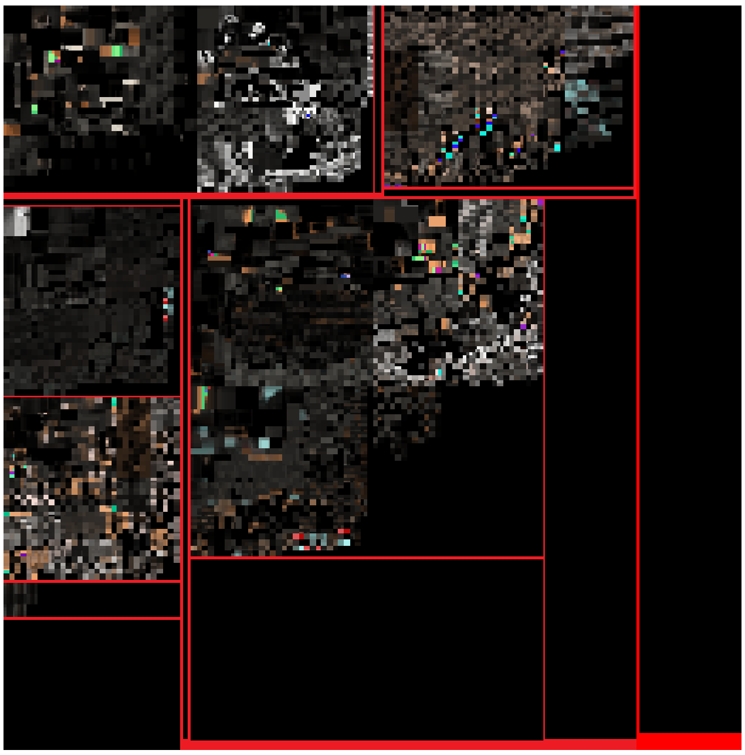

Figure 7.4: Initial segmentation without size-requirements.

```
Generated tree                   Position  Axis
   Root                               220  V
     1                                210  H
       2                               66  H
         3                            130  V
           Leaf
           Leaf
         3                             66  V
           4                          198  H
             5                        134  H
               Leaf
               Leaf
             Leaf
           4                          188  V
             5                        192  H
               Leaf
               Leaf
             Leaf
       Leaf
     Leaf
```

Table 7.1: Tree generated from spatial segmentation with nodes numbered by depth. Generated with threshold = 0.8.

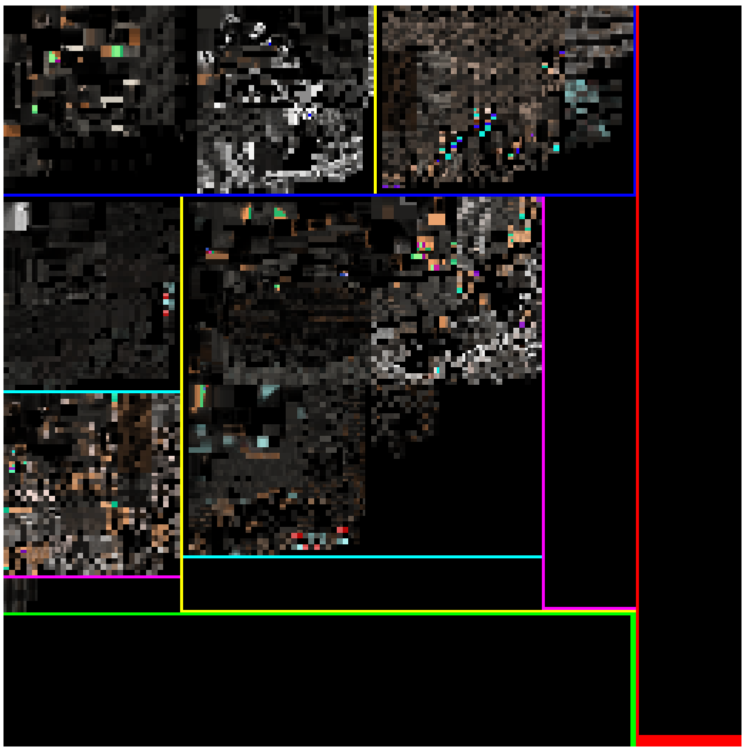

Figure 7.5: One frame of irradiance data matching the above tree, padded along the cuts to separate the segments so that all data is visible.

7.4.2 Temporal segmentation

The intended algorithm for use was the piecewise linear approximation by Hamann et al [42], however after further investigation it was deemed too computationally difficult to implement for this data. The primary difficulty was adjusting the algorithm for multiple data-sets changing in tandem, but possibly in different directions.

The algorithm was therefore simplified significantly to use a discrete integral of the root-mean-square-error as a measure of the change between two frames. It iterates over all frames while calculating the heuristics function (7.5) on the following page for the current frame and the next. When the integration value becomes greater than the threshold τ_t the previous frame is selected as a keyframe and the integral is reset to zero. This reversing behaviour means that the total number of comparisons is greater than the total number of frames.

Let:

C_F be the total number of frames

τ_t be the temporal segmentation threshold

Then:

$$H_t(f)_k = f_{\max\{i|h(i)\leq k\tau_t\}} \qquad \text{for } k \leq \left\lfloor \frac{h_t(C_F)}{\tau_t} \right\rfloor \tag{7.5}$$

$$h_t(k) = \sum_{i=0}^{k} \texttt{RMS}\left(f_{i+1} - f_i\right) \tag{7.6}$$

$$RMS(f) = \frac{1}{A_{f_i}} \sqrt{\sum_{p\ \in\ f} p^2} \tag{7.7}$$

The reason for this choice is that if f_k is at the beginning of a climb, and point f_l is at the peak, then it is more likely that the threshold is broken when comparing f_l to f_k than when comparing f_k to f_j. f_k is therefore used as a keyframe so the interpolation is done along the rise. This however comes at the drawback of the function possibly leaping over small valleys and peaks. An attempt at visualisation of this algorithm can be seen in fig. 7.6.

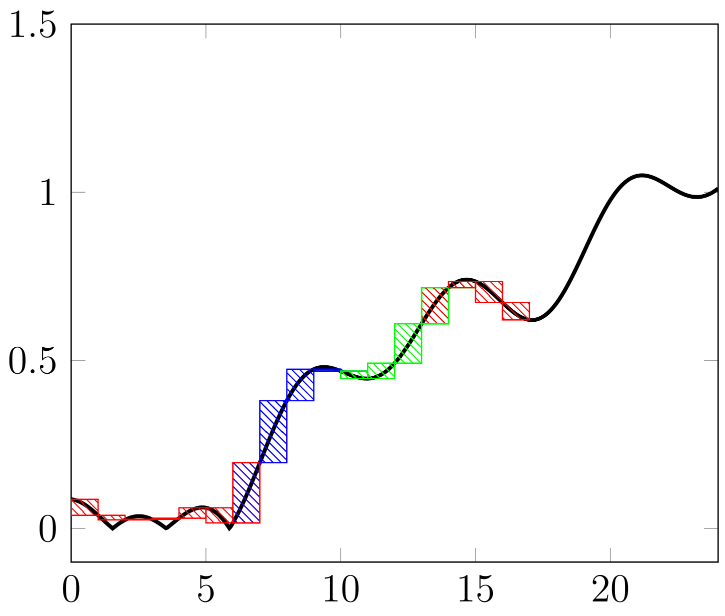

Figure 7.6: Example plot of algorithm with reversal. The colors indicate a segment of the algorithm, with reset of the integral when changing color. Not representative of real data.

However, the algorithm suffers from one big drawback when considering the circular nature of the data. The basic algorithm starts from frame 0 and iterates forward, which means that all spatial segments will start with frame 0 as a keyframe. However, there may be other solutions that both provide better interpolation and less keyframes. The algorithm was therefore extended to try all points in the range $[-30, 30)$ as start-points, instead of starting with 0. The best out of all these start points is then used for the actual tree-generation. If no more than 1 keyframe is needed, a random point is chosen in the interval.

As with the image segmentation, the threshold τ_t is important for the result. Mathematically, assuming that frame distribution is/will be uniform[I], the overall noise introduced by interpolation can be estimated using $\hat{e}$ shown in (7.8). This equation however has three unknown variables: τ_t, $\hat{e}$ and K. The equation can instead be rewritten to create eq. (7.9). This equation shows that the ratio between average noise and threshold is proportional to the fraction of keyframes over all frames and segments. Considering that this fraction should be similar to the fraction of frames left, it can be assumed that it is linearly inversely proportional with the target compression ratio, e.g. $1 - T$. This reduces the unknown variables by one. Finally, the equation can again be rewritten to (7.10) giving τ_t as a function of average noise. The reverse function (7.11) shows another obvious property: if the target compression is increased, then the noise must increase proportionally.

[I] We know this is false, it is one of the premises for this entire project. But let's assume it's true.

Let:

$\hat{e}$ be the average error
K be the number of keyframes
T be the target compression rate
S be the number of spatial segments
τ_t be the temporal segmentation threshold

Then:

$$\hat{e} = \frac{K}{C \cdot S} \times \tau_t \tag{7.8}$$

$$\frac{\hat{e}}{\tau_t} = \frac{K}{C \times S} \tag{7.9}$$

$$\tau_t(\hat{e}) = \frac{\hat{e}}{1 - T} \tag{7.10}$$

$$\hat{e}(\tau_t) = \tau_t \times (1 - T) \tag{7.11}$$

For example, if the average noise is allowed to be 0.002 then τ_t becomes $0.01\bar{3}$. Initial testing showed that this actually gave a much higher compression rate and noise, as the distribution is not uniform spatially or temporally. Reducing the threshold down to 0.01 improved the results, with a still higher compression rate than the target. One full run with $\tau_t = 0.01$ can be seen in fig. 7.7 on the following page, and a full visualisation with both spatial and temporal segmentation can be seen in fig. 7.8 on the next page. More information about the errors can be found in chapter 8 on page 49.

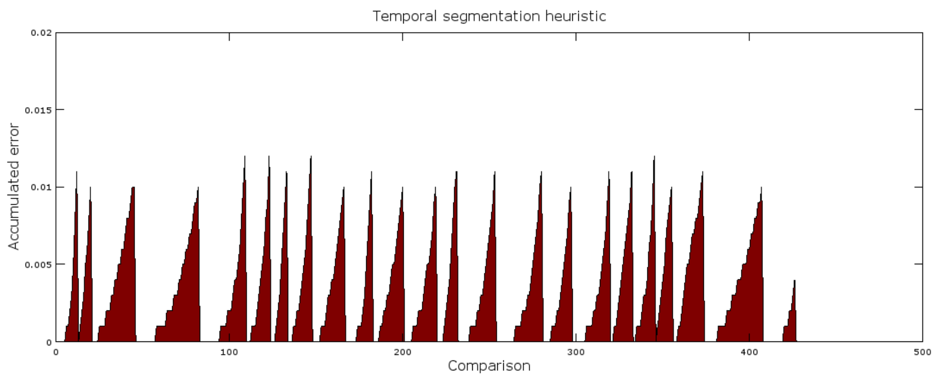

Figure 7.7: Graph of the max of all heuristic channels used for temporal segmentation, plotted against the comparison number.

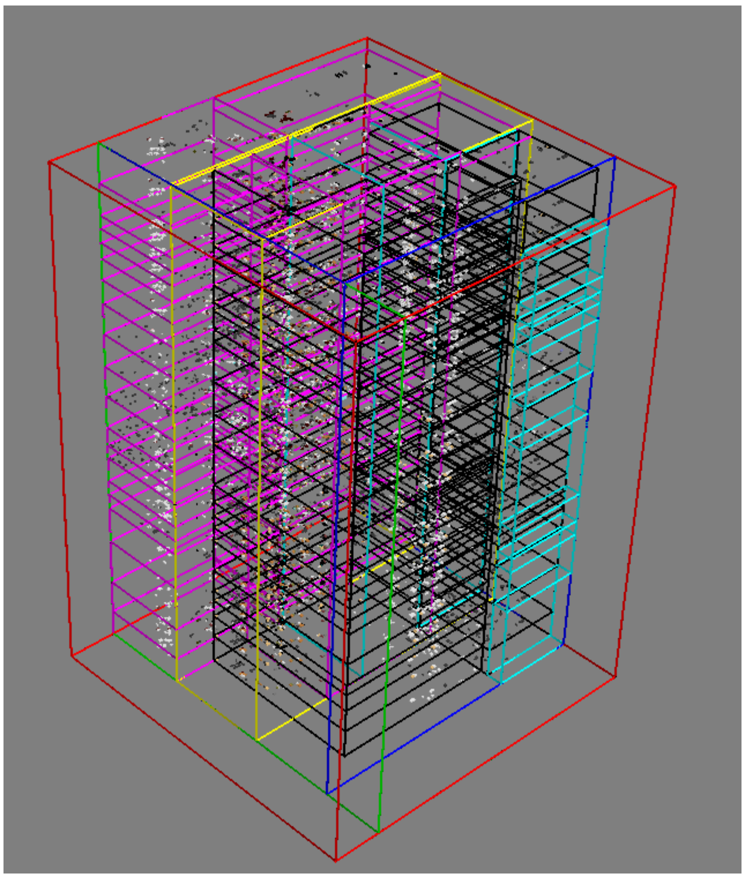

Figure 7.8: 3D-visulsation with bounding boxes for each interpolation system at the leaf-level, and with the same tree as above in the spatial domain. Note that the colors match: the first cut is made in the red domain, which is the root bounding box.

7.4.3 The heuristic function

The heuristics function shown above is the one used in the prototype, but other heuristic functions were also tested.

Three different functions based on first-order derivatives were tested. The one that that is used above is roughly equal to *the difference between the current frame and the next frame*, which is a first-order delta. However, *the difference between the previous keyframe and the next frame* was also tested, which allowed a higher threshold while creating a more uniform distribution as the integral scales with distance. The third heuristics function tested attempted to combine the two other approaches as *the difference between the previous keyframe and the next frame, divided by the distance between the frames.* The idea was to react slower to slow change, but maintain the reaction speeds of setup two. However, the results in both cases produced more and larger artifacts.

An attempt was also made to use a second-order delta, as the algorithm only needs to insert keypoints whenever there is a change of direction rather than a change of value. However, due to the increased computational complexity of a numerical second-order delta the algorithm was terminated after running for over one hour.

7.5 Compression

7.5.1 PNG

The PNG-standard consists of two compression parts. The first part is the filter, which is briefly mentioned in section 5.1.4 on page 26. The actual predictive coder used in PNG actually consists of five different coders, and the best one is chosen for each part of the data [55]. Of special note is the Paeth filter, which is an adaptive filter by itself [56]. The output from the filter is a token for the filter used, followed by the filtered data.

The PNG-filters encode based on bytes, but is designed to handle different formats by always operating on matching bytes from different parts. For example, if a normal RGB8-format is used[I], the red is always matched with red in the previous pixel. If a RGB16-format was used, the first red byte would be matched against the other first red byte. When referencing other bytes, it is assumed that the data is unfiltered or already defiltered. The reference system used can be seen in fig. 7.9 on the following page.

[I] one byte per channel

The filters are shown mathematically in eq. (7.12) on the next page-(7.17) on the following page, though some deviations were made from the standard. In canonical implementations the unavailable parts are

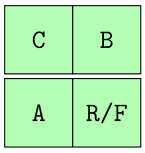

Figure 7.9: The references used in PNG-filter operations.

treated as 0, for example references to the previous line while working on the current line. In this implementation however, the filters that use unavailable data on the same row fall back to the first filter until data is available. Similarly, the filters that need a previously encoded row to function properly do not apply for the first row, in order to reduce computational complexity.

Let:

R be the original data
A be the previous match
B be the match in the previous row
C be the previous matching byte in the previous row
F_{1-5} be the filter functions
F_o be the filtered data

Then:

$$F_1 = R \tag{7.12}$$

$$F_2 = R - A \tag{7.13}$$

$$F_3 = R - B \tag{7.14}$$

$$F_4 = R - \left\lfloor \frac{A+B}{2} \right\rfloor \tag{7.15}$$

$$F_5 = R - \min_{|p-x|} |p - x| \qquad \begin{cases} x \in \{A, B, C\} \\ P = A + B - C \end{cases} \tag{7.16}$$

$$F_o = f_{\arg\min_x \{\sum(f_x)\}} \qquad for\ x\ \in\ [1, 5] \tag{7.17}$$

7.5.2 LZ4

The LZ4 compression used the open-source Python implementation of LZ4. The data from a filtered segment was compressed by the LZ4-library, before being written to the file.

7.6 File format

The file structure is header-based format as described in chapter 6 on page 27. The general structure and sizes of the header can be seen in fig. 7.2 on the facing page. In the original implementation, the data was appended to the header, but it was made more robust by moving the data to its own file.

Header	Type	Count	Total size
Spatial info	*per*	*spatial*	*segment*
└── Size	Short	2	4
└── Position	Short	2	4
└── Type:	Byte	1	1
└── └── Node (0)			
└── └── └── Axis	Byte	1	1
└── └── Leaf (1)			
└── └── └── No_ Keyframes	Short	1	2
└── └── └── No_ Frames	Short	1	2
└── └── └── Keyframe data		*per keyframe*	
└── └── └── └── Key	Short	1	2
└── └── └── └── Buffer	Long	1	4
offset	Long	1	4
└── └── └── └── Length	Long	4	16
└── └── └── └── Length of parts			

Table 7.2: The structure of the file header

7.7 Decompression of a frame

As noted in section 7.3 on page 38 the decompression works in reverse of the compression algorithm. Whenever a lighting system is loaded, the full header is parsed into memory. When using the data, there are two distinct functions that need to run at regular intervals. The first function updates the interpolation texture, which contains interpolation information for every pixel in the textures. The second function updates the textures that are interpolated between to calculate the actual texture value.

The interpolation texture is updated each time the function is called, no matter the time-point. The interpolation value is the fraction of time that has passed between the previous keyframe and the current keyframe, for the particular segment. This value is calculated using function (7.18). The interpolation value is adjusted using modulo logic to make sure the interpolation is seamless when passing midnight.

Let:

T be the current time
T_p be the previous keyframe time
T_n be the next keyframe time

Then:

$$\delta(T) = \frac{T - T_p}{T_n - T_p} \tag{7.18}$$

The other two textures are updated only updated when a segment has a keyframe at exactly the current timepoint. When this occurs, that keyframe and the next keyframe are loaded from the file, decompressed and added to their corresponding textures. A way of mentally visualising this is to see the two textures as the top and bottom of a box. The interpolation textures then form loading bars, that rise at different speeds for different parts of the texture. When a loading bar is full, that part of the textures need to be updated.

The LZ4 decompression is done using the same open source library that was used for compression, and the data is then passed to the defiltering function. The defiltering operation works exactly like the filtering operation, with sign changes only. This can be seen by comparing (7.19)-(7.23) to (7.12)- (7.16) on page 46. The choice of defiltering function is based on which token was included the stream.

Let:

F be the defiltered data
R_{1-5} be the defiltering functions

Then:

$$R_1 = F \tag{7.19}$$

$$R_2 = F + A \tag{7.20}$$

$$R_3 = F + B \tag{7.21}$$

$$R_4 = F + \left\lfloor \frac{A+B}{2} \right\rfloor \tag{7.22}$$

$$R_5 = F + \min_{|p-x|} |p - x| \qquad \begin{cases} x \in \{A, B, C\} \\ P = A + B - C \end{cases} \tag{7.23}$$

$$\tag{7.24}$$

Chapter 8

Results

This chapter describes prototype, the compression ratios achieved and various error measurements.

8.1 The prototype

A standalone prototype was created implementing the full algorithm, as well as tools for visualising the algorithm in various ways. It has been tested on three different data-sets, and the results have been visually and quantitatively measured. As this is a performance improvement special care was taken to optimise the prototype as well as possible, to get good measurements of execution speed.

8.1.1 Execution speed

The compression time is proportional to the size of the data to compress, and tests showed that increasing texture size by a factor 4 increased compression times by roughly a factor 2. The decompression times were so fast that it was not possible to measure accurately using Python, but the time figures were always less than one hundredth of a second. The largest bottleneck was reading from the hard drive, which could cause noticeable hitches. However, this is a task that can be done asynchronously, and should not be an issue in a real implementation. Recalling the loading bar analogy from section 7.7 on page 47, it is possible to use it as actual loading time to preload the upcoming keyframes.

Though the performance gain by going from Python to C++ is hard estimate, some extrapolation can be done from other benchmarks. Based on various benchmarks a reliable performance gain going from Python 3 to C++ is at least one order of magnitude [57, 58]. However, the prototype was written in Python 2 to be able to access visualisation tools that are not ported to Python 3 yet. There are few benchmarks available for this conversion, but the average performance of Python 3 is equal to Python 2 with varying case-to-case performance [59]. It is therefore likely that a C++ implementation reliably will run below $1ms$ per frame.

8.1.2 Compression efficiency

The efficiency of the compression was measured simply as the size reduction, which is the ratio between compressed and uncompressed sizes as shown in (8.1). As noted several times before, the size of the lightprobe data is an order of magnitude smaller than the other data, and is not included in the compression measurements. As shown in 8.1.2 the are very high for all three datasets.

$$R_c = \frac{Compressed\ size}{Uncompressed\ size} \tag{8.1}$$

Data-set	Original size	Compressed size	Efficiency	Lightprobe size
1	479.1 MB	7.4 MB	98.45 %	(+7 MB)
2	1873 MB	22.3 MB	98.77 %	(+147 MB)
3	480.0 MB	5.0 MB	98.96 %	(+41.7 MB)

Table 8.1: The compression efficiency for three data-sets. The original and compressed sizes both exclude the lightprobe data, shown in a separate column.

8.2 Visual and quantitative errors

The data was compared to the original data using two different three different measures. The simplest measurement was a purely visual inspection by constructing a difference image between the interpolated image and the original data. This difference image was animated over the full time-span, and checked for points with high-intensity. Mostly, the difference was so small it was not visible, but a few errors were seen where a few pixels in an area would deviate large from the area mean, being smoothed out by the local mean. Two examples of the comparison can be seen in fig. 8.1 on the facing page.

In order to quantitatively measure the errors, the *Root-Mean-Square-Error* (RMSE) and *Signal-To-Noise ratio* (SNR) measurements were used. The equations for these can be seen in (8.2)-(8.3) on the next page. The RMSE measurement is almost equal to the average error, but as the sum is before the root, it has a weighing feature which gives greater importance to big differences than small.

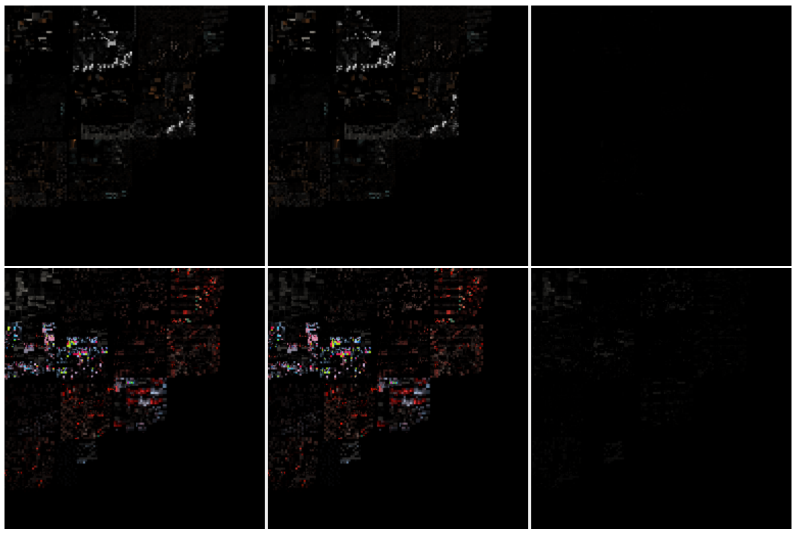

Figure 8.1: Original, interpolated and difference image at $t = 105$ and $t = 132$, respectively. Dataset 2 was too large to visualize.

SNR instead measures a relative strength of the original signal compared to the noise signal [60]. As the name implies, a large result implies that the signal is much stronger than the noise, while a small results implies that the noise is large relative to the original image, i e there is a large distortion.

These two measurements are visualised in fig. 8.2 on page 53. Due to the difference in data-types it was not possible to visualise all the data parts together. As the irradiance data is the most notable part, it was chosen for the visualisation.

Let:

O be the original image
I be the interpolated image

Then:

$$\texttt{RMSE} = \sqrt{\frac{1}{\texttt{card}(pixels)} \sum_{p \in pixels} (O_p - I_p)^2} \tag{8.2}$$

$$\texttt{SNR}_{\texttt{DB}} = 10 \log_{10} \frac{|I|^2}{|I - O|^2} \tag{8.3}$$

Though the best measure of actual quality would be looking at an actual rendering, we can draw some conclusions from the graphs in fig. 8.2 on page 53. For the RMSE-measurement, the original data exists mostly in the range $[0, 1]$. Compared to this, the error ranges in the first two sets are very low. The third one however has a very spiked pattern, and it has several visible defects in the data. Upon inspection, the spatial subdivision did not create a clear enough cut, and instead cut the whole upper left quadrant as a single segment as can be seen in 8.3 on page 54.

The SNR-graphs show that the interpolation for the second data-set has very little noise, shown by both the large SNR-value as well as the stable plot. It is notable that this graph also has the most concentrated RMSE-values. This data was too large to visualize, but inspection of the generated tree showed that the keyframes had a globally uniform distribution, as well as segments of similar size. The other two graphs have much lower SNR-values, and more dynamic RMSE-values.

It is likely that the quantitative errors are very dependent upon the quality of the spatial and temporal segmentations as the most well-formed tree also had the best error measurements. It follows from this that the parameters of the algorithm need to be tuned for each data-set to achieve a good segmentation in both dimensions.

One way of achieving this could be by using an adaptive tuning algorithm, which uses the two metrics above to decide whether a generated tree is good enough. However, this falls outside the scope of this report. While the parameters could technically be set manually for each data-set, the primary users will not be programmers. Though this may not necessarily be a bad thing in itself, it requires a lot of implementation details to be expressed in non-technical terms to allow tuning by other means than trial-and-error.

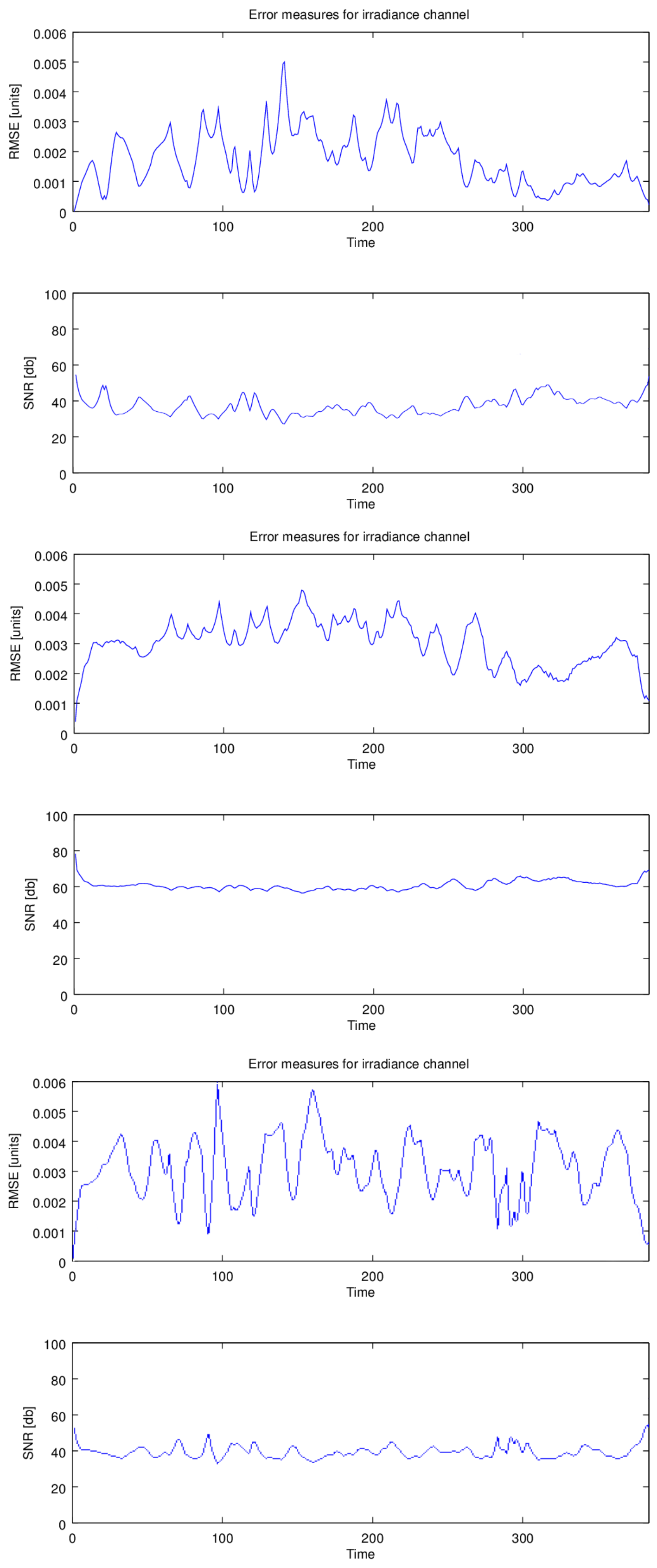

Figure 8.2: Measured error as RMSE and SNR for the three data-sets above.

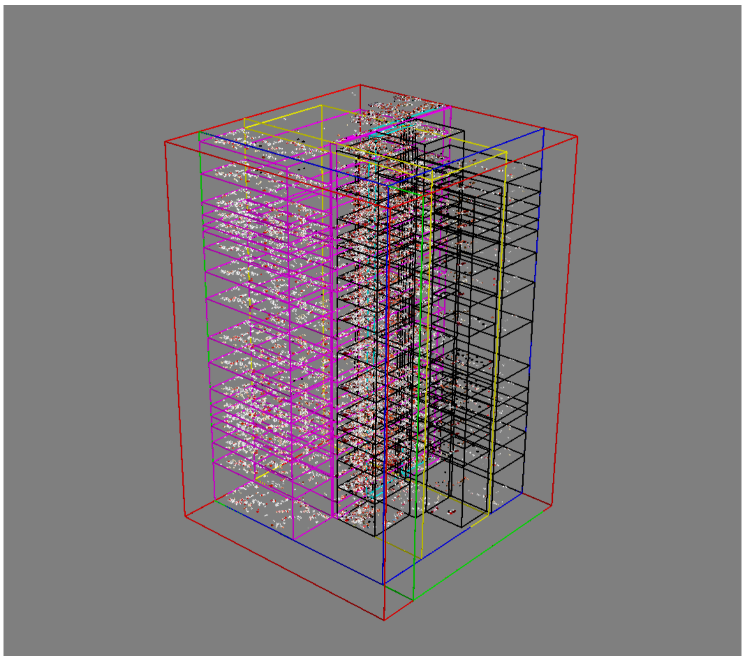

Figure 8.3: Full segmentation of data-set three, similar to the tree shown in chapter 6 on page 27

Chapter 9

Discussion

9.1 Compliance with the project requirements

The results are very satisfying, both in relation to the original goal, but also in relation to the goals that have been set as the project progressed. Though jokingly talking about 99 % compression as a goal during the project, it was almost achieved in the end. This is far above what conventional compression methods achieve, which was the original point of comparison. There is a point to be made about the lightprobes not being included. However, the same approach minus the segmentation should be usable, and achieve almost as good compression efficiency.

There is no doubt that the data can be compressed effectively enough using the methods proposed in this report, and there is still a lot of room for improvement that could not fit within the time or scope of the thesis. One part that could cause issues is the inclusion of scripted lights, which could not be used due to technical limitations. Including them will introduce new and interesting challenges, and may need the original data to be generated at very high temporal resolution to accurately represent lights turning on or off.

As shortly noted at the end of chapter 8 on page 49 there is also a concern of allowing not only dynamics inside each data-set, but also letting each data-set achieve its optimal compression. This can then be tied into the general workflow of the tool. Though most of it was deemed to be too specific to include in this report, there are several ways of generating the original data, and it needs to be possible to switch these without modifying the compression pipeline. The intended usage is that the tool shall run directly after data generation, but if tuning is required it quite likely needs to be decoupled from the baking step.

9.2 Special results and conclusions

As noted above the achieved compression ratio is far above what was expected based on what conventional tools achieve. Having the ability to achieve these high compression ratios allows a great deal of tuning to be done in either direction. There is lots of room for both increasing the thresholds used, depending on the importance of the data.

9.3 Project development

There are many different directions to expand both the primary subject and the secondary subjects that were touched upon in this project. The first part would be looking at more heuristics function inside an actual implementation - it may be possible to present the end-user with a few different heuristic functions that have different characteristics, or implement an adaptive approach. There are also many opportunities for improving the compression algorithm and finding more effective ways of representing data, and this is an area I would like to explore in the future. But this approach should also be possible to apply for more areas in software development, and generalizing the concept may prove useful.

Another part that could be improved upon is to explore how large the degradation can be before it is noticed. As this project purely used indirect lighting data, it might be possible to degrade it a lot before anything is noticed, and this is also an area I would like to explore. There is research comparing different shading models, but not degradation of the same model. Consider for example the graphic settings in a game - they are optimal for performance but it is entirely possible that it is not the most cost-effective settings, in relation to consumer enjoyment.

9.4 Reflection on own learning

The biggest takeaway on my own knowledge from this project is that for this project, I had no knowledge. The concept of data-oriented design, which is very important for modern games development was essentially unknown. Equally unknown as the implementation of compression algorithms, as well as both the mathematical and implementation background for spherical harmonics. Similarly I had never had the need to implement according to a specification. As such, a large amount of time of this project has been spent reading, in the region of 200-250 articles and papers. This is something that has not been a large part of my education, as there have been only two courses in the last three years where it has been relevant or required.

Though the number of 200-250 is not a good measure of learning, it perhaps highlights a lack of experience in finding good scientific material. Though there is room for self-criticism in this, I also felt that I did not have tools to find the information I needed. Throughout my three years at university I can remember only once that there has been any information about actual information search, which was during this thesis project.

Similarly, I feel that I lack the scientific methodologies I would have needed here. I originally planned to use a literature review and a decision-matrix approach for information review and choosing a comp algorithm. When decision matrix was not applicable I did not have any more tools to apply for this situation. While research about tools can be seen as part of the course, Rome was not built in a day. Having at least heard about different methods would have been useful.

There is a disconnect between I feel what has been expected from me from **DICE** compared to Örebro University. Indeed, the productivity throughout this project has been hindered by trying to fulfil them both while also not doing (too much) overtime. It has however been a great relief to have four different persons - three at DICE and one at ORU - that push and pull in various directions, and it forces a goal-oriented mindset. The biggest difference is that the university does not seem to care what is done as long as long as a results chapter is produced. On the opposite end, when working in a scrum-team there is a certain pressure to make progress, but no definite goal. Though a bit exaggerated, this is an observed difference that for me was very startling at first but showed a reasonable middle road in the end. The university mindset creates a habit of obsessing over every detail to create a perfect result, which is alleviated by always having to progress. On the other hand, it also prevents too large deviations from the plan, and keeps everything moving in the same direction.

Bibliography

[1] Simon Niedenthal. *Complicated shadows: the aesthetic significance of simulated illumination in digital games.* Department of Interaction and System Design, Blekinge Institute of Technology, Karlskrona, 2007.

[2] James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[3] David S Immel, Michael F Cohen, and Donald P Greenberg. A radiosity method for non-diffuse environments. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 133–142. ACM, 1986.

[4] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[5] James F Blinn. Models of light reflection for computer synthesized pictures. In *ACM SIGGRAPH Computer Graphics*, volume 11, pages 192–198. ACM, 1977.

[6] Peter-Pike Sloan. Stupid spherical harmonics (SH) tricks. Companion Paper, 2008. Game Developers Conference 2008.

[7] Jason Mitchell, Gary McTaggart, and Chris Green. Shading in valve's source engine. In *ACM SIGGRAPH 2006 Courses*, pages 129–142. ACM, 2006.

[8] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 497–500. ACM, 2001.

[9] Robin Green. *Spherical Harmonic Lighting: The Gritty Details.* Sony Computer Entertainment America, 1 edition, 2003.

[10] Sebastian Deorowicz. *Universal lossless data compression algorithms.* PhD thesis, Silesian University of Technology, 2003.

[11] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[12] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 23(3):337–343, 1977.

[13] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 24(5):530–536, 1978.

[14] Guy E Blelloch. *Introduction to data compression.* Computer Science Department, Carnegie Mellon University, 2001.

[15] Andrey Iones, Anton Krupkin, Mateu Sbert, and Sergey Zhukov. Fast, realistic lighting for video games. *IEEE computer graphics and applications*, 23(3):54–64, 2003.

[16] Meilin Yang, Ye He, Fengqing Zhu, et al. Video coding: Death is not near. In *ELMAR, 2011 Proceedings*, pages 85–88, Sept 2011.

[17] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.

[18] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, June 1987.

[19] Amir Said. Comparative Analysis of Arithmetic Coding Computational Complexity. In *Data Compression Conference*, page 562, 2004.

[20] Jarek Duda. Asymmetric numeral systems. *CoRR*, abs/0902.0271, 2009.

[21] Jarek Duda. Asymmetric numeral systems as close to capacity low state entropy coders. *CoRR*, abs/1311.2540, 2013.

[22] Charles Bloom. Understanding ANS. `http://cbloomrants.blogspot.fr/2014/01/1-30-14-understanding-ans-1.html`, 2014. Series of blog posts, 13 parts. Last accessed April 23, 2015.

[23] Fabian Giesen. rANS notes. `https://fgiesen.wordpress.com/2014/02/02/rans-notes/`, 2014. Blog post. Last accessed April 23, 2015.

[24] Google Codec Developers. New entropy coding: faster than Huffman, compression rate like arithmetic. `https://groups.google.com/a/webmproject.org/forum/#\protect\kern-.1667em\relaxtopic/codec-devel/idezdUoV1yY`, 2014. Forum discussion. Last accesed April 24, 2015.

[25] Yann Collet. Finite State Entropy. (December 16), 2013. Blog post. Last accessed April 23, 2015.

[26] Matt Mahoney. Data Compression Programs. http://mattmahoney.net/dc/. Subheading: FPAQ. Last accessed April 23, 2015.

[27] Igor Pavloc. LZMA specification. http://www.7-zip.org/sdk.html. Last accessed April 22, 2015.

[28] Matt Mahoney. 10 GB compression benchmark. http://mattmahoney.net/dc/10gb.html, 2015. Last accessed April 22, 2015.

[29] Maximum Compression. Summary of the multiple file compression benchmark tests. www.maximumcompression.com/data/summary_mf.php. Last accessed April 22, 2015.

[30] Charles Bloom. Lzp: A new data compression algorithm. In *Data Compression Conference*, pages 425–425. IEEE Computer Society, 1996.

[31] Matt Mahoney. Data compression programs. http://mattmahoney.net/dc/, 2015. Last accessed April 22, 2015.

[32] Yann Collet. Lz4: Extremely fast compression algorithm, 2013. Documentation page. https://code.google.com/p/lz4/. Last accessed April 23, 2014.

[33] Yann Collet. Lz4 explained. Blog, may 2011. http://fastcompression.blogspot.se/2011/05/lz4-explained.html. Last accessed April 25, 2015.

[34] Craig G Nevill-Manning and Ian H Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2 and 3):103–116, 1997.

[35] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, HP Software Research Center, 1994.

[36] Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012.

[37] S. Golomb. Run-length encodings (corresp.). *Information Theory, IEEE Transactions on*, 12(3):399–401, Jul 1966.

[38] Andrew B Watson. Image compression using the discrete cosine transform. *Mathematica journal*, 4(1):81, 1994.

[39] Ronald A DeVore, Björn Jawerth, and Bradley J Lucier. Image compression through wavelet transform coding. *Information Theory, IEEE Transactions on*, 38(2):719–746, 1992.

[40] JB O'neal. Predictive quantizing systems (differential pulse code modulation) for the transmission of television signals. *Bell System Technical Journal*, 45(5):689–721, 1966.

[41] P. Elias. Predictive coding–i. *Information Theory, IRE Transactions on*, 1(1):16–24, March 1955.

[42] Bernd Hamann and Jiann-Liang Chen. Data point selection for piecewise linear curve approximation. *Computer Aided Geometric Design*, 11(3):289–301, 1994.

[43] Mark W Jones. Distance field compression. *Journal of WSCG*, 12(1-3):199–206, 2004.

[44] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: theory and its application to image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(11):1101–1113, Nov 1993.

[45] P.F. Felzenszwalb and D.P. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2):167–181, 2004.

[46] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[47] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):603–619, 2002.

[48] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, Nov 1986.

[49] Shimon D Yanowitz and Alfred M Bruckstein. A new method for image segmentation. In *Pattern Recognition, 1988., 9th International Conference on*, pages 270–275. IEEE, 1988.

[50] Digital Illusions CE. Introduction to data-oriented programming. Presentation slides. `http://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf`. Last accessed May 20, 2015.

[51] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[52] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[53] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[54] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[55] Mark Adler, Boutell Thomas, John Bowler, et al. Portable Network Graphics (PNG) Specification. Specification V1.2, W3C, 2003.

[56] Alan W. Paeth. II.9 - IMAGE FILE COMPRESSION MADE EASY. In JAMES ARVO, editor, *Graphics Gems II*, pages 93 – 100. Morgan Kaufmann, San Diego, 1991.

[57] Debian Project. The Computer Language Benchmarks Game. `http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=python3&lang2=gpp`, June 2015. Last accessed June 2, 2015.

[58] JuliaLang.org. Julia Benchmarks. `http://julialang.org/benchmarks/`. Last accessed June 2, 2015.

[59] Brett Cannon. Python 3.3: Trust Me, It's Better Than Python 2.7. In *PyCon 2013*, 2013. Presentation slides. `http://speakerdeck.com/pyconslides/python-3-dot-3-trust-me-its-better-than-python-2-dot-7-by-dr-brett-cannon`. Last accessed June 2, 2015.

[60] Philippe Hanhart, Marco Bernardo, Pavel Korshunov, et al. HDR image compression: a new challenge for objective quality metrics. In *6th International Workshop on Quality of Multimedia Experience (QoMEX)*, number EPFL-CONF-200191, 2014.