TU WIEN Informatics

Rendering of Point Clouds via WebGPU

Performance comparisons between WebGPU and common desktop methods on the subject of rendering large Point Clouds

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

David Bauer
Registration Number 012120495

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. Dipl.-Ing. Dr.techn. BSc Markus Schütz

Vienna, January 1, 2001

David Bauer	Michael Wimmer

Erklärung zur Verfassung der Arbeit

David Bauer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. Jänner 2001

David Bauer

Danksagung

Acknowledgements

Kurzfassung

Diese Arbeit beschäftigt sich mit dem Rendern von Punktwolken im Browser mithilfe der Technologie WebGPU. Der Hauptgrund für die Verwendung dieser neuen Technologie statt dem etablierten WebGL, ist die Einsatzmöglichkeit von Compute-Shadern anstelle einer üblichen Rendering-Pipeline. Diese sind für große Datensätze wie Punktwolken besser geeignet und machen es möglich, mit ihrer Hilfe, eine Leistungssteigerung zu gewöhnlichen Rendering Pipelines heraus zu arbeiten.

Das Ziel dieser Arbeit ist es, einen WebGPU Compute-Shader zu konstruieren, um in einer Website im Browser in Echtzeit große Punktwolken zu rendern. Es wird eine Rendering Pipeline gebaut, die anhand von mehrerer Compute-Shader die Punkte verarbeitet und in einen Framebuffer schreibt. Dieser Framebuffer wird gelesen im Browser angezeigt. In dieser Arbeit wird ebenfalls die Leistungscharakteristik von WebGPU Compute-Shadern untersucht. Mehrere System-Konfigurationen werden herangezogen um Schlüsse über die effiziente Verwendung dieser GPU API zusammen mit JavaScript im Browser ziehen zu können.

Abstract

This thesis explores rendering of point clouds in a browser with WebGPU. The main argument for the new and unestablished technology WebGPU is the potential of using compute shaders in a rendering pipeline. The current web solution, WebGL, does not offer this feature. Compute shaders are well suited for large amounts of arbitrary data. In related work, a noticeable speed-increase can be measured when employing compute shaders instead of normal rendering pipelines.

The goal of this work is to use WebGPU compute shaders in a website to render large point clouds in real time. A rendering pipeline is built that employs multiple compute shaders to transform the points and write their projection into a framebuffer. This can then be read and displayed on the website. Additionally, this work examines the performance characteristics of WebGPU compute shaders. Multiple different system configurations are tested to draw conclusions about efficient use of this GPU API in the browser in tandem with JavaScript.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Method 2
1.2 Contributions 3
1.3 Structure of this Work 3

2 Background **5**
2.1 Compute Shaders 5
2.2 WebGPU 6
2.3 LAS 7

3 Related Work **9**
3.1 Point Cloud Rendering 9
3.2 Compute Shaders 10

4 Structure of the Rendering Pipeline **11**
4.1 Batches and Batch-wise Culling 12
4.2 Depth Pass 13
4.3 Colour Accumulation Pass 13
4.4 Display Pass 14

5 Results **15**

6 Conclusion **21**

Overview of Generative AI Tools Used **23**

Übersicht verwendeter Hilfsmittel **25**

List of Figures **27**

List of Tables **29**

List of Algorithms **31**

Bibliography **33**

CHAPTER 1

Introduction

Point clouds are a vital part of capturing a real world 3D environment or object. Whenever a laser scan is performed, the created model will initially be some form of point cloud. Point clouds are used in various industries. Object scans can be used for asset creation in video games or movies. [LYC+23] Construction companies can use scans of construction sites for quality controlling their work [WSS+15] They can also use point clouds of bridges, roads and tunnels to check for cracks [FLL+22], deformations or other anomalies. An example of this can be seen in image 1.1. Various governments routinely scan entire swaths of land for use in height and terrain maps.

As is evident by the previous examples, point clouds tend to be used by agencies or larger

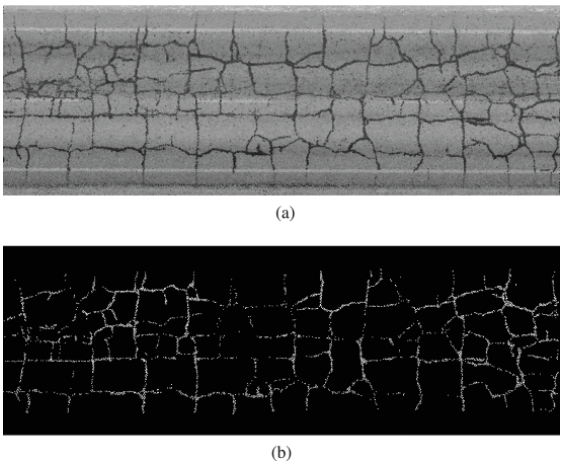

Figure 1.1: Crack detection of a 30m long point cloud scene. (a) Reflection intensity of the point cloud. (b) Detection result of the proposed method. Image taken from Luo et al. [FLL+22].

corporations. These point clouds are often gigabytes in size and hold multiple millions if not billions of points. The required throughput of data and fidelity of the result is often a key concern.

Handling large point clouds is made possible by taking advantage of the computing capabilities of modern graphics hardware. With their increased multiprocessing performance GPUs are a great option for speeding up necessary calculations. One such process for which the use of GPUs is advisable is the aim of this work: to take point clouds and to recreate them visually for manual inspection or showcase.

One technique for rendering point clouds is to view the data samples as vertices and to draw them using a simple vertex shader. The problem with this approach is that vertices are normally used for meshes and are therefore optimised for this use-case. Vertex shader outputs are often limited to transformed vertex positions, normals, and other per-vertex attributes. For large point clouds, this can become a bottleneck because the amount of data that needs to be processed and passed through the shader pipeline can be enormous, leading to performance issues.

Handling most of the calculations using the much more flexible compute shaders instead of just relying on vertex shaders offers up considerable improvements. The method employed in this thesis will be based on simply rendering the points via a projection as is common for rendering via vertex and fragment shaders. Some optimisations will be explained and applied.

1.1 Method

In this thesis, a renderer is built using the TypeScript superset of the JavaScript programming language and the Graphics API WebGPU. The goal is to build a renderer that relies on compute shaders to handle the highly parallelised computational work that comes along with large datasets such as point clouds.

This renderer implements level-of-precision (LOP) optimisations to improve memory fetching and use during rendering. This approach was first proposed in a work by Meyer et. al. [MSGS11]. The aim for this paper is that the more efficient memory use ultimately might also affect performance. We utilize GPU buffers to write data in our own designated format in order to reduce the amount of accessed memory needed for rendering points at lower LOP levels.

The language TypeScript was chosen because this project will be hosted as a website. JavaScript is dominant in this field. TypeScript, which is a superset of JavaScript enforces types, which is very helpful when working with lower-level concepts such as buffers and hardware-access.

The WebGPU graphics API is used in order to employ compute shaders. Because this thesis constructs a website, WebGPU is also one of few possible choices because the browsers themselves need to implement functionality. WebGPU is a fairly new graphics

API so analysing it's performance in tandem with JavaScript might provide insight for future development efforts.

1.2 Contributions

This thesis will show how to design a performant compute shader for working with point cloud or other similarly structured data when working with WebGPU and TypeScript. A rendering technique will be highlighted which utilises multiple buffers for faster memory access.

1.3 Structure of this Work

Chapter 2 will be about the theory of computer shaders and explaining how to use them to render an image. Some advantages and limitations will be discussed. In chapter 3 we show related work such as the optimisation strategy used in this thesis. Chapter 4 presents the rendering engine and accompanying website. The necessary steps to produce an image from point cloud data will be shown. The results as well as the performance characteristics will be handled in chapter 5 and, finally, in chapter 6 will hold the conclusion of this work.

CHAPTER 2

Background

This chapter serves to give an overview of the theory behind Compute Shaders and WebGPU. Some limitations will also be discussed. Furthermore, the LAS standard will be briefly introduced.

2.1 Compute Shaders

To render an image via the GPU, graphics pipelines or rendering pipelines are used. These serve as a framework that outlines the necessary procedures for creating the image. A lot of pipeline steps are implemented in hardware directly which one one hand makes them very fast but on the other hand means there can be large differences between GPU vendors such as Nvidia, AMD and Intel. For this reason graphics APIs such as OpenGL, Direct3D and Vulkan have been developed to create an abstraction over the hardware and help standardise code.

There are currently three major ways to render an image via the GPU. These are rasterisation, ray-tracing and compute shaders. Large parts of these are equal and interchangeable and there is also the possibility of constructing a hybrid renderer.

Rasterisation and Ray-Tracing is built on the idea of triangle meshes, however, since this work only uses points, these methods have unnecessary overhead. Compute shaders on the contrary are much less integrated into a pipeline. How they are used is largely up to the developer. They do not have a well-defined set of input or output values, these have to be specified and created as buffers manually. Furthermore the number of compute shader invocations [khr19] is also up to the specific use-case.

The invocations of a compute shader can be thought of similar to multi-threading on the CPU. However, they have multiple layers of hierarchy. The smallest unit is a thread, this corresponds to a single running instance of code. Threads are grouped into subgroups (also known as warps on NVIDIA and wavefronts on AMD), which execute in lockstep on

the GPU hardware. These subgroups are further organized into workgroups, which are defined by the shader and can synchronize and share memory. While WebGPU exposes workgroups explicitly, access to subgroups is limited and must be supported by the device [W3C25]. Workgroups are quite abstract as they can be defined to have one, two or even three dimensions. Workgroups themselves can also be created in a three dimensional space. This was done to make compute shaders adaptable to use-cases like image-transformation. In this example, work groups can be spawned two dimensionally and one thread can be spawned per pixel of the image. Similarly we could have one workgroup that spawns multiple threads itself however, the amount of threads a workgroup can hold is much smaller than the total number of workgroups possible. The WebGPU standard enforces a minimum of 256 x 256 x 64 workgroups with a total of at least 65535 workgroups to be handled by the device. However, only a maximum of 256 invocations per workgroup are required by the standard [Web25].

Threads within a subgroup can run in lockstep, this means that an operation is performed on all threads in the same cycle. As WebGPU only organizes in workgroups and not subgroups, how many threads per workgroup can benefit from this is up to the hardware and driver itself. Workgroup sizes of 64 threads are a recommended baseline for WebGPU, working well across multiple devices [weba]. Benchmarking is required to find the optimal workgroup size for specific compute shaders on specific hardware.

2.2 WebGPU

WebGPU is a GPU API with bindings for JavaScript, C, C++ and Rust that was designed for portability. It can be used in stand alone applications and imbedded in websites. In the web context it is the successor to WebGL. It is currently being developed by a W3C community group by Software-developers from Apple, Mozilla, Microsoft, Google and others. The goal is a more portable, faster and up to date graphics API. The predecessor, WebGL, was based on OpenGL ES which is based on OpenGL and therefore has an almost identical design to the desktop API. WebGPU on the other hand has no real connection to any existing graphics interface. Still there are conceptual similarities to Vulkan, Metal and Direct3D 12.

The figure 2.1 below shows a simplified diagram of how to set up WebGPU to draw a triangles via a vertex and fragment shader. The process reminds of the Vulkan API, where buffers, bind groups, and the pipeline among others need to be created by the program explicitly before being used.

To execute shaders these resources need to be set up. The process however, is not as complicated as is the case with Vulkan. Most of these resources can not be changed after they have been created. A buffer will remain the same size during its entire use-cycle. If, for example, a bigger buffer is needed, the old one needs to be discarded and a new one created.

Command buffers are used to control the state of the WebGPU runtime. When executing

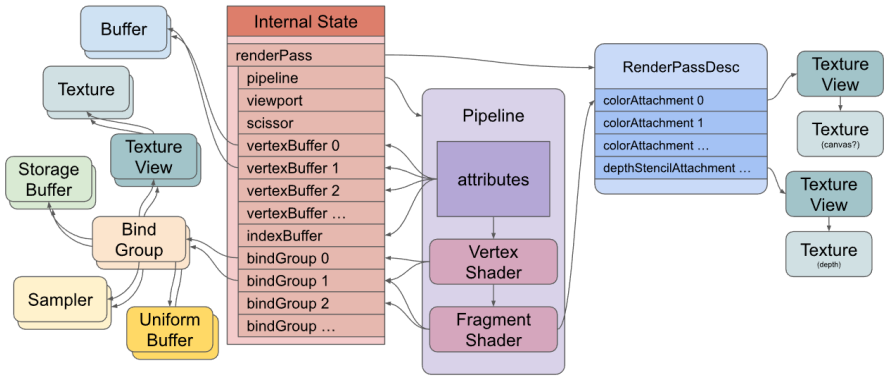

Figure 2.1: Simplified diagram of WebGPU setup to draw triangles via vertex and fragment shader. Image taken from webgpufundamentals.org [Webb].

actions with WebGPU, the hardware is not called directly. Rather, a command encoder encodes the desired actions into commands that can then be stored in a command buffer. Once all desired actions have been accrued, the command buffer can be submitted to WebGPU which will thereafter execute the commands. The *dispatchWorkgroup* command, to give a relevant example, tells the GPU to execute a compute shader.

All rendering is ultimately done to a texture. Usually this texture is requested from a HTML canvas element in the browser and then drawn to in order to present the result of the rendering process.

2.3 LAS

LAS is a file format standard for "storage and distribution of airborne and mobile LiDAR data" [Ise13]. The file standard was created by the ASPRS for the "interchange of 3-dimensional point cloud data data between data users" [ES19].

The file is made up of a header, commonly followed by a body which hold the individual points. Arbitrary user and application-defined data may be interspersed before the body. Their existence and length however must to be noted in the header.

Every point record in the body has an x, y and z coordinate describing the position of the data point. Each of these is stored as a signed 4 byte integer. Rather than storing positions directly as floating-point numbers, LAS uses fixed-precision integers in combination with scaling factors and offsets, which are defined in the file header. This was done to counteract floating point inaccuracies that emerge when working with numbers far away from zero. The scaling factor determines the precision of the stored values and the offset shifts the coordinate space to a local origin. A record also holds other fields such as color, intensity, scan angle, etc.

CHAPTER 3

Related Work

In this section, we will discuss the advancements in point cloud rendering in terms of algorithms and the hardware that made them possible. Because of the importance of point clouds in classification and scene reconstruction we will also explore this field briefly.

3.1 Point Cloud Rendering

The conceptual foundation for using point-based representations for rendering was laid in a work by Leoy Whitted et al. [LW85], where they introduced the idea of using points as display primitives. Using points instead of the traditional polygon primitive enabled the direct rendering of sampled surface data without the need for surface reconstruction.

Improvements in the quality of point-based rendering came from splatting techniques. Surface splatting is used to achieve visually smooth results from unstructured point data. The concept of surfels (surface elements), was introduced in a work by Pfister et al. [PZvBG00]. They represent surfaces using oriented discs that store position, normal, color, and other attributes for efficient rendering of complex geometry without the need for explicit connectivity information. In a work by Botsch et al. [BHZK05], a splat-rendering pipeline based on deferred shading was introduced that takes advantage of the computing capabilities of GPUs. This enabled "high-quality per-pixel shading as well as significant performance improvements" [BHZK05].

A main focus of research is handling the large datasets created by modern sensing hardware. There are many different approaches to combat this issue. Compression algorithms, acceleration structures or visual tricks may be employed to ease the computational burden on hardware. In Potree [Sch15], an LOD system is employed to render large point clouds via WebGL. A adapted version of modifiable nested octree (MNO) [SW11] is employed to segment the point clouds and accelerate rendering.

A different work by Markus Schuetz, et al. [SKW19] proposes to make use of the disjointed nature of point clouds and defines the level of detail continuously. The LOD state also takes not only the distance to the point into account but also considers if the point is near the center of the view, because more attention is paid to that region.

In some applications only the visual result and coherence of the point cloud is important. A work by Ta Hu et al. [HXCJ23] introduces a learning-based method to render photo-realistic images from point clouds. The point cloud is first clustered into a grid structure. It then expands on previous works like NeRF by Mildenhall et al. [MST+21] by building different groupings in x y and z direction that are then used in a Unet to get the features for image construction.

3.2 Compute Shaders

Initially GPUs were hardware designed for a very limited scope. They only handled functions like texture mapping or depth calculations. Using the GPUs hardware for computational purposes was first widely adopted with the advent of the CUDA development platform developed by Nvidia in 2007. Roughly two years later the open standard OpenCL was introduced. Only much later, in 2012 would the first major GPU API - in this case OpenGL - incorporate compute shaders into their core standard. [MS13] Nowadays they are commonly used for various diverse fields such as machine learning, image processing, linear algebra and even stock options trading [Ole18].

CHAPTER 4

Structure of the Rendering Pipeline

This chapter describes the implementation of the point cloud renderer. As depicted in the graphic below 4.1, the renderer goes through four stages to create one final frame. The first stage is a preliminary batch-wise culling on the host side. The pipeline itself has three passes, adopted from the high-quality shading approach of Botsch et al. [BHZK05]. First there is the depth pre-pass, then a colour accumulation pass and finally a display pass. Because the number of workgroups and threads is limited, the depth and accumulation pass may be repeated for multiple batches of points before the final image is displayed. The results of each pass are required in subsequent passes, synchronisation is therefore necessary. In our case, however, this is handled by the command buffer, so no additional statements like fences or barriers are needed.

The renderer always uses depth and frustum culling. No points outside the view are considered for any of the calculations. Nevertheless, the threads to handle them are spawned, so the points still have a small performance impact. Whole batches of points may be excluded outright as we will discus in the following chapter.

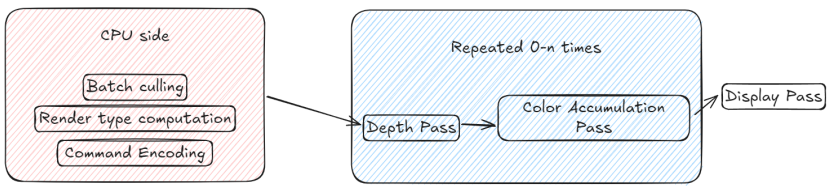

Figure 4.1: Rendering pipeline used for the program.

4.1 Batches and Batch-wise Culling

The renderer holds groups of points as batches. The size of these batches are arbitrary, but later testing proved about four to eight million points per batch to lead to good performance on most systems. These have arbitrary structure and distribution. When a model is loaded in, it is split according to the maximum number of points allowed for one batch. A minimum bounding box is calculated as can be seen in 4.2.

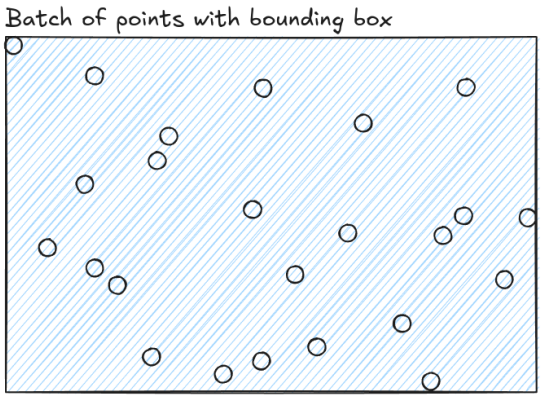

Figure 4.2: Arbitrary distribution of points with minimal bounding box

One batch always has four GPU buffers associated to it. These are the coarse, medium, fine-precision coordinate buffers and the colour buffer. The first three contain coordinate values in cumulatively higher precision, allowing us to process distant points with a reduced pressure on memory bandwidth. During rendering, we load as many coordinate bits as necessary and assemble them back together. For example, for distant points, we only load the 10 coarsest bits. For close points, the coarse, medium and fine bits are merged together to recover a 30 bit fixed-precision integer coordinate.

The first step is to transform the point into the coordinate system of the bounding box. For this the bounding box origin, meaning the minimum x, y and z values as well as the bounding box extent are calculated. The x, y and z coordinates are represented as unsigned 30 bit integers with 0 being the bounding box origin and 2^{30} being the furthest possible point per axis.

The following procedure is illustrated in 4.3. Each coordinate is split into three different ten bit values, the most significant ten bit, the middle ten bit and the least significant ten bit. The more significant a bit is, the more important it is for the position of the point. We therefore call the first ten bits "coarse", the next ten "medium" and the last ten "fine". The coarse x, y and z bits are packed into a single 32 bit integer which thus holds a low precision coordinate of a point. This operation is repeated with the medium and fine bits. The resulting three coarse, medium and fine values are saved to their own corresponding buffer.

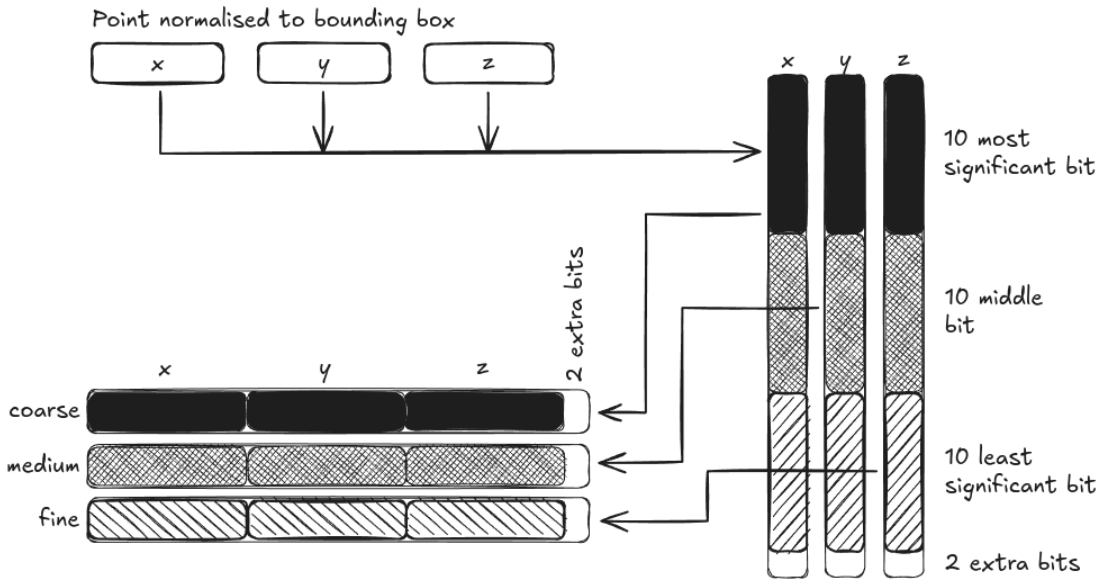

Figure 4.3: Splitting the coordinates of a point into coarse, medium and fine parts for buffer storage.

4.1.1 Course, Medium and Fine Buffer

The real-world position of the points is later reconstructed in the shader. The reconstruction process defines what Level-of-Precision (LOP) level the points are viewed at. If a batch is small on the canvas, the error from using less precision in the calculations is small and only the coarse buffer is needed. The required LOP level is calculated on the CPU side and passed in via a uniform buffer. The benefit of this process is that, even though more operations are needed, time is saved from not retrieving the unneeded buffers from ram. This results in a net performance gain.

4.2 Depth Pass

The entire scene is rendered in the depth pre-pass stage, but only depth values are stored. The colour buffer is not required in this operation. AtomicMin is used to calculate and store only the minimum depth value for each pixel in a buffer for use in the following passes.

4.3 Colour Accumulation Pass

During rendering, multiple points with similar or equal depth value may lie on the same pixel. The colour accumulation pass deals with this issue by adding up the colours of all points that fall into the same pixel and which are at most 1-2% farther away than the closest point. The additions are done using atomicAdd. To make sure the calculations

don't result in overflows, the red, green and blue channel each save their values in a unsigned 32-bit integer. A fourth number, the "opacity", is reserved for keeping track of how many points have been accumulated per pixel.

4.4 Display Pass

The display pass shows the final result on the canvas. This is the only stage that does not use a compute shader. A fragment shader is employed. The accumulated colour values from the previous pass are averaged per pixel. The three colour channels are each divided by the tracked total number of accumulated colours.

A full-screen quad is rendered, the result of the computations is sampled like a texture and drawn to this quad. The browser then ultimately presents this in a HTML canvas.

CHAPTER 5

Results

This chapter showcases the results of the paper. A few different systems as well as operating systems will be tested. The scene is an aerial scan of Morro Bay [Ope13] with 136 million points, filling the entire viewport and looking straight down. For testing the canvas will be set to a resolution of 1280×720 to make different systems more comparable. A visualisation of the rendered scene as produced by the program can be seen in the figure (5.1) below.

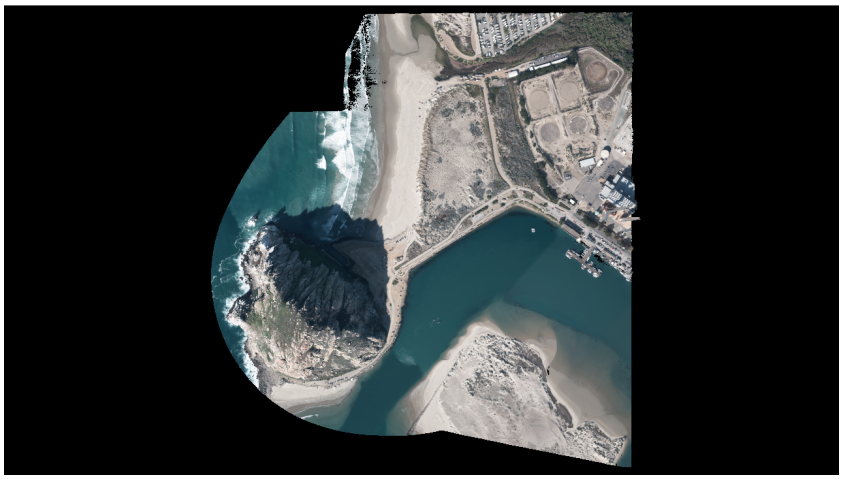

Figure 5.1: Morro bay point cloud (36m points) as rendered by the discussed program.

5.0.1 Testing Methodology

All scenes were rendered from the same viewpoints. Varying factors between tests are the following. As is written in the pipeline explanation, the point information is divided up into three equal parts defining the rendering quality of the point cloud. However, no performance impact could be observed when changing this setting. The performance overhead from transforming more points must therefore be minimal. Because the different quality steps are divided up into their own pipelines and bindgroups, the assumption is that fetching and loading the data on the GPU is faster when a lower quality setting is used. When contrasted with the highest setting, the lower setting only needs a third of the data for computation. As is apparent by preliminary testing, this does not impact performance as much as was anticipated, at least not for the tested model size. This change might still result in a performance impact when larger models are used.

The first parameter for testing is the size of the buffers that the point cloud is divided into. Here, three different configurations will be tested, 32 megabit (roughly two million points), 64 megabit, 128 megabit an 256 megabit. The second parameter is the number of threads per workgroup used for the rendering computations. Here, 64, 128 and 256 threads per work group will be tested.

GPUs tested

GPU	OS	Backend	Best Config	FPS
RX 6600XT	Windows	OpenGLES Direct3D12	128M-256T	45
GTX 1660Ti Max-Q	Linux	Vulkan	128M-256T	52
RX 6600XT	Linux	Vulkan	128M-256T	54
RTX 4080m	Windows	OpenGLES Direct3D11	128M-64T	64
RTX 2060	Windows	OpenGLES Direct3D11	128M-256T	71
RTX 3090	Windows	OpenGLES Direct3D11	128M-256T	145

Table 5.1: GPUs used in testing as well as the used operating system and graphics back-end. It also shows the best setting and what performance was achieved using this setting.

All GPUs used for testing the developed program are listed in table 5.1. Over all test instances, using 128 megabit and 256 threads per workgroups performed the best consistently. This can be further observed when looking at the pacing distributions listed below.

Frame Timings

The following plots use cpu and gpu frame timing informations in milliseconds. The CPU time includes time spent on logic like batch visibility calculations, as well as command encoding and submitting to the GPU. The values are retrieved using timing the according Typescript code portions, over a timespan of five seconds. For timing the GPU compute

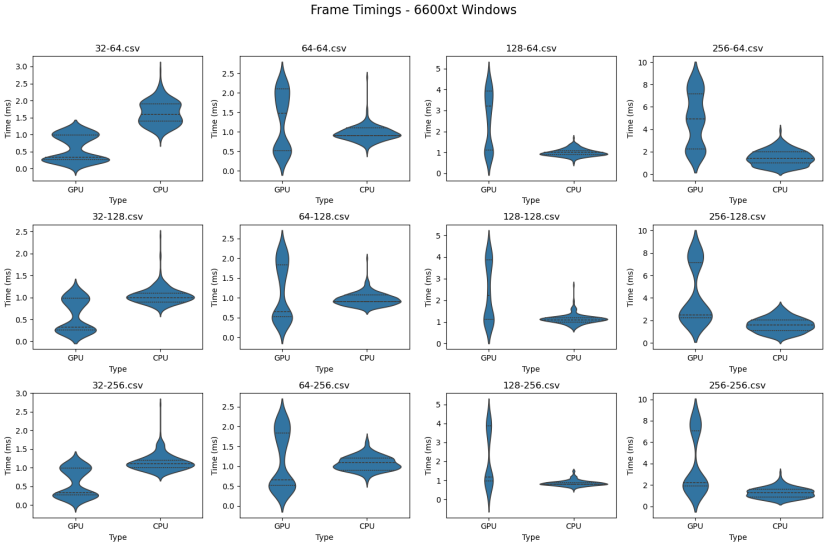

Figure 5.2: Frame timings of 6600XT on Windows

time, a helping script from WebGPU Fundamentals [webc] was used. The shown plots were created using the Seaborn python library. They depict the distribution of the gpu and cpu timings over the five second runs in a violin plot. This makes it possible to see inconsistencies as well as compare relative performance. For a full picture, the table with the best setting as well as the recorded fps at that setting (5.1) can be taken into account.

The 6600xt on windows (5.2) and on linux (5.3 have about the same CPU timings but the GPU times are consistently worse on the windows side. In both runs, however, the best setting was 128 megabit and 256 threads per workgroup.

The 1660Ti graphics card was tested on a laptop in linux (5.4) as well and solid performance can be observed. The best fps was noted with the 128 megabit and 256 threads setting. In contrast to this, the other tested laptop, which utilised an RTX 4080 mobile GPU (5.5), showed strong fluctuations in the CPU timings while the GPU timings remained closely clustered. This might be explained by the CPU being noted to have overheating issues.

On the RTX 4080 mobile GPU, the best fps was observed at 128 megabit and 64 threads (5.1) This could be because the CPU shows large timing inconsistencies at all other settings, as can be seen in the timing graph 5.4. Because the laptop CPU is noted to sometimes have issues with overheating this result should be scrutinized. It might be plausible that the best fps can be achieved with 128 megabit and 256 threads like with the other tested systems, if no such issues were present.

Interestingly, the 2060 had notably better performance in this test than the 6600xt even though they can often be seen with similar performance, with the 6600xt leading most of

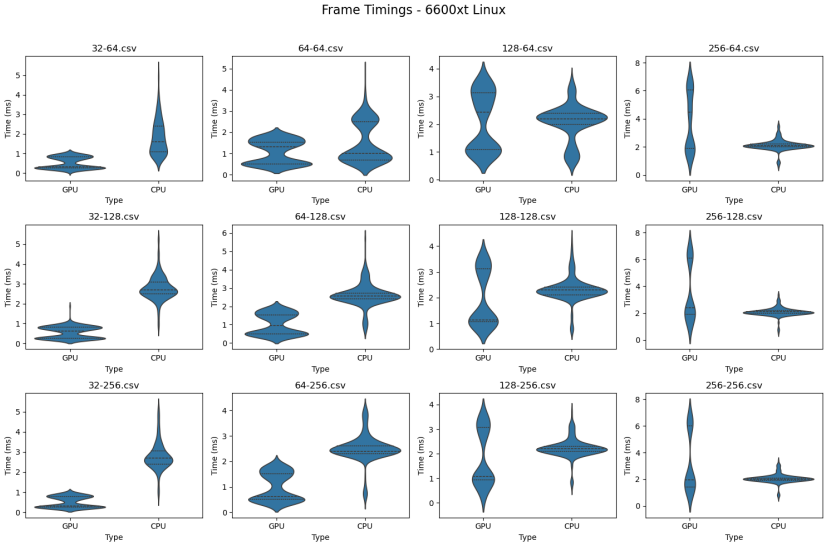

Figure 5.3: Frame timings of 6600XT on Linux

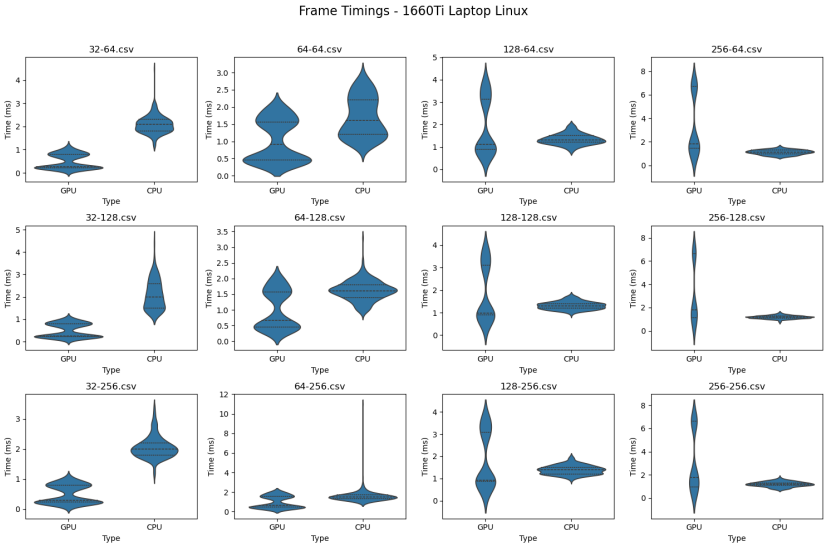

Figure 5.4: Frame timings of 1660Ti Max-Q on Linux

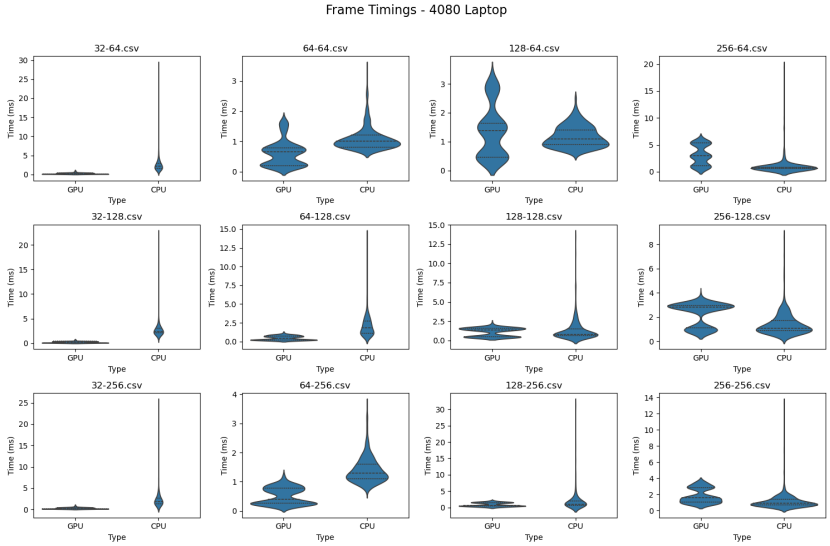

Figure 5.5: Frame timings of 4080 mobile edition on Windows

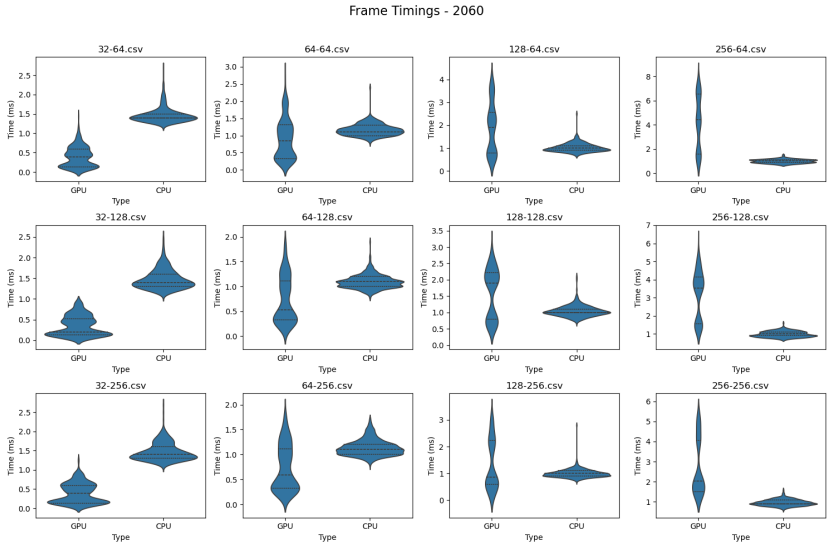

Figure 5.6: Frame timings of 2060 on Windows

the time, when benchmarked in popular games. The 2060s (5.6) best performing setting was, again, 128 megabit and 256 threads with 71 fps. To contrast this, the 6600xt only got 54.

The RTX 3090 (5.7) achieved the highest overall fps at 145 using the 128 megabit and 256 threads setting, far eclipsing the other tested systems. From this result we can infer that this solution scales well with increased computational power.

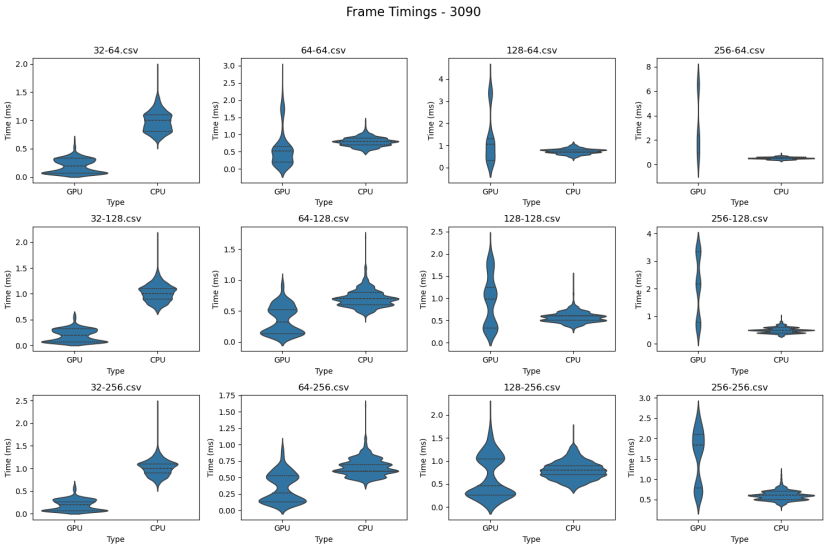

Figure 5.7: Frame timings of 3090 on Windows

Overall robust performance can be observed on the different tested systems. Even on less powerful laptop hardware the program remains usable with 54 fps (5.1) when using the large Morro Bay point cloud with 136 million points. Except for the laptop with the RTX 4080 mobile GPU, the CPU as well as the GPU frame timings are consistent with little outliers. The two bands that are visible in most measurements may represent the different stages the program uses as these were measured independently. The assumption is that calculating depth values takes less time than calculating the colour values, as the latter performs the same calculations and more.

While the program achieved adequate performance, it's implementation deviates significantly from previous works like "Software Rasterization of 2 Billion Points in Real Time" [SKW22] and can therefore not be compared to these implementations.

CHAPTER 6

Conclusion

In this thesis, a program was developed to render LAS point clouds in the browser in real-time. This was accomplished by utilising the users GPU through WebGPU and Typescript. The developed program was tested using a large point cloud and showed adequate performance. This signals that WebGPU can effectively be used to render point clouds in the browser, even at larger file sizes. Future work might include more advanced rendering systems like voxelisation or LOD structures to improve performance.

Overview of Generative AI Tools Used

Übersicht verwendeter Hilfsmittel

List of Figures

1.1 Crack detection of a 30m long point cloud scene. (a) Reflection intensity of the point cloud. (b) Detection result of the proposed method. Image taken from Luo et al. [FLL+22]. 1

2.1 Simplified diagram of WebGPU setup to draw triangles via vertex and fragment shader. Image taken from webgpufundamentals.org [Webb]. 7

4.1 Rendering pipeline used for the program. 11
4.2 Arbitrary distribution of points with minimal bounding box 12
4.3 Splitting the coordinates of a point into coarse, medium and fine parts for buffer storage. 13

5.1 Morro bay point cloud (36m points) as rendered by the discussed program. 15
5.2 Frame timings of 6600XT on Windows 17
5.3 Frame timings of 6600XT on Linux 18
5.4 Frame timings of 1660Ti Max-Q on Linux 18
5.5 Frame timings of 4080 mobile edition on Windows 19
5.6 Frame timings of 2060 on Windows 19
5.7 Frame timings of 3090 on Windows 20

List of Tables

5.1 GPUs used in testing as well as the used operating system and graphics back-end. It also shows the best setting and what performance was achieved using this setting. 16

List of Algorithms

Bibliography

[BHZK05] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*, pages 17–141, 2005.

[ES19] Kristian Damkjer Evon Silvia, Howard Butler. LASer (LAS) File Format Exchange Activities &x2013; ASPRS — asprs.org. `https://www.asprs.org/divisions-committees/lidar-division/laser-las-file-format-exchange-activities`, 2019. [Accessed 03-07-2025].

[FLL+22] Huifang Feng, Wen Li, Zhipeng Luo, Yiping Chen, Sarah Narges Fatholahi, Ming Cheng, Cheng Wang, José Marcato Junior, and Jonathan Li. Gcn-based pavement crack detection using mobile lidar point clouds. *IEEE Transactions on Intelligent Transportation Systems*, 23(8):11052–11061, 2022.

[HXCJ23] Tao Hu, Xiaogang Xu, Ruihang Chu, and Jiaya Jia. Trivol: Point cloud rendering via triple volumes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20732–20741, 2023.

[Ise13] Martin Isenburg. Laszip: lossless compression of lidar data. *Photogrammetric engineering and remote sensing*, 79(2):209–217, 2013.

[khr19] Compute Shader - OpenGL Wiki — khronos.org. `https://www.khronos.org/opengl/wiki/Compute_Shader`, April 2019. [Accessed 30-06-2025].

[LW85] Marc Levoy and Turner Whitted. The use of points as a display primitive. *Stanford*, 1985.

[LYC+23] Jiarong Lin, Chongjian Yuan, Yixi Cai, Haotian Li, Yunfan Ren, Yuying Zou, Xiaoping Hong, and Fu Zhang. Immesh: An immediate lidar localization and meshing framework. *IEEE Transactions on Robotics*, 39(6):4312–4331, 2023.

[MS13] Kurt Akeley Mark Segal. Opengl 4.3 spec, February 2013. [Online; accessed 2025-02-27].

[MSGS11] Quirin Meyer, Gerd Sussner, Günter Greiner, and Marc Stamminger. Adaptive Level-of-Precision for GPU-Rendering. In Peter Eisert, Joachim Hornegger, and Konrad Polthier, editors, *Vision, Modeling, and Visualization (2011)*. The Eurographics Association, 2011.

[MST+21] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

[Ole18] Olena. A brief history of gpu. today, a gpu is one of the most crucial... | by olena | altumea | medium, 2 2018. [Online; accessed 2025-02-27].

[Ope13] OpenTopography. Pgamp;e diablo canyon power plant (dcpp): San simeon and cambria faults, ca, 2013.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 335–342, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[Sch15] Markus Schütz. *Potree: Rendering large point clouds in web browsers*. PhD thesis, Technische Universität Wien, 2015.

[SKW19] Markus Schütz, Katharina Krösl, and Michael Wimmer. Real-time continuous level of detail rendering of point clouds. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 103–110, 2019.

[SKW22] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. Software rasterization of 2 billion points in real time. *Proc. ACM Comput. Graph. Interact. Tech.*, 5(3), July 2022.

[SW11] Claus Scheiblauer and Michael Wimmer. Out-of-core selection and editing of huge point clouds. *Computers Graphics*, 35(2):342–351, 2011. Virtual Reality in Brazil Visual Computing in Biology and Medicine Semantic 3D media and content Cultural Heritage.

[W3C25] W3C. WebGPU Shading Language — gpuweb.github.io. `https://gpuweb.github.io/gpuweb/wgsl/#subgroup-builtin-functions`, 2025. [Accessed 03-07-2025].

[weba] WebGPU Compute Shader Basics — webgpufundamentals.org. `https://webgpufundamentals.org/webgpu/lessons/webgpu-compute-shaders.html`. [Accessed 03-07-2025].

[Webb] Webgpu fundamentals. [Online; accessed 2025-02-26].

[webc] WebGPU Timing Performance — webgpufundamentals.org. https://webgpufundamentals.org/webgpu/lessons/webgpu-timing.html. [Accessed 30-06-2025].

[Web25] Webgpu, 2 2025. [Online; accessed 2025-02-25].

[WSS+15] Jun Wang, Weizhuo Sun, Wenchi Shou, Xiangyu Wang, Changzhi Wu, Heap-Yih Chong, Yan Liu, and Cenfei Sun. Integrating bim and lidar for real-time construction quality control. *Journal of Intelligent & Robotic Systems*, 79:417–432, 2015.