FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Progressive Rendering of Massive Point Clouds in WebGL 2.0 Compute

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Wolfgang Rumpler
Matrikelnummer 01526299

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Markus Schütz

Wien, 10. Oktober 2019

Wolfgang Rumpler	Michael Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Progressive Rendering of Massive Point Clouds in WebGL 2.0 Compute

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Wolfgang Rumpler
Registration Number 01526299

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Markus Schütz

Vienna, 10th October, 2019

Wolfgang Rumpler	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Wolfgang Rumpler

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.

Wien, 10. Oktober 2019

Wolfgang Rumpler

Danksagung

An dieser Stelle möchte ich allen Personen danken die mich bei dem Abschluss dieses Studienabschnitts unterstützt haben.

Die Begleitung und Unterstützung meiner Freunde während des Studiums gab mir die nötige Motivation und Freude es bis hier her zu schaffen. Ich konnte in den vergangenen Jahren vieles von ihnen lernen, egal ob menschlich oder fachlich. Sie haben mir geholfen meine persönlichen Grenzen zu übertreffen und weiter zu kommen als es alleine möglich gewesen wäre. Ich bin ihnen dafür sehr dankbar.

Weiteres möchte ich meiner Familie, die es mir ermöglicht hat mich sorglos auf mein Studium zu konzentrieren und mich in anstrengenden Zeiten ebenfalls dazu motiviert hat weiterzumachen, danken.

Schlussendlich spreche ich meinem Betreuer, Markus Schütz, meinen herzlichsten Dank aus. Seine professionelle Hilfe bei Fragen jeglicher Art war eine Bereicherung während der Umsetzung dieser Arbeit.

Vielen Dank!

Acknowledgements

This section is devoted to all people that assisted me in achieving my bachelor's degree.

The accompaniment and support of my friends provided me with the necessary motivation and joy to achieve my goals, and I learned a lot from them during the last years. They helped me to challenge my limits and to become a better and more thoughtful person. I am grateful for having those friends.

Also, I want to thank my family, who enabled me to concentrate on my education and encouraged me to continue on this path in difficult times.

Finally, I want to give my thanks to my supervisor, Markus Schütz. His professional help and answers to all my questions were an enrichment during the work on this thesis.

Thank you!

Kurzfassung

Das Rendern von großen Punktwolken ist ein rechenintensiver Prozess. Verschiedene Optimierungen sind notwendig um die notwendige Performance für Echtzeitanwendungen zu erreichen. Die Daten werden normalerweise in hierarchischen Strukturen abgelegt um einen schnellen Zugriff und effiziente Sichtbarkeitstests in Punktwolken mit mehreren milliarden Punkten zu ermöglichen. Die Bildsynthese vieler Punkte ist jedoch noch immer eine herausfordernde Aufgabe auf modernen Grafikkarten. Besonders auf mobile Geräte kann es beim Rendern von millionen von Punkten zu schlechten Frameraten kommen. Daher sind Renderingtechnicken, welche progressive die Punkte über mehrere Frames hinweg zeichnen, entwickelt worden um die GPU Auslastung zu reduzieren. Diese verwenden das Ergebnis des vorherigen Frames und fügen zusätzliche Details hinzu. Die Kombination von hierarchischen Strukturen mit dem progressiven Rendering beherbergt ein interessantes Potenzial an Optimierungen.

Diese Arbeit untersucht einen neuartigen Ansatz um riesige Punktwolken progressive im Browser zu rendern, indem die hierarchische Struktur lokal in eine unstrukturierte Menge von Punkten umgewandelt wird. Diese Menge wird dann mittels Compute Shadern gerendert und durchgängig mit neuen Knoten aus dem Octree erweitert.

Abstract

Rendering large point clouds is a computationally expensive task, and various optimizations are required to achieve the desired performance for realtime applications. It is typical to store the point data hierarchically to enable fast retrieval and visibility testing in point clouds that consist of billions of points. However, rendering the selected nodes is still a demanding task for the graphics units on modern devices. Especially on mobile devices rendering millions of points every frame is often not possible with sufficient frame rates. Techniques that progressively render the points of a point cloud were proposed to reduce the load on the GPU. The results of the previous frames are recycled, and details are accumulated over multiple frames. Combining hierarchical structures with progressive rendering, therefore, houses an exciting opportunity for increasing the performance for massive point clouds.

This work investigates a novel approach to render massive point clouds progressively in the browser by transforming the hierarchical structure locally into an unstructured pool of points. The pool is then rendered progressively with compute shaders and continuously updated with new nodes from the octree.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Point clouds typically refer to an unstructured list of points in three-dimensional space and can be obtained by a process called LiDAR scanning [PTC17]. The acquisition of data is rapid, and millions of points can be generated in a few seconds [FTB16]. Such rapid scanning procedures often results in large file sizes that do not fit into the main memory of computers. For example, Schütz mentions a scanning program of the complete United States by the United States Geological Survey (USGS) that results in about 540 terabytes of uncompressed data [Sch16]. Therefore, many considerations went into the development of out-of-core rendering techniques that allow displaying such point clouds in realtime. However, the increasing market share of mobile devices like laptops, smartphones, or the more recent Oculus Quest[1] raise the bar on the requirements of high-performance rendering techniques. Such low power devices struggle to deliver reasonable frame rates when rendering large amounts of points every frame [Sch16]. Furthermore, VR and AR devices also require a very high frame rate and low overall latencies of less than 15–20 ms for immersive experiences that do not result in motion sickness [EPBD18].

Hence, reducing the load on the GPU while preserving the visual quality of point cloud renderings is still an essential subject of research. One approach to achieve this is by progressively accumulating detail over time and only render a subset of data every frame.

1.2 Related Work

QSplat is often referred to as the first out-of-core implementation of a point-based renderer that achieves excellent performance by hierarchically structuring the data for

[1] https://www.oculus.com/quest/

fast LOD retrieval [RL00]. Implementations like the open-source Potree renderer apply a similar approach to render massive point clouds [Sch16].

Futterlieb et al. accumulate detail over multiple frames if the camera is not moved and create a vertex buffer of the rendered points in discrete intervals [FTB16]. This enables them to achieve high performance by only rendering what contributes to the final image.

Ponto et al. show how to render hierarchical point clouds progressively with a suitable performance for virtual reality displays without a trade-off between quality and interactivity [PTC17]. They achieve it by reprojecting the results of the previous frame into the new one and adding a subset of the visible nodes every frame.

Schütz and Wimmer present a progressive approach for reducing the performance required to render large unstructured point clouds in a visually pleasing way [SWMO19]. They generate a new index buffer for a random subset of the available points every frame and therefore render the dataset progressively. This allows rendering a vast amount of points even on low-end devices over several frames.

This work is based on the ideas of Ponto et al. and Schütz and Wimmer. We try to progressively render a hierarchical structure with the visually pleasing random convergence of the approach taken for unstructured point clouds.

1.3 Problem Statement

The goal of this thesis is to investigate a novel approach of how the advantages of the progressive rendering techniques from Schütz and Wimmer [SWMO19] and Ponto et al. [PTC17] can be combined. This would allow us to progressively render large hierarchical point clouds similar to Ponto et al. but with a randomly converging image that looks visually pleasing, as presented by Schütz and Wimmer.

An additional requirement is the implementation of the renderer into the existing code base of the Potree framework that was written in JavaScript with WebGL for modern browsers. The approach taken to solve this problem requires the feature of compute shaders. Since those are not available in WebGL 2.0, the, at the time of writing, experimental API “WebGL 2.0 Compute” is utilized for the implementation.

The main challenges consist of developing a suitable way to implement progressive rendering into an existing hierarchical out-of-core point cloud renderer and to maintain excellent performance with the limited resources provided by the browser.

1.4 Structure of the Work

This thesis is separated into five chapters that discuss the approach taken to achieve the desired results. First, the introduction to the problem and related work is mentioned in Chapter 1. Then the exact methodology is described in Chapter 2. Chapter 3 consists of a detailed description of the implementation. A thorough evaluation of the performance

and quality of the presented technique follows in Chapter 4. Finally, a conclusion about the usability and result of the implementation is drawn, and open questions and potential further research is presented in Chapter 5.

CHAPTER 2

Methodology

It is required to reproject the data from the last frame, and to add additional details, to achieve the desired results over multiple frames. This is done by rendering a subset of the visible points instead of the visible nodes every frame. Schütz and Wimmer shuffle all points of the cloud on the GPU in a buffer of S points. This way, it is possible to render n points every frame and ensure that every point of the cloud is rendered after n/S frames. The image converges to the final quality with every frame. This approach is possible if all data of the point cloud is available right from the start. In the case of an out-of-core renderer, however, we cannot simply dump the data into the memory upfront.

This work, therefore, presents an alternative approach. Our solution utilizes a memory management technique that allows us to fill and manage a shuffled pool of points on the GPU. It allows inserting the points of the visible nodes at pseudo-random positions into the pool during runtime. Nodes that are not required anymore can be removed from the pool if necessary. For the rendering process, every frame a subset of the shuffled point pool is rendered and added to the previous result. The final world positions and colors of the points are stored in a texture for the next frame.

Shuffling a considerable amount of data can be computationally expensive and should be done efficiently in parallel to achieve a fast initialization time. A simple way to calculate the shuffled indices for the pool, based on the perfect shuffle [Sto71], is presented. It enables us to test arbitrary pool sizes and still achieve satisfactory visual results.

Furthermore, the complete rendering pipeline is implemented with compute shaders. Unfortunately, WebGL does not support compute shaders currently. The implementation in the browser is only possible due to the experimental WebGL 2.0 Compute API.

Our approach for the progressive rendering of massive point clouds is implemented in the browser-based point cloud viewer Potree. Potree already supports the rendering of arbitrarily large point clouds via an octree structure, but we replace the way these octree nodes are rendered with our progressive method.

2.1 Potree

Potree[1] is an open-source web browser-based software for viewing large point clouds. It was developed at the TU Wien by Schütz [Sch16] to increase the public availability of the technology required for rendering such large datasets. These datasets typically do not fit directly into the memory of a system and therefore have to be rendered out-of-core. Nodes are requested from a server as needed and cached locally. The renderer iterates through all visible nodes and renders each with a single draw call.

The existing code base of Potree is used as a basis for this work. The implementation of the network logic, the local caching, the hierarchical data structure, and the visibility testing was left as is. The new implementation replaces only the renderer and is incorporated with the rest of the framework.

2.2 WebGL 2.0 Compute

WebGL 2.0 Compute is a browser API specification that closely matches the specification of OpenGL ES 3.1. The API allows utilizing features such as compute shaders, shader storage buffers, and atomics [Gro19]. These features result in a plethora of new use cases for WebGL in the browser. However, at the time of writing, the specification is a work in progress. No final implementation has shipped in any browser. The developers of the chromium browser implement the specification in the Blink rendering engine, and as of May 2019, the experimental implementation can be activated in the Chrome browser [Dev19].

[1] https://potree.org

CHAPTER 3

Progressive Rendering

This thesis presents a novel and experimental implementation of a point cloud renderer that utilizes the, at the time of writing experimental, WebGL 2.0 Compute API to render large hierarchical point clouds progressively in the web browser. It resembles the attempt to implement the work presented by Ponto et al. [PTC17] and Schütz and Wimmer [SWMO19] into the existing out-of-core point cloud renderer "Potree".

This goal is achieved by continuously loading nodes of the octree into a pre-shuffled point pool and writing a simple memory manager that keeps track of and manages the stored nodes. This procedure effectively reduces a subset of the hierarchical structure into a sequential array that can be rendered with the approach presented by Schütz and Wimmer. The data from the previous frame is reprojected. Then a specific amount of new points is rendered from the pool and added to the result. The entire rendering pipeline is thereby implemented with compute shaders.

The accomplished raw performance is comparable to the original Potree renderer that draws all visible nodes every frame. Due to the progressive rendering, it is possible to render far fewer points every frame and still get the same visual quality over multiple frames. This characteristic reduces the amount of processing that has to be done on the GPU and can increase the responsiveness on slower devices if the point pool is large enough. An additional advantage of the implementation is the ability to allocate an arbitrary amount of memory on the GPU for the point pool. This trait allows easy adaptation to the executing device.

The content of this chapter is separated into seven different sections. The main components of the implementation are the initialization of GPU memory, storing nodes on the GPU, the management of the GPU memory, and the actual rendering. These parts are explained in detail in the following sections.

3.1 Memory Requirements

It is required to allocate enough memory on the GPU when the renderer is instantiated. Three different primary buffers are required for the presented implementation of the point pool. All buffers are created as shader storage buffers to allow read and write access from a shader.

First, a large buffer for the point pool itself is required. It is used to hold all the necessary data of the points that are uploaded to the GPU. This buffer is called the "point pool" and its size is directly related to the maximum number of points that have to be managed and potentially rendered. For example, if the data of a single point consists of three 32 bit floats for the coordinates and a single 32-bit float that contains the packed RGBA data, then it would be necessary to allocate about 382MB of memory on the GPU for a pool size of 25 million points.

The second buffer that is required is the "permutation index buffer". This buffer is a simple lookup table to know where a specific point resides in the point pool and is required by the memory manager for correctly allocating and freeing memory. The size of this buffer is directly related to the point pool size. Every entry in the pool requires an index. A 32-bit integer is used for indexing into the pool, since the shading language GLSL ES 3.1, that is supported in WebGL 2.0 Compute, only specifies up to 32-bit integers. Lower precision integer values may vary in bit size depending on the implementation [Gro19, Gro16]. In most cases, this is sufficient as a 32-bit integer allows for indexing a 64GB point pool with the compact point data described earlier. In the case of 25 million points, this permutation index buffer is about 96MB in size.

The last buffer is required to upload new data into the point pool. All points are passed through this buffer into the point pool. It is the only buffer of the three into which the CPU writes data directly and frequently. The size of this "stream buffer" is determined by how much data has to be uploaded at once. It also depends on how the data is provided to the renderer. In this implementation, nodes are uploaded one at a time, and the buffer is large enough to fit the data of 100,000 points. No nodes that contain more points have been observed. A node that would contain more points would only be uploaded partially. The existing implementation of Potree provides the data as two arrays for position and color. Those arrays are directly copied into two separate stream buffers. This data is then processed and packed together during the insertion into the pool.

3.2 Initialization

As soon as the memory for the buffers is allocated, it is required to initialize the permutation index buffer. Every element in the buffer points to precisely one random index in the point pool, and no element is indexed twice. This mapping between the indices in the permutation index buffer to random indices in the point pool allows managing the memory in chunks.

Such a mapping is also known as a pseudorandom permutation (PRP) [LR85]. PRPs can be calculated in many different ways. Calculating a permutation sequentially with the use of the linear congruential method is relatively straightforward. A linear congruential sequence can be obtained with Equation 3.1 and suitable numbers for a, c and m [Knu14].

$$x_i = (a * x_{i-1} + c) \bmod m \tag{3.1}$$

Two problems with this method prevent a useful application in our case. First, the choice of the parameters a, c, and m is essential. Hull and Dobell show how these numbers have to be chosen to generate a permutation [HD62], but this prevents from quickly changing the size of the pool. The second disadvantage is that the indices can not be calculated efficiently in parallel. However, it would still be possible to sequentially initialize the permutation on the CPU for a specific size with this approach.

There are other ways of calculating PRPs as well, but to simplify the implementation and to allow any arbitrary pool size that is divisible by two, we decided to use a non-random shuffle that is based on multiple iterations of the perfect shuffle. The perfect shuffle can be described as dividing the elements of a set in half and then continuously selecting the next element alternating between the two halves. This process produces sufficiently good visual results in many cases and can be computed trivially on the GPU. One iteration of the mapping for a given index i can be calculated by Equation 3.2 where p is the pool size.

$$x_i = \frac{p}{2} * (i \bmod 2) + \left\lfloor \frac{i}{2} \right\rfloor \tag{3.2}$$

The best shuffling in regards to the visual result is achieved with $[\frac{\log_2 p}{2}]$ iterations with this method. This number is the highest iteration count before the elements start to become ordered again. For example, if the size p is a power of 2, then $\log_2 p$ iterations results in the initial state and not shuffle anything at all. Figure 3.1 shows an index buffer with 16 elements and two iterations of the perfect shuffle.

Since the position of an element is deterministic and can be computed quickly, the permutation index buffer could be skipped, and the index calculated whenever needed. However, it is useful for trying different permutations without changing other parts of the code and would be necessary if some permutation is used that cannot be calculated. Such a scenario, for example, would be if the permutation is pulled from a pre-calculated file or if the calculation is too complex to calculate it every time it is needed without significant performance impact.

Unfortunately, this shuffling has a drawback. Not all pool sizes produce equally good results if the points are ordered spatially within the nodes. This problem is depicted in detail in 4.2.2.

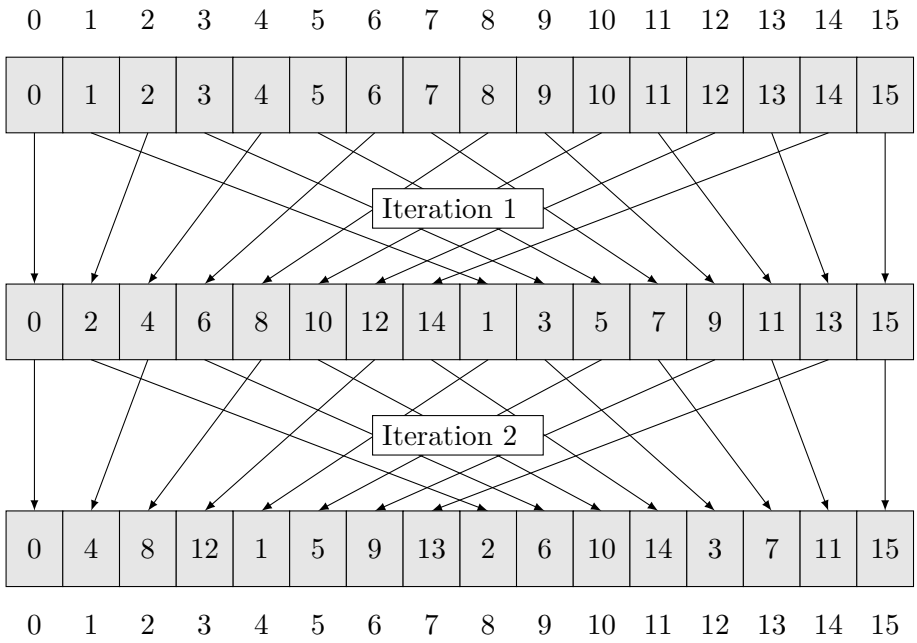

Figure 3.1: Two iterations of the perfect shuffle with 16 elements. This shuffling would be the suggested pattern.

3.3 Storing Data on the GPU

Potree serves the renderer every frame with all nodes of the octree that currently reside in the main memory. However, only the visible nodes are processed further. All points in these nodes should be loaded into the point pool for the rendering. The amount of points in these nodes depends on the point budget that is configured. This point budget defines the number of points that should be rendered in a particular view over multiple frames.

For every node, the memory manager is asked if the node already resides in the pool. The manager requests a new block of storage with the size of the node if this is not the case. The data is copied into the stream buffers. A compute shader is then invoked, which processes the data, packs every point into a single four-element vector and inserts it into the right position in the point pool. The target position for the insertion is looked up in the permutation index buffer according to the assigned block of storage. If no space is left in the pool, then the node replaces some previously stored data.

The number of nodes uploaded in a single frame can also be limited. In this case, a random subset of nodes is chosen and uploaded. This prevents large spikes in the frame time if suddenly many nodes have to be uploaded. Unfortunately, this feature has a significant drawback in the presented implementation. The nodes uploaded in the previous frames are not tracked explicitly. Therefore, the same nodes may be uploaded over and over again if other nodes immediately overwrite them. This scenario is only the case if the pool is tiny in comparison to the amount of data that has to be rendered

for a particular view. Under these circumstances, no time frame can be specified that guarantees that all visible points have been rendered. However, this is typically not the case, and most of the time, the buffer is at least as large as the point budget.

3.4 Memory Management

3.4.1 The Manager

The memory manager administers the allocation and freeing of the memory on the GPU. This task can not be done directly for the point pool because the points in a node are scattered throughout the buffer. Direct management would require to track every point and its position separately, and this would take too much time and memory. Therefore, the implementation utilizes the permutation index buffer for the management and allocates a continuous block for every node in the range of indices. To find a block of storage that can be allocated, a simple first-fit approach is taken. According to Shore [Sho75] and Bays [Bay77] this approach seems to be the most efficient because the allocated blocks are only a tiny fraction of the available memory.

The data structure of the manager is a simple doubly-linked list of memory management entries (MME) that represent either a free or allocated block of storage. Such MMEs consist of the start address, size, and the status if it is free. Additionally, they store an "importance" field that represents how important the node currently is. This field is required for the garbage collection and explained in 3.4.2. Initially, only one MME that occupies the complete index range is stored. Finding a block for allocation is done by the manager by iterating through the list and selecting the first block that is large enough for the requested size of r. This block is then split into two new MMEs such that the first block is exactly of size r, and this block is returned.

The function `free()` of the manager is called with the MME as an argument to free an MME. This function flags the block as free and merges it with the surrounding free blocks. The data in the pool is not removed if a block is freed. This fact results in "ghost" points that reside in the pool and are rendered until they are overwritten. These points are not a problem since they are still rendered correctly and do not lead to any unusual artifacts.

Figure 3.2 depicts a simple index buffer with two allocated nodes and their mapping into the point pool.

3.4.2 Garbage Collection

The garbage collection is executed whenever there is no space left in the GPU memory for allocation. It is crucial to not free nodes from the pool that are currently in the view. Otherwise, the viewer might be confronted with missing points in the rendering that correspond to the missing nodes. Furthermore, disturbing flickering artifacts may appear

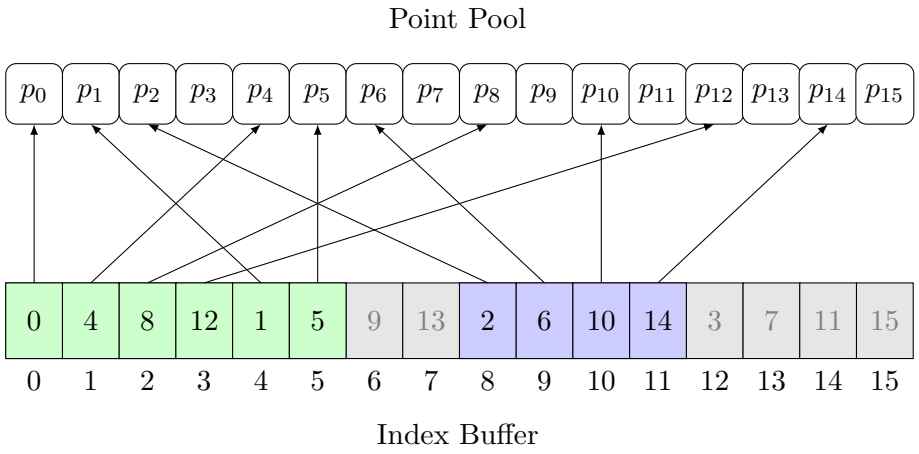

Figure 3.2: A visualization of how the memory manager manages the point pool. The two colored regions each represent a node.

if nodes from the view are continuously removed and inserted into the pool again during movement.

Therefore, the memory manager keeps track of the essential nodes in the memory. The tracking is done by using the additional importance field of the MME. Every time the manager is asked if a node already resides in the pool, this field is incremented. After every frame, the field is decremented for all stored nodes. This way, nodes that are not in the view continuously lose importance. The node with the least importance is freed if no space is left for a new node. This procedure also favors older nodes over recent ones.

In Chapter 4 it is noted that the implemented memory manager seems to be the single most significant bottleneck and, therefore, should be optimized as much as possible. Additional criteria like the distance to the camera, possibly improve the overall quality of the garbage collection and rendering. However, such criteria were not evaluated in the course of this thesis. Another option for improvement would be to bulk the insertion and deletion of small nodes such that the overhead of memory management and the synchronization between CPU and GPU is reduced.

3.5 Rendering

The rendering procedure is executed after all nodes for the current frame have been uploaded to the GPU. It consists of four stages, the reprojection of the last frame, the rendering of the new points, a depth pass to determine the visibility, and the combination of the results. All of the stages are computed directly in compute shaders, and the intermediate results are stored in simple 32-bit RGBA textures. After the stages, the result is rendered to a full-screen quad for displaying it on screen.

It is not strictly necessary to render the points with compute shaders only. The traditional

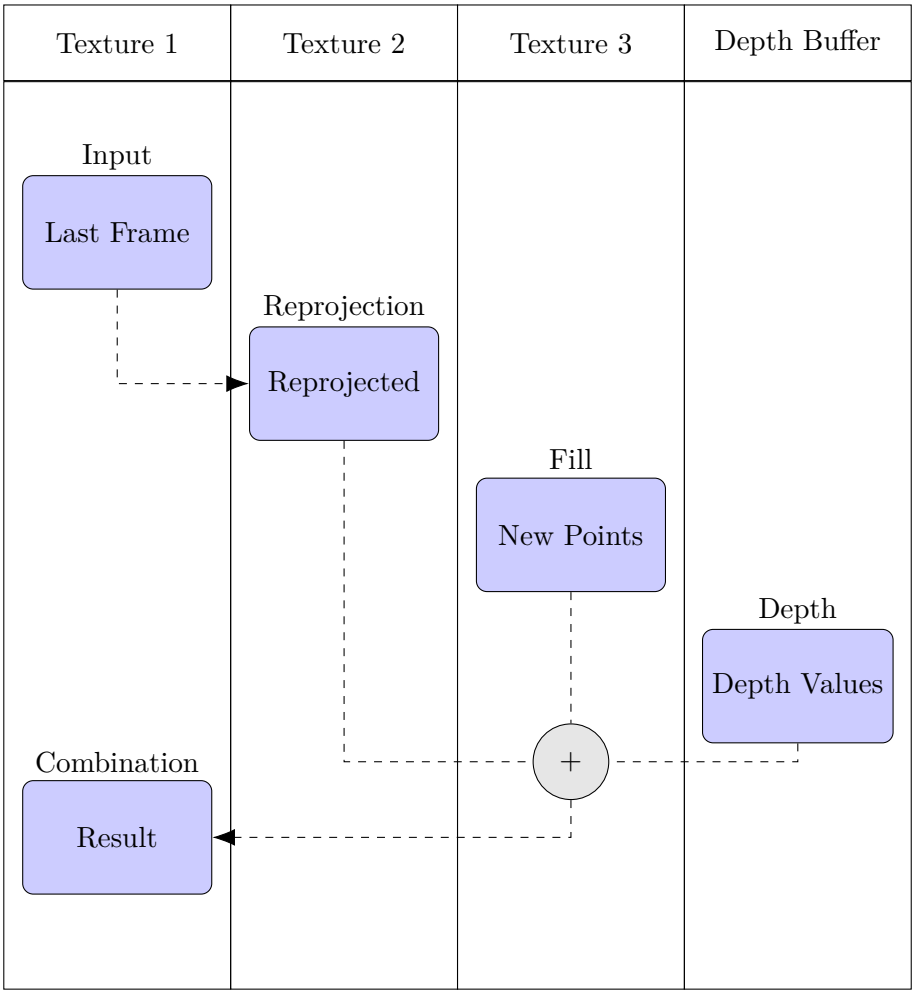

Figure 3.3: The rendering process. In each frame, the previous frame is reprojected, additional points are rendered to fill missing data, and then the data is combined.

pipeline can be used as well, as shown by Schütz and Wimmer [SWMO19]. The approach presented resembles an experiment that depicts the performance that can be achieved with the use of compute shaders only.

The three textures for the reprojection, the new points, and the result contain the three coordinates of world position of the points and the color in the fourth component. Additional textures or another way of storage is necessary if additional properties like normals are required for the rendering. It is not possible to store indices to the date as the data in the pool could change. This would result in invalid reprojections.

Figure 3.3 shows an illustration of the entire rendering process.

3.5.1 Reprojection Pass

Two buffers/textures are required for the reprojection. One is used as a source and contains the result of the last rendering. The other one is used as an empty write target. Empty pixels in the source are immediately skipped. Otherwise, the corresponding world coordinates are read and transformed into the clip space with the current view transformation and projection matrix. The points are clipped and transformed into image coordinates. Finally, the data gets stored in the write target. The source texture is cleared after the reprojection because it is used as a write target for the final result in the combination step.

3.5.2 Fill Pass

For the fill pass, a simple compute shader is dispatched that directly draws the next n points from the point pool to a texture. The complete pool is, therefore, rendered every S/n frames, where S is the pool size. The size of n, also called the fill budget, thereby defines the speed of convergence.

However, a point in the point pool might have never become initialized or contains "ghost" points that have been freed by the memory manager but were not removed from the point pool. These points are not a problem, because we can assume that they are distributed over the point pool due to the shuffling pattern that is described in 3.2. Furthermore, the "ghost" points do not produce any artifacts as they are projected correctly as all other points. They might even contribute to the overall convergence if they happen to be visible in the current view.

3.5.3 Depth Pass

The depth value of every point from the reprojection and the fill pass is rendered into a depth buffer using a minimum atomic operation with its final screen space size. After this pass, the minimum depth for every pixel is stored in the buffer. These values are used in the next step to determine what point has to be written to the final result.

There are two problems with this implementation. First, if two points on the same position have the same depth value, then the data that should be rendered in the next step is ambiguous. This might result in z-fighting of the points. Furthermore, no depth testing was performed in the previous stages. The reprojection in 3.5.1 and the rendering of new points in 3.5.2 have no correct visibility testing. The depth test is a separate pass and only resolves the visibility between those two previous passes with the appropriate point size. It is, therefore, possible that the closest points are never rendered to the final output because distant points overwrite them.

In order to avoid performance losses from overdraw, the points are rendered with a fixed screen size of 1px in the fill and reprojection pass. The final screen size for a point is only processed during the depth and the combine pass and, therefore, if and only if it was visible in one of the previous passes.

We currently do not apply proper depth testing in our thesis due to the way that point rendering was implemented with compute shaders instead of the regular glDrawArrays based WebGL rendering pipeline. This issue can be resolved by using the standard OpenGL rendering pipeline or by implementing a depth buffer in the compute shader based approach, e.g., by encoding depth and colors into a 64-bit integer and finding the closest point via atomicMin [SW19]. However, the second solution can not be applied in web browsers since the shading languages for WebGL 2.0 Compute and the upcoming WebGPU standard both do not support 64-bit integers [Gro16, MPM18].

3.5.4 Combine Pass

The next step is to combine the results of the reprojection, the new points, and the depth buffer into a single texture. All points are rendered with the same procedure as in the depth pass. The calculated depth is compared to the stored depth in the depth buffer. If both depths are equal, the color of the point is stored in the output texture. This approach can lead to z-fighting, as explained earlier. It would be necessary to identify every point in the depth buffer uniquely to prevent this problem. The artifacts also become more evident if the point size is increased.

3.5.5 Displaying the Result

Finally, the combined data is rendered to a screen-filling quad. The point data is selected from the result of the last stage based on the currently rendered fragment. The color is extracted from the fourth component of the data and stored as the fragment color.

3.5.6 Floating Point Precision

OpenGL ES 3.1 does not support 64-bit floating point numbers [Gro16]. This is a problem for large point clouds and point clouds that require very high precision. The precision of a floating-point number depends on its magnitude or in other words, its distance to zero. This is due to how floating-point numbers are represented in binary [Tho07]. The further away the camera is from the origin, the higher is the floating-point error. A significant enough error results in the inability to represented points accurately. Different approaches to reduce this imprecision exist [Tho07].

Potree provides the points of every node relative to the node's origin. It would be sufficient to calculate the model view matrix, that transforms a point into the view of the camera, on the CPU with 64-bit precision to mitigate the floating-point imprecision. However, this is not really an option because it would require to remember all model matrices for all rendered points to be able to do the reprojection.

The selected solution is to shift the origin to the camera's position if the camera is moved by a distance of 10,000 units. All points in the pool are inserted relative to the current origin and, therefore, can be represented with high precision near the camera. The downside of this approach is that the complete point pool has to be cleared. This

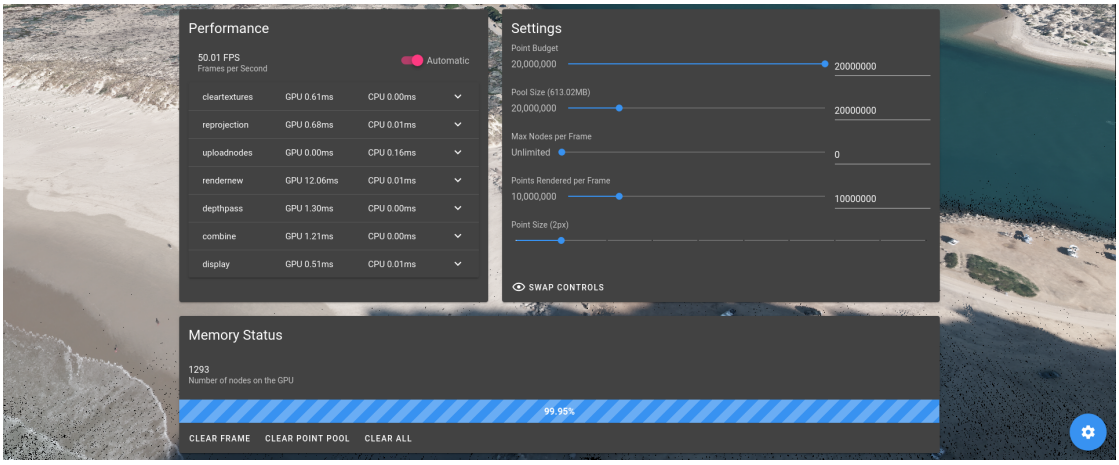

Figure 3.4: The control panel.

also requires to reinsert all points again when the origin is shifted. Otherwise, the old points would be rendered in the wrong positions. This also leads to a small lag as the pool has to be cleared in a single frame.

3.6 Control Panel

The control panel is a simple HTML interface that allows to change settings and provides insights into the current status of the renderer. Figure 3.4 shows this interface. It presents the current memory status, the timings for the different stages of the renderer, and the current configuration. Displaying this information helps to recognize bottlenecks and problems in the implementation.

The interface can be opened by clicking on a floating button in the right lower corner of the screen. It is overplayed over the rendering canvas, so it does not interfere with the performance by altering the resolution.

3.7 Profiling

A simple profiler was implemented that allows measuring the time spent on the GPU and the CPU for every stage in the rendering process. OpenGL Timer Queries are used for measuring the time spent on the GPU. The execution time is also measured on the CPU by sampling timestamps before and after it. A second version of the same profiler was ported to the old renderer to enable a fair comparison of the two implementations.

CHAPTER 4

Evaluation

This chapter depicts how the renderer was evaluated and the implementation compared to the previous implementation of the Potree renderer. The primary area of interest was the performance of the renderer. Additionally, the visual quality of the rendering was evaluated. All testing was done using the point cloud CA13 that was used already used by Schütz in his work [Sch16]. The original data set is provided by the OpenTopography Facility and is named "PG&E Diablo Canyon Power Plant (DCPP): San Simeon and Cambria Faults, CA" [Fac13]. It consists of about 17.7 billion points and thus is a useful data set for the tests.

4.1 Performance

To evaluate the performance of the renderer, we measure the time required on the CPU and the GPU in different scenarios. All stages of the rendering process are tracked to determine the exact time needed for the different processes. These stages are described in detail in the chapter 3 about the implementation.

Probably, due to the experimental state of WebGL 2.0 Compute at the time of writing, substantial performance differences have been observed between different builds of the Chrome browser. All tests were therefore performed with the same version of the browser to ensure comparable results. The version used is "Version 78.0.3904.34 (Official Build) beta (64-bit)".

Table 4.1 lists the hardware that was used to perform the tests. Every system has a name assigned to it that is used to identify it throughout this chapter. Unfortunately, no test could be performed on mobile devices and integrated graphics cards of laptops. Mobile devices currently have no support for WebGL 2.0 Compute, and integrated graphics cards did, unfortunately, not render correctly or did not work at all.

Name	GPU	CPU	OS	Resolution
Desktop 1	GTX 960	i5-6600K @ 4.4Ghz	Linux 5.3.1	1920x1200
Laptop 1	GTX 1050	i7-8565U CPU @ 1.80GHz	Linux 5.3.1	1920x1080

Table 4.1: The hardware configurations used for the evaluation.

4.1.1 Performance of the Different Stages

Table 4.2 provides an overview of the performance of the different rendering stages on different systems. All measurements were taken from the same camera view with the exact same settings over 10 seconds.

The configuration was a point pool size of 10 million points, a point budget of 20 million points, 10 million points rendered per frame, and a point size of 2px. The point budget was selected to be higher than the pool size to ensure that the memory manager has to allocate and free memory consistently during the rendering. We therefore also have to limit the nodes per frame to 10, since the memory manager would otherwise consume a significant amount of time.

The large amount of 10 million new points every frame was selected to reduce the frames per second below 60 since disabling v-sync for the browser was not possible. A smaller amount, for example, 1 million points would be sufficient for fast convergence of the rendering. The performance for smaller amounts is evaluated in 4.1.3.

		Desktop 1 (ms)			Laptop 1 (ms)		
		Avg	Min	Max	Avg	Min	Max
Clear Textures	GPU	0.57	0.37	0.99	0.70	0.47	0.95
	CPU	0.00	0.00	0.35	0.00	0.00	0.03
Upload Nodes	GPU	1.96	1.03	3.48	2.60	1.59	3.60
	CPU	1.19	0.77	7.03	1.83	0.67	4.82
Render New Points	GPU	11.81	11.63	11.98	11.72	11.51	11.92
	CPU	0.01	0.00	0.04	0.01	0.00	0.03
Reproject Points	GPU	0.70	0.69	0.72	0.63	0.63	0.64
	CPU	0.01	0.00	0.02	0.01	0.00	0.03
Depth Pass	GPU	1.19	1.17	1.20	1.14	1.13	1.15
	CPU	0.00	0.00	0.06	0.01	0.00	0.05
Combine	GPU	1.25	1.24	1.27	1.36	1.36	1.37
	CPU	0.00	0.00	0.04	0.01	0.00	0.03
Average FPS		47.60 FPS			28.30 FPS		

Table 4.2: Timings of the different rendering stages on CPU and GPU

These measurements show primarily two noteworthy aspects. First, the rendering of the new points is the most significant part of the rendering process and should be optimized as much as possible. However, it is not required to set the number of points that are rendered so high, and the overhead might shift towards the reprojection as the resolution of displays grow. For example, a resolution of 3840 x 2160 where every pixel is covered by a point probably results in a significant amount of time since this would be $8,294,400$ additional points to render. The second fact is that the CPU is not utilized a lot during the rendering. The CPU performance plays only a role when uploading nodes. Essentially, this is the only section during the renderer where logic has to be run on the CPU side. The high maximum of $7.03ms$, on the other hand, might be a problem, as such behavior can lead to inconsistent frame rates. Optimizations to the memory manager would probably yield better overall performance and minimize the effect of these outliers. Especially since uploading ten nodes already requires 1-2 milliseconds of work.

4.1.2 Performance Impact of Points outside the View

This test evaluates the impact of points that are inside the point pool but do not contribute to the final output because they are not within the view frustum. It is done by measuring the time required for rendering the complete buffer when it is half-full in a static view. The camera is then turned 180 degrees until all nodes from the view are inserted into the pool. Then the camera is turned again, and a new measurement is taken. The difference between the two measurements shows the overhead of the points that do not contribute to the final image.

The point pool size and the render amount was set to 40 million. The point budget was set to 20 million to fill the pool with two views.

The results of the measurements are displayed in the Table 4.3. Additional points in the pool do not contribute to the frame time with any significant amount. The rendering times for the new points and the frame rate stay nearly identical. However, this shows that we have to render more points per frame to achieve the same speed of convergence if the pool is enlarged. For example, if the pool is twice as large as the point budget, we have to render twice as many points to render all points that contribute to the view.

	Desktop 1					
	Half-Full Pool (ms)			Full Pool (ms)		
	Avg	Min	Max	Avg	Min	Max
GPU	25.19	25.10	25.39	25.19	25.11	25.27
CPU	0.01	0.00	0.03	0.01	0.00	0.06
	31.40 FPS			31.50 FPS		

Table 4.3: Measurements for rendering a half-full and a full point pool.

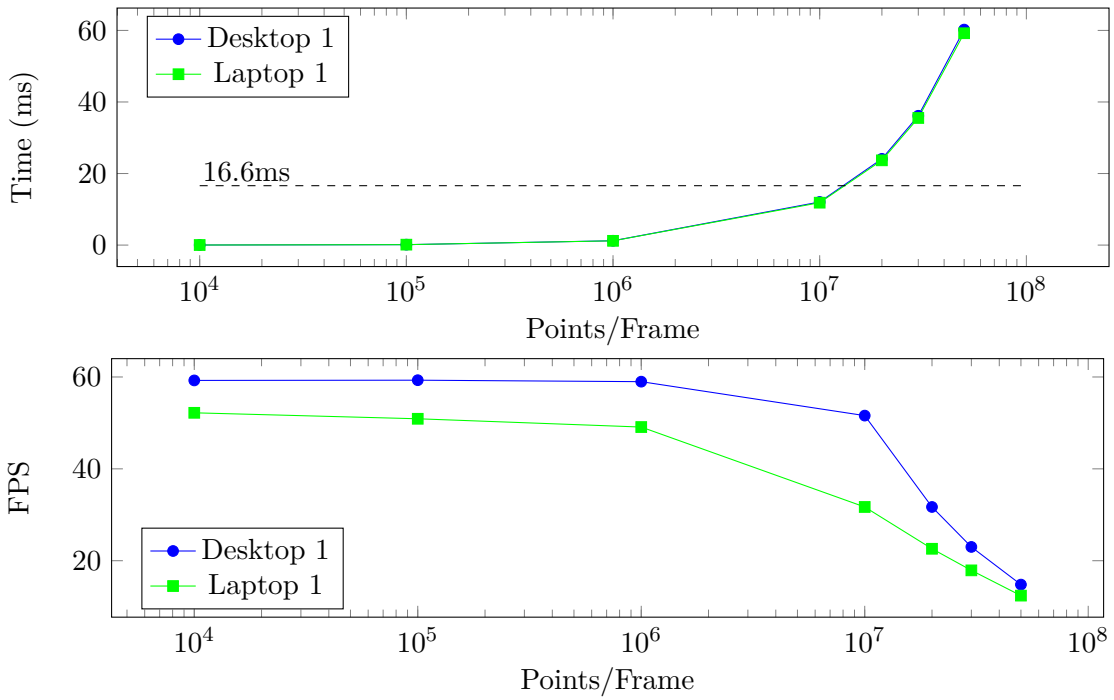

Figure 4.1: The performance impact of the fill pass.

4.1.3 Number of new Points per Frame

A series of measurements were taken to test the isolated performance of the fill pass. Each measurement was recorded with a different number of points rendered per frame.

The point pool size is set to 20 million and is filled as close to 100% as possible by selecting a fixed position for the camera and setting the point budget to the pool size. Since the point budget determines the number of points loaded in a given view and the pool is filled sequentially, we can guarantee that results are reproducible. During the tests, 99.95% of the point pool stayed filled, and the memory manager required nearly no time as no nodes had to be freed or loaded to the GPU. Most points are rendered to the screen since no nodes from outside the view frustum were stored on the GPU.

The results of the fill pass, and the overall frames per second are visible in Figure 4.1. This information can be used to reduce the amount of rendered points if the desired frame time is not met. The difference in FPS comes from the weaker CPU performance of the laptop that affects other parts of the point cloud viewer.

On the tested graphics card, up to 10 million points can be rendered to the screen with an acceptable frame rate.

4.1.4 Comparison to the Potree Renderer

The original renderer does not have the same stages as the compute implementation. Every node is rendered by a single draw call. Therefore, only the overall time required by the renderer is measured and compared against our implementation.

The settings are set to be as equal as possible. The EDL (Eye Dome Lighting) is turned off in the original renderer. The point size is set to two, and the point budget is increased to 10 million.

The Table 4.4 shows the result of the test. It lists the measurements of the original, the compute, and the compute renderer with the continuous work of the memory manager. The two measurements of the compute renderer use the settings from 4.1.1. The point budget was set to the same size as the render amount to measure the time without the load of the memory manager.

In general, the raw performance of the two implementations is equivalent, but the new implementation might house a good potential for further optimizations. Since there is no immediate advantage in raw performance over the original renderer, we consider our implementation of the renderer with only compute shader as a drawback as it introduces difficulties with the depth testing. We, therefore, do not generally recommend to approach the rendering the same way. However, the overall method for the progressive renderer seems to be advantageous since it is not required to render all points every frame and hence can save a substantial amount of processing time on the GPU.

A notable observation of the original renderer is that the CPU requires nearly as much computational time as the GPU. While we don't know for sure, we presume that this is caused by inefficiencies of how the hundreds of nodes that are rendered through hundreds of WebGL draw calls and uniform operations. Our approach, on the other hand, only requires a few calls to reproject previous data and then draw a subset of the point pool.

	Desktop 1								
	Original (ms)			Compute (ms)			Compute + MM (ms)		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
GPU	18.38	17.00	25.69	16.96	16.35	17.69	18.69	17.71	20.54
CPU	18.01	17.68	19.42	0.59	0.44	2.25	3.00	2.09	12.83
	48.18 FPS			52.90 FPS			47.60 FPS		

Table 4.4: Comparison of performance of the original, the compute, and the compute renderer with a continuous load of the memory manager.

4.1.5 Memory Utilization

The efficiency of the memory manager was tested by measuring how much memory was utilized after 10 minutes of continuous usage. The camera was moved randomly over the

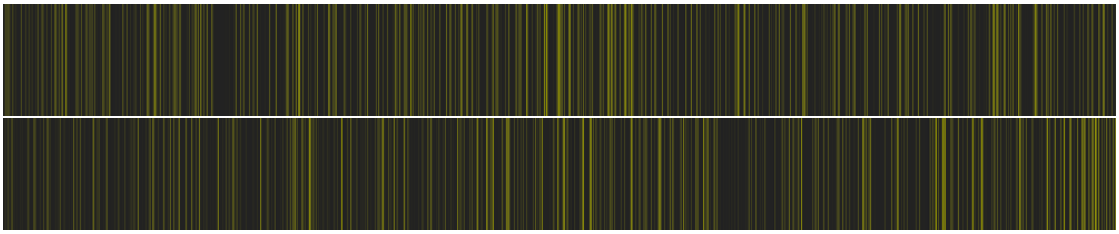

Figure 4.2: A visualization of the fragmentation of nodes in the point pool after 5 and 10 minutes.

map until the memory manager started freeing memory. This procedure initialized the to pool with a random fragmentation. Then the random movement was continued for two periods of five minutes. After each period, the camera was stopped until all nodes of this view were stored on the GPU. The percentage of memory used at this point provides a rough overview of the efficiency of the allocation strategy and the effect of fragmentation.

The settings used for this test were a point budget of 20 million, a pool size of 25 million, and an unlimited amount of nodes that are allowed to be uploaded every frame. The other settings do not impact the results of this test.

The first pass revealed a memory utilization of 90.50%, and the second pass ended with a nearly identical utilization of 90.71%. This suggests that the utilization is stable but also shows that the memory manager reduces the speed of convergence by about 10% since this storage is not used for point data and is still processed. The allocated blocks are scattered throughout the pool since the memory manager performs no defragmentation. The fragmentation pattern for both passes is visible in Figure 4.2. Defragmentation could enhance the utilization but most likely adds additional overhead to the memory manager that diminishes the gains in convergence speed.

The test was repeated with the same settings except with a very larger point pool of 100 million points. This pool has roughly a size of 2 GB. The first pass utilized 96.86% of the memory, and the second pass resulted in the utilization of 95.34%. These results hint that the utilization of the memory is getting better with larger pool sizes.

4.2 Visual Quality

4.2.1 Point Size

The point size has a significant impact on visual quality. During camera movements, many of the reprojected points are projected onto the same positions, and holes appear in the rendering. This can be enhanced by increasing the point size such that the holes remain filled by the remaining points. It turns out that a fixed point size of two by two pixels is large enough to achieve a very consistent view of the scenery. Figures 4.3, 4.4, and 4.5 show a comparison of a still image and a moving camera for near and far objects with different point sizes.

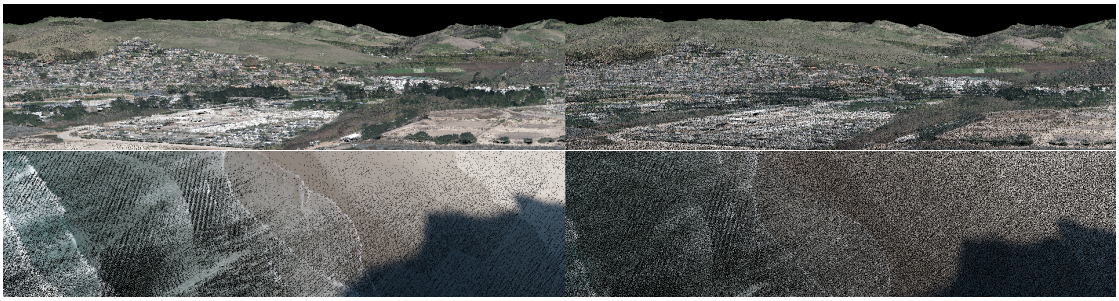

Figure 4.3: Rendering of a still (left) and a moving (right) camera with 1px per point.

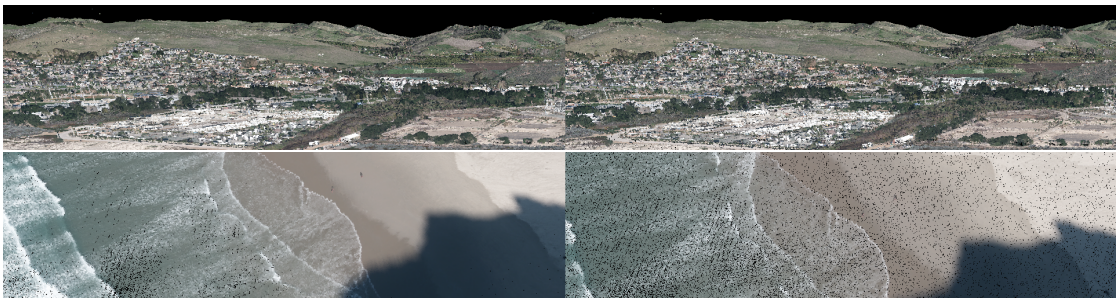

Figure 4.4: Rendering of a still (left) and a moving (right) camera with 2px per point.

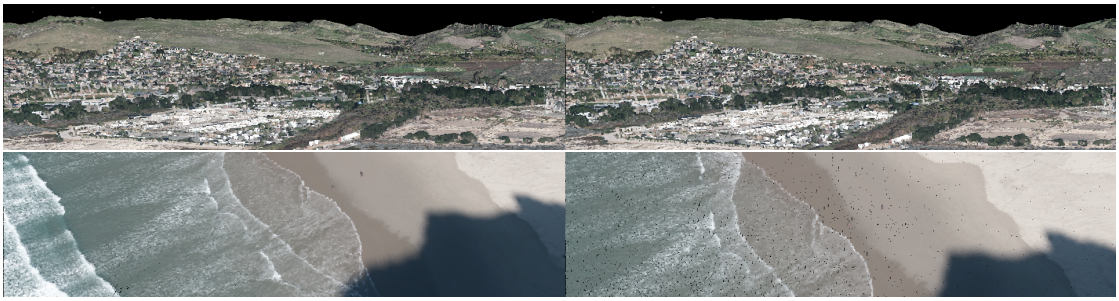

Figure 4.5: Rendering of a still (left) and a moving (right) camera with 4px per point.

With a point size of 1px, the noise during movement is significant, and the scene background becomes visible through the holes in the result. A 4px point size has nearly no artifacts during movement, but details begin to be less visible in the distance. 2px seems to be a good default selection for the renderer. However, these best point size depends on the point density and the distance to the viewer. The implementation of adaptive point sizes, as described by Schütz, could help with the mentioned artifacts [Sch16].

4.2.2 Visual Pattern of Shuffeling

The presented method of shuffling is not suitable for every pool size in our case. It appears that it does an excellent job of mixing the nodes but not necessarily the points

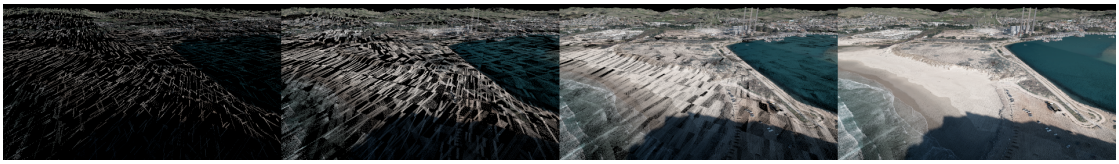

Figure 4.6: The rendering pattern with a pool size of 2^{25} (33.6M).

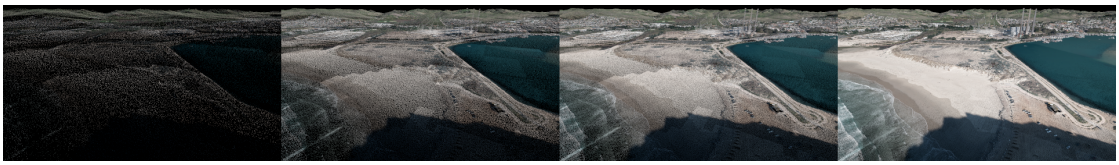

Figure 4.7: The rendering pattern with a pool size of 30 million.

in the nodes. The ordering of the points inside the nodes become visible with particular pool sizes. A pool size of a power of two seems to produce the worst results. Figure 4.6 shows the patterns for a pool size of a power of two, and Figure 4.7 depicts an appropriate pool size that results in a seemingly uniform distribution.

These patterns could be avoided entirely by using a different shuffling algorithm that is more random in nature or by shuffling the points in the nodes during the creation of the octree. However, the suggested shuffling works good enough for various pool sizes, is straightforward to implement, and allows fast testing with different pool sizes.

CHAPTER 5

Conclusion

The presented implementation in the Potree point cloud viewer is an experimental approach to combine the performance advantages of the progressive rendering technique with hierarchical level of detail by Ponto et al. [PTC17] with the visually appealing convergence of the progressive method by Schütz and Wimmer [SWMO19]. The results were achieved by continuously inserting nodes of the octree into a shuffled point pool that is managed by a memory manager. The last frame is reprojected into the new camera perspective, and additional details are accumulated by adding a subset of the point pool every frame. The resulting performance is on par with the original Potree renderer. However, due to the progressive rendering, it is possible to process fewer points every frame and still converge towards the same visual quality. This trait dramatically reduces the load on the GPU when viewing massive amounts of points. Additionally, the suggested implementation works with arbitrarily large point clouds and arbitrary GPU memory sizes.

The most significant bottleneck of the renderer is the simple memory manager and the insertion of new nodes into the pool. Especially on slower devices, this bottleneck becomes apparent and leads to a juddery experience. The manager has the potential for further optimizations, and it might be possible to reduce this bottleneck significantly. Therefore, the presented solution seems to be a feasible option if progressive rendering has to be combined with a hierarchical structure.

5.1 Limitations and Future Work

The bottleneck of the memory manager should be reduced. This problem is essentially an optimization task that should focus on minimizing the amount and frequency of data that is passed to the GPU every frame. A reasonably simple optimization could be to manage and insert nodes in bulk.

Our compute shader based point rendering approach did not achieve performance advantages over the traditional rendering pipeline based on glDrawArrays for each node. It also does not implement proper depth testing and may, therefore, present distant points over closer points. Future work consists of finding a compute shader based approach that includes correct depth testing and reevaluating the performance. Our experience within this thesis, however, hints at reduced efficiency of compute based point rendering for browsers as approaches based on atomicMin [SW19] can not be employed due to the missing support of 64-bit integers [MPM18, Gro16].

A better shuffling method is required that works equally well for all sizes and does not result in visual patterns during the rendering in some cases.

The implemented fix for the limited precision of 32-bit floating-point numbers should be revisited. There might be a solution that does not require to clear the complete point pool and therefore, can result in lags.

New nodes with a high point density that are inserted into the point pool appear rather suddenly. Further considerations may be taken to identify possible ways how such nodes can be blended into the view without drawing too much attention from the viewer. A similar problem was also mentioned and solved by Futterlieb et al.[FTB16], and maybe their solution also applies to the mentioned problem.

The holes that appear during camera movements should be reduced further for a better visual result. For example, filling small holes with morphological operations could increase the stability of the rendering.

The renderer should be reevaluated on mobile devices as soon as the WebGL 2.0 Compute API is available on mobile devices.

A re-implementation with the upcoming WebGPU API [ftWCG19] that is currently under development could be considered to test for improved compute performance.

List of Figures

List of Tables

Bibliography

[Bay77] Carter Bays. A comparison of next-fit, first-fit, and best-fit. *Communications of the ACM*, 20(3):191–192, 1977.

[Dev19] Chromium/Blink Developers. Intent to implement: Webgl 2.0 compute. `https://groups.google.com/a/chromium.org/d/msg/blink-dev/bPD47wqY-r8/5DzgvEwFBAAJ`, 2019. [Online; accessed 10-September-2019].

[EPBD18] Mohammed S Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. Toward low-latency and ultra-reliable virtual reality. *IEEE Network*, 32(2):78–84, 2018.

[Fac13] OpenTopography Facility. Pg&e diablo canyon power plant (dcpp): San simeon, ca central coast. `http://opentopo.sdsc.edu/lidarDataset?opentopoID=OTLAS.032013.26910.2`, 2013. [Online; accessed 29-September-2019].

[FTB16] Jörg Futterlieb, Christian Teutsch, and Dirk Berndt. Smooth visualization of large point clouds. *IADIS International Journal on Computer Science and Information Systems*, 11(2):146–158, 2016.

[ftWCG19] GPU for the Web Community Group. WebGPU editor's draft, 27 september 2019. `https://gpuweb.github.io/gpuweb/`, 2019. [Online; accessed 29-September-2019].

[Gro16] Khronos Group. The OpenGL ES® Shading Language language version: 3.10. `https://www.khronos.org/registry/OpenGL/specs/es/3.1/GLSL_ES_Specification_3.10.withchanges.pdf`, 2016. [Online; accessed 10-September-2019].

[Gro19] Khronos WebGL Working Group. WebGL 2.0 Compute editor's draft, 25 september 2019. `https://www.khronos.org/registry/webgl/specs/latest/2.0-compute/`, 2019. [Online; accessed 29-September-2019].

[HD62] Thomas E Hull and Alan R Dobell. Random number generators. *SIAM review*, 4(3):230–254, 1962.

[Knu14] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[LR85] Michael Luby and Charles Rackoff. How to construct pseudo-random permutations from pseudo-random functions. In *Advances in Cryptology–CRYPTO*, volume 85, page 447, 1985.

[MPM18] Robin Morisset, Filip Pizlo, and Myles C. Maxfield. Wsl specification. `https://gpuweb.github.io/WSL/`, 2018. [Online; accessed 13-October-2019].

[PTC17] Kevin Ponto, Ross Tredinnick, and Gail Casper. Simulating the experience of home environments. In *2017 International Conference on Virtual Rehabilitation (ICVR)*, pages 1–9. IEEE, 2017.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352. ACM Press/Addison-Wesley Publishing Co., 2000.

[Sch16] Markus Schütz. Potree: Rendering large point clouds in web browsers. *Technische Universität Wien, Wiedeń*, 2016.

[Sho75] John E Shore. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Communications of the ACM*, 18(8):433–440, 1975.

[Sto71] Harold S Stone. Parallel processing with the perfect shuffle. *IEEE transactions on computers*, 100(2):153–161, 1971.

[SW19] Markus Schütz and Michael Wimmer. Rendering point clouds with compute shaders, 2019.

[SWMO19] Markus Schütz, Michael Wimmer, Gottfried Mandlburger, and Johannes Otepka. Progressive real-time rendering of one billion points without hierarchical acceleration structures. Technical report, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, October 2019. human contact: technicalreport@cg.tuwien.ac.at.

[Tho07] Chris Thorne. *Origin-centric techniques for optimising scalability and the fidelity of motion, interaction and rendering*. University of Western Australia, 2007.