FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Rendering Large Point Clouds in Unity

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Simon Maximilian Fraiss
Matrikelnummer 01425602

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, 14. September 2017

Simon Maximilian Fraiss   Michael Wimmer

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Rendering Large Point Clouds in Unity

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Simon Maximilian Fraiss
Registration Number 01425602

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, 14th September, 2017

Simon Maximilian Fraiss   Michael Wimmer

Erklärung zur Verfassung der Arbeit

Simon Maximilian Fraiss
Alser Straße 26, 1090 Wien

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, 14. September 2017

Simon Maximilian Fraiss

Danksagung

Mein Dank gilt Markus Schütz, welcher mich bei dem Verfassen dieser Arbeit unterstützt hat und jederzeit bei Fragen zur Verfügung stand.

Darüber hinaus möchte ich meinen Unifreunden danken, welche das Studium zu einem unterhaltsamen Erlebnis gemacht haben. Letztendlich danke ich auch meinen Eltern für ihre kontinuierliche Unterstützung.

Acknowledgements

I would like to thank Markus Schütz, who supported me at writing the thesis and was always available for questions.

Furthermore, I would like to thank my friends from university, who made studying a fun experience. Lastly, I also thank my parents for their continuous support.

Kurzfassung

In dieser Arbeit wurde ein Punktwolken-Renderer in der Spiel-Engine Unity entwickelt. Der Fokus liegt hierbei speziell auf sehr großen Punktwolken mit mehreren Millionen oder Millarden Punkten, welche nicht einfach komplett in den Speicher geladen und dargestellt werden können. Spezielle Datenstrukturen und Algorithmen müssen genutzt werden, um jederzeit nur die Teile der Punkte, die für die aktuelle Kameraposition relevant sind, zu laden und zu rendern. Das Ergebnis ist ein effizientes Rendering-System mit variablen Einstellungen und verschiedenen Rendering-Methoden. Das Projekt ist auf GitHub verfügbar.

Abstract

In this thesis, a point-cloud renderer was developed in the game engine Unity. The focus lies especially on very big point clouds with several millions or billions of points, which cannot be loaded completely into memory. Special data structures and algorithms are needed to load and render continuously only the parts of the point-cloud that are relevant for the current camera position. The result is an efficient rendering system with variable settings and various rendering methods. The project is available on GitHub.

Contents

CHAPTER 1

Introduction

1.1 Motivation

A point cloud is a set of independent spatial points, which is used to represent a three-dimensional object. Point clouds are often generated by 3D-scanning techniques like laser scanning or photogrammetry. They are a very simple data structure that can be transformed or combined with other point clouds very easily. Rendering can be done by simply projecting every point to a pixel on the screen. However, if the points are not dense enough or the camera is too close, holes can be seen in the object. Therefore, various techniques are often used to transform a point cloud into a polygon mesh. A disadvantage of point clouds is their enormous memory consumption. To model large complex objects, many points are needed.

This thesis focuses on the real-time rendering of very large point clouds with several millions or even billions of points. Examples of such point clouds are large 3D scans of topographical data, such as the ones provided by OpenTopography [ope]. Some of their point clouds consist of several hundreds of billions of points. Creating meshes is not always an option, because it can take very long, it might not work well when the point cloud has a low point density, and often users just want to see the raw scanned data without any post processing done. In case of such point clouds, real-time rendering is more complex, because the data cannot be loaded into memory completely. Efficient out-of-core algorithms and spatial data structures are needed to render only as many points as needed for an acceptable graphical representation.

Unity is a popular game engine. Graphical applications, such as games, can be programmed fairly easily using it. It also provides building options for a lot of different operating systems. Games can be exported to PC, Mac, Linux, Android, iOS, WebGL, XBox One, PlayStation 4 and others. Virtual Reality applications can also be easily created with Unity. This makes it a very attractive option for creating graphical applications. Unity does not provide a built-in solution for rendering point clouds. Plugins for displaying point clouds are available (for example the Point Cloud Free Viewer [pcv]). However, they are usually very simple and not suited for rendering gigantic point clouds.

1.2 Goal of the Thesis

The goal of this thesis is to develop a point cloud renderer in Unity for viewing large point clouds in real-time.

The point clouds are stored in the Potree file format (see Chapter 2 and [Sch16]). It is possible to navigate the camera through the scene. At any time, only the points of the nodes that are currently inside the view frustum and whose bounding boxes exceed a certain minimum screen size, are loaded into the memory. Also, a point budget is given, which is the maximum number of points being displayed. An LRU cache is used to keep track and remove the least recently used nodes from memory. The rendering of the points can either be done by projecting each point to a single pixel or by displaying each point as a screen faced square or circle with a given size in screen or world units. Additionally, a point-interpolation-mode is implemented by rendering the nodes as screen-faced cones or paraboloids.

Figure 1.1 shows an example of a rendered image. The final Unity project is available on Github [Fra].

1.3 Structure of the Work

Chapter 2 gives an introduction to Unity and an overview of related work about point cloud rendering, on which this work is building on. Chapter 3 describes the algorithms and the implementation. Chapter 4 examines the performance of the renderer and compares different settings. Chapter 5 concludes the thesis and reflects on possible future work.

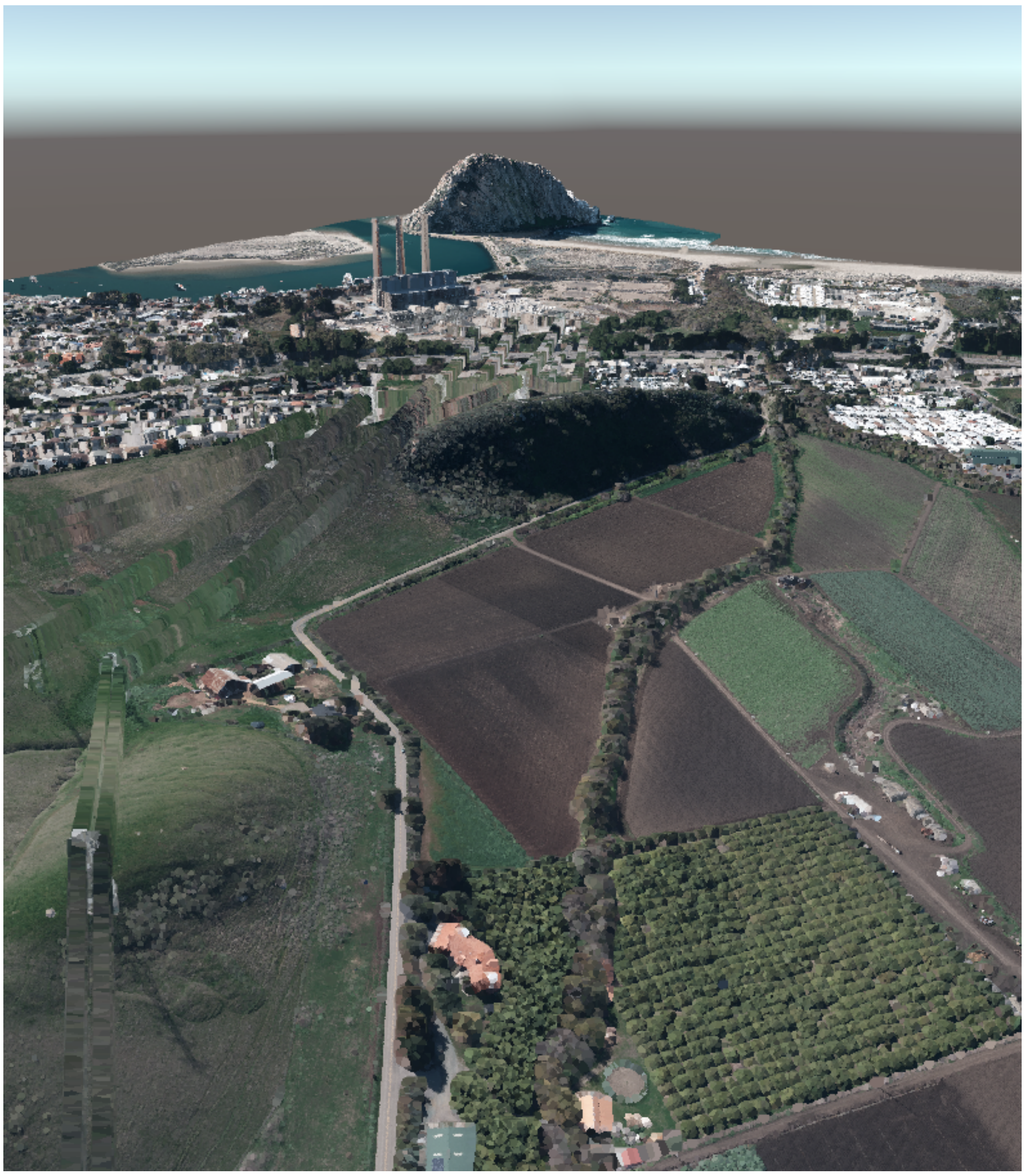


Figure 1.1: A part of the "San Simeon, CA Central Coast" point cloud from OpenTopography [san] rendered with the implemented system

CHAPTER 2

Background and Related Work

2.1 Introduction to Unity

The most important element in Unity is the scene graph, which is a tree structure that contains so called *Game Objects*. A Game Object may consist of several *components*. Each Game Object has a *transform*-component, giving it a position in 3D-space. Additionally, other components can be added, such as lights, cameras, rigid bodies for physics simulations or meshes. Scripts can also be attached as a component. Scripts can be written either in C# or in JavaScript. For this thesis, C# was used. When creating a new script, a new C# class is created, that extends the class *MonoBehaviour*. A start-method can be implemented, which is getting called when the script starts (this happens as soon as the Game Object, to which the script is attached, appears in the scene), as well as an update-method, which gets called every frame. Game Objects can be disabled so that they still exist in the scene graph, but they are not updated, nor do they have any effect on the scene (for example, attached meshes are not displayed).

One important component is the *Mesh Filter*. With a Mesh Filter, a polygon- or point-mesh can be defined. A mesh consists of vertices with their corresponding colors, texture coordinates, and normals. These can be assembled to polygons by providing indices and a mesh topology (Unity provides the topologies *Triangles*, *Quads*, *Lines*, *LineStrip* and *Points*). A mesh can consist of up to 65,000 vertices.

A Mesh Filter alone does not yet display a mesh. A *Mesh Renderer* is needed for that as well. In a Mesh Renderer, the options for rendering can be changed, such as shadow-settings or the material used for rendering. Materials are derived from shaders, which are written in a variant of HLSL called Cg.

2.2 Rendering Large Point Clouds

Levoy and Whitted [LW85] were the first ones to consider points as rendering primitives for solid objects. Points have been used for intangible objects like fire or smoke before, but not for continuous three-dimensional surfaces.

While small point clouds up to a few million points can be rendered in real-time on current GPUs, spatial data structures and out-of-core algorithms are required in order to render large point clouds with hundreds of millions of points in real-time. Several tree-based data structures have been proposed. For real-time rendering of gigantic point clouds, it is important that the data structure supports levels of detail (LOD) so that the point cloud can be rendered with a different amount of details, depending on the distance to the camera.

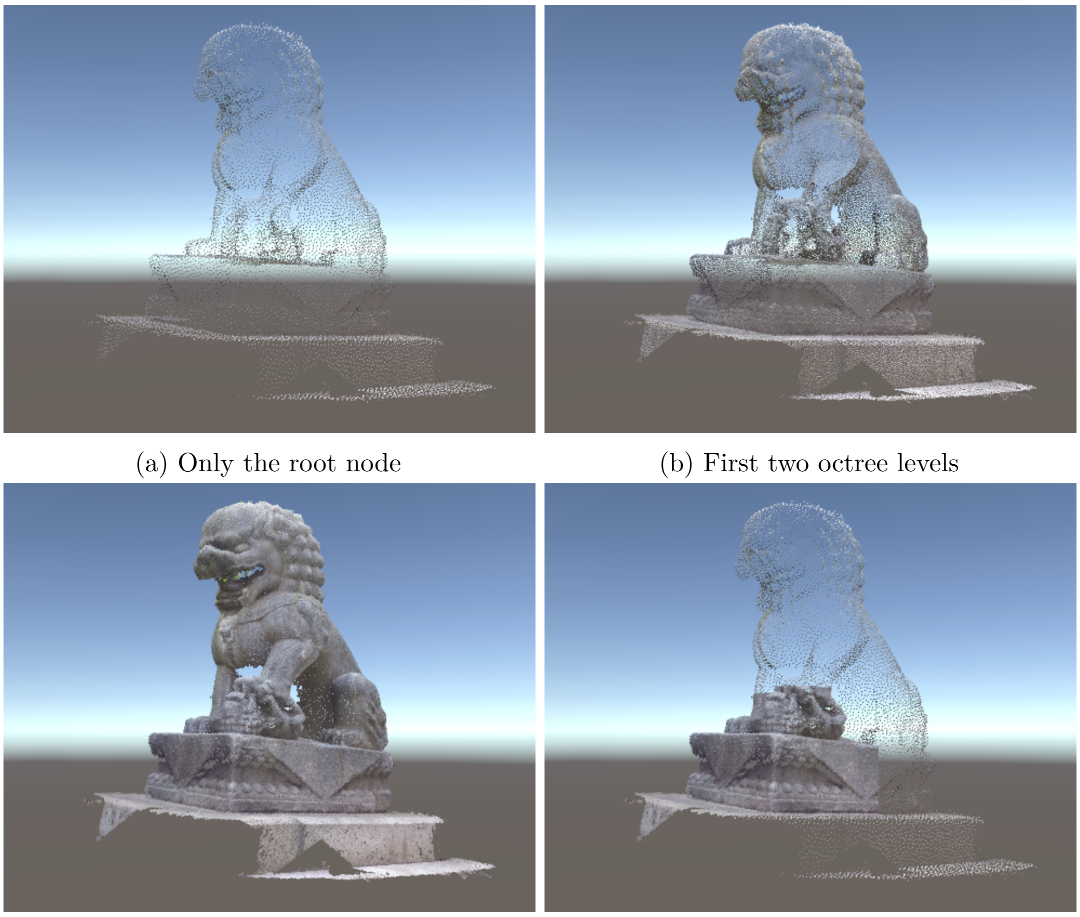
One of the first data structures for rendering large point clouds was *QSplat* by Rusinkiewicz and Levoy [RL00]. It uses a bounding sphere hierarchy, in which the leaf nodes contain the data points, while the inner nodes contain averages of their children, in order to represent the children on a lower level of detail. During rendering, the nodes are traversed until the desired level of detail is reached, for example, until the projected bounding sphere of a node is smaller than a predefined maximum splat size. If a leaf node is reached, a splat is drawn. If the node is not a leaf, but the size of its projected bounding sphere is small enough, a splat that represents the node and its descendants is rendered and the traversal of this sub-tree stops.

Gobbetti and Marton [GM04] proposed a GPU-friendlier rendering system called *Layered Point Clouds* (LPC). The point cloud is stored in a binary tree. Each node contains a subset of the point cloud. By rendering just the points in the first levels of the hierarchy, a coarse approximation can be rendered. Unlike QSplat, which computes averages, no new points have to be created for this data structure.

Wimmer and Scheiblauer [WS06] introduced the so-called *nested octree* as a data structure for point clouds. The nested octree is an octree whose nodes contain subsamples of the points inside the bounding box represented by the octree node, similar to LPC. The points in every node are stored in an inner octree, where the root node of the inner octree has the same bounding box as the corresponding node in the outer octree. The more levels are rendered, the more detailed the point cloud becomes. Each inner octree is stored in its own file. Rendering such a tree can be done by loading the outer octree into main memory and traversing it each frame. Each visited node gets stored inside a priority queue, where the priority is the projected size of the bounding box of the node. For each node, it is checked whether it fulfills the rendering conditions, i.e. if it is inside the view frustum, if its projected size exceeds a certain minimum size, and if the accumulated number of points remains below the point budget. If the node fulfills the conditions, it is either put into a render queue if the inner octree is already loaded or it is scheduled to be loaded.

In his Ph.D. thesis, Scheiblauer [Sch14] proposed an adaption of the nested octree which is more suitable for editing points. The so called *modifiable nested octree* (MNO) does not store the points at each node in an inner octree, but instead in a regular grid. Rendering works similarly as with the classical nested octree.

Schütz [Sch16] developed a WebGL based rendering system for gigantic point clouds called *Potree*. It uses a similar octree structure as the MNO. The creation of the tree is slightly different, as a different subsampling approach is used. For Potree, a converter was developed, which transforms a point cloud from classical formats, such as laz or las, to the Potree octree structure. Potree as well as the Potree Converter are available online [Sch].



(a) Only the root node

(b) First two octree levels

(c) First three octree levels

(d) Mixed LODs

Figure 2.1: A point cloud in the Potree format rendered with different levels of detail

CHAPTER 3

Implementation

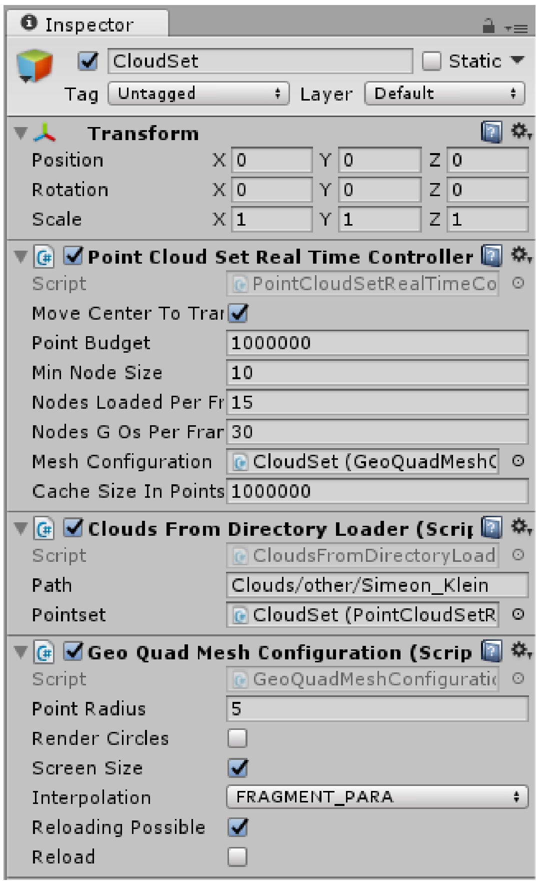
This chapter describes the implementation and the used algorithms of the Unity point cloud renderer. It is split into three parts: The first part gives an overview over the implemented system. The second part describes loading the files, handling the octree nodes, determining node visibility and the multithreaded loading system. The third part explains the various rendering techniques for the points in detail.

3.1 Overview

The implemented rendering system is not a stand-alone application but provides a framework for other applications to build upon. There is no specialized user interface to select the rendered point cloud or to change the settings. All these options are defined by Unity components and can, therefore, be changed inside the Unity editor or by use of code. Figure 3.1 shows an example for a Unity Game Object with several script-components attached to it, defining the rendering options and the path to the point cloud.

It is possible to define a *point cloud set*, which controls the loading settings, like point budget, minimum node size, cache size etc. These settings are described in section 3.2. Such a point cloud set can be seen in Figure 3.1 as the component *Point Cloud Set Real Time Controller*. There are *loader* components for loading a single point cloud, as well as for loading several point clouds at once. For example, a *Clouds From Directory Loader* can be used to load all point clouds in a specified path (this can also be seen in said Figure). A point cloud set has to be assigned to the loader so that the loader can pass the loaded cloud to the point set, which then starts the actual rendering process.

The separation of loader and point set enables including several point clouds in the scene, while the options are defined for all clouds at once. This makes it possible to define one shared point budget for all point clouds instead of having own budgets for each point cloud.



Inspector

✓ CloudSet ☐ Static

Tag Untagged Layer Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

✓ Point Cloud Set Real Time Controller

Script	PointCloudSetRealTimeCo
Move Center To Trar	✓
Point Budget	1000000
Min Node Size	10
Nodes Loaded Per Fr	15
Nodes G Os Per Frar	30
Mesh Configuration	CloudSet (GeoQuadMeshC
Cache Size In Points	1000000

✓ Clouds From Directory Loader (Scrip

Script	CloudsFromDirectoryLoad
Path	Clouds/other/Simeon_Klein
Pointset	CloudSet (PointCloudSetR

✓ Geo Quad Mesh Configuration (Scrip

Script	GeoQuadMeshConfigurati
Point Radius	5
Render Circles	☐
Screen Size	✓
Interpolation	FRAGMENT_PARA
Reloading Possible	✓
Reload	☐

Figure 3.1: An example for a Unity Game Object defining the rendering options

Additionally, further rendering options are set by defining a *mesh configuration*, which decides upon which rendering technique (see section 3.3) is used. In the example, geometry shader quad rendering is used with a point radius of five pixels, square splats, and paraboloid-interpolation.

3.2 Multithreaded Point Loading

3.2.1 Loading the Data

The point clouds are stored in the Potree file format (see section 2) and have been created with the Potree converter. Therefore, the points are stored in an octree data structure, where every node in the octree contains a subset of points. The hierarchy is stored in different files than the points themselves.

Even though the file format supports other point attributes, for simplicity the implemented loader assumes only the position and the colors are stored per point and each point is therefore represented by 16 bytes. Other attributes are not supported yet.

One important thing to note is that Unity stores the components of vectors as floats. As points might have very high values as positions, there might be a loss of precision. Where possible, values are stored as double values as long as possible, but all values have to be converted to floats before passing them on to the GPU. The implemented system provides the option to move the point cloud to the origin, which results in a smaller loss of precision.

At the beginning of the program, the hierarchy is loaded from the hierarchy files. Each node of the octree is represented as an object of a class *Node* in our application. The hierarchy files contain the number of points for each node as well. Due to a bug in the converter, these numbers are wrong and are therefore ignored. The real point counts can be deducted from the size of the node files and the number of bytes for each point. The correct point count for each node is determined the first time the points from the node are loaded into memory. The loading of the hierarchy is done in a parallel thread, called in one of the Unity start functions. This is done so that other objects in the scene can already be rendered while the point cloud hierarchy is being loaded. As soon as the hierarchy is loaded, the actual point cloud rendering process starts.

3.2.2 Basic Multithreaded Loading Concept

Rendering objects in Unity is done by creating Game Objects for the objects to render. For each node that should be visible, one or several Game Objects have to be created with a fitting Mesh Filter and a Mesh Renderer. For each octree node that we want to display, we use one Game Object. Only in case the node consists of more than 65,000 vertices, several Game Objects are needed.

The loading process uses three threads: The main thread of Unity, a traversal thread, and a loading thread. In the main thread, the visible Game Objects are updated once per frame if necessary changes have been detected in the traversal thread. Game Objects are created for octree nodes that should be visible and do not have Game Objects yet, and nodes that should not be visible anymore have their Game Objects deleted. Determining which nodes Game Objects have to be created or deleted is the job of the traversal thread. The loading thread is used for loading the point data from the files. Each thread is described further in the upcoming sections.

3.2.3 Main Thread

In the Unity main thread, the update-function of the renderer is called once per frame. In this function, the renderer informs the traversal thread of the current camera data (position, view frustum etc.), because this data can only be acquired in the Unity main thread. Then it checks its *toDelete*-queue. This queue has been filled by the traversal thread and contains all nodes whose Game Objects should be deleted. The main thread deletes those Unity objects but keeps the points in memory in the case that the node becomes visible again. Then the main thread checks its *toRender*-queue, which has also been filled by the traversal thread, and creates Game Objects for every node in there. At the end, the traversal thread is notified that the updating of the objects is done, so it can continue with a new octree traversal.

3.2.4 Traversal Thread

The traversal thread is responsible for determining which nodes should be visible. It traverses the octree structure and fills the *toDelete*-queue and the *toRender*-queue, which is then passed to the main thread.

First, new empty queues are created. Then the actual traversal is done. For every root node (there might be several root nodes if more than one point cloud is in the scene) it is checked, whether its projected size is higher or equal to the minimum projected size. If this is the case, the node is inserted into a priority queue, where the priority is a combination of the projected size and the centrality on the screen (see section 3.2.5). If this is not the case, the node should not be visible, so if it already has Game Objects, the node and its visible children are inserted into the *toDelete*-Queue.

Afterwards, a rendering point count variable, which will be used to count how many points will be rendered, is initialized with zero and the priority queue is iterated in a loop. In each loop pass, the node with the highest priority is removed from the queue and the following possibilities are checked:

1. The node is outside of the view frustum: If the node has Game Objects, the node and its visible children are inserted into the *toDelete*-queue.

2. The point count of the node is not known yet: Schedule the node for loading by the loading thread. It will not be rendered this frame.

3. The point count is known and adding it to the rendering point count does not exceed the point budget:

    a) The node has Game Objects: Increase the rendering point count by the number of points in this node. The node does not have to be put into the *toRender*-queue, because it is already visible.

    b) The node does not have Game Objects, but the points of this node are in the memory: Increase the rendering point count and put the node into the *toRender*-queue.

    c) The node has neither Game Objects nor stored point data: Schedule it for loading by the loading thread. The node will not be rendered this frame.

4. The point count is known and adding it to the rendering point count would exceed the point budget: If they have Game Objects, this node and its children are put into the *toDelete*-queue. The queue iteration stops.

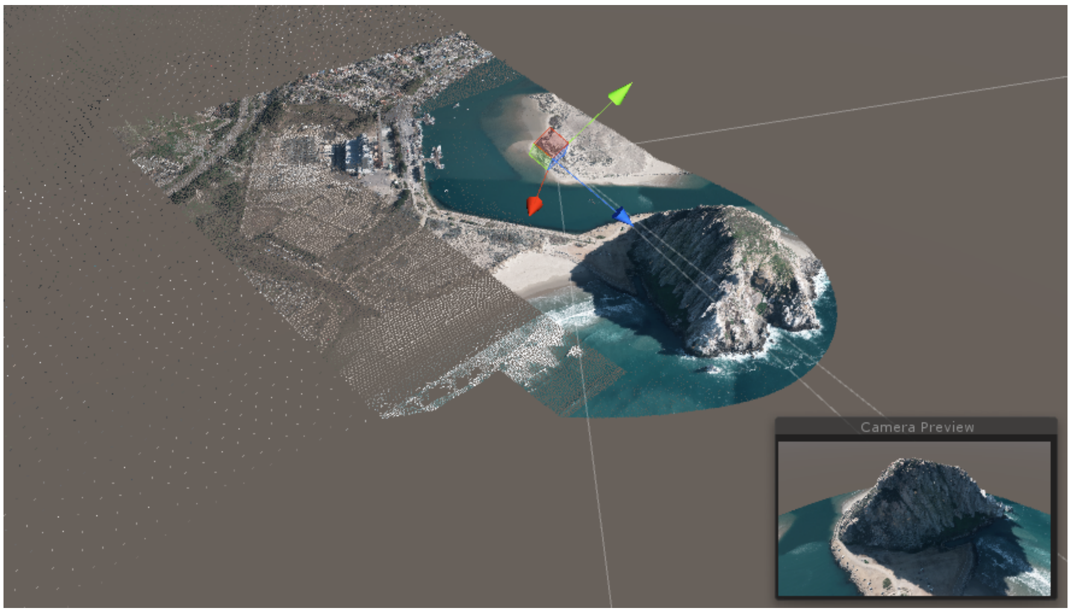


Figure 3.2: Screenshot from the scene view in Unity. Objects in front of the camera are rendered in more detail than objects only partially in the view frustum

After this check, a loop iterates over the children of the node. If a child's projected size is higher or equal to the minimum projected size, it is inserted into the priority queue as well. If it is below the minimum projected size and it has Game Objects, it and its visible children are inserted into the *toDelete*-queue.

The number of nodes to be scheduled for loading per traversal is limited, as is the number of scheduled Game Object creations. That way, the loading of nodes that might not be used in the next frame gets reduced. The number of Game Objects being created is limited, because the creation of Game Objects can be a costly operation.

The priority queue iteration stops, when the point budget is reached, the maximum number of scheduled loadings is reached, the maximum number of scheduled Game Object creations is reached or the queue is empty.

After this iteration, it is checked if there are still nodes that have Game Objects, even though they should not be visible. These nodes were not checked during the iteration loop, because the exceedance of the point budget stopped the loop before checking them. These nodes are put into the *toDelete*-queue as well. This is realized by having lists of the visible nodes for the current and the last traversal. If a node is determined to be visible it is inserted into the new list and removed from the old one. Then, only the remaining nodes of the old list have to be put into the *toDelete*-queue.

Originally, the traversal was done in the main thread. However, this sometimes led to a low number of frames rendered per second when using big point clouds. So it was moved

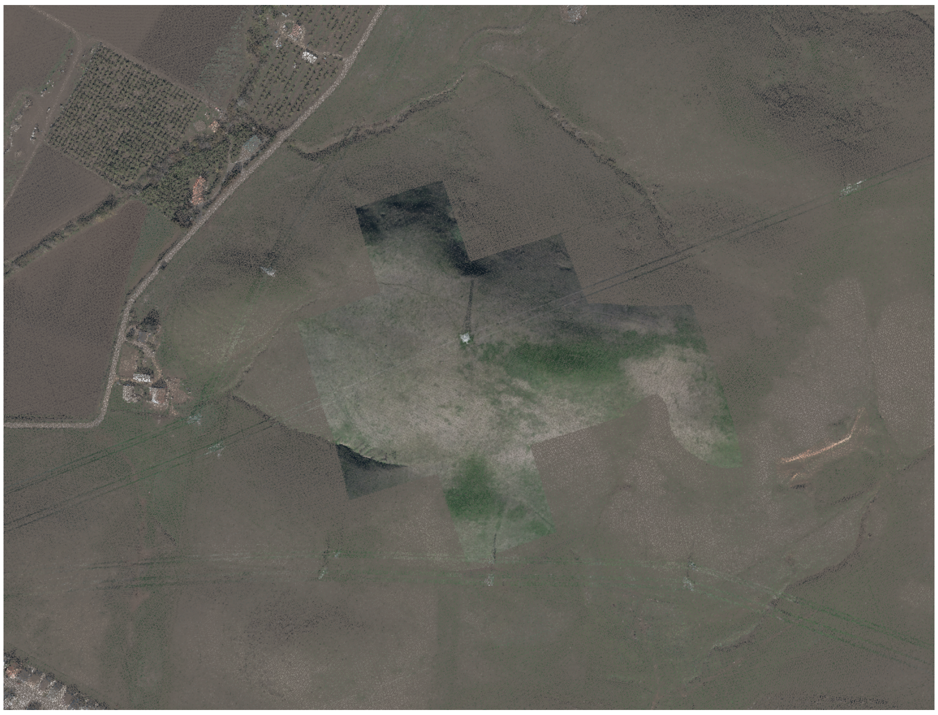


Figure 3.3: Areas in the center of the screen are loaded in more detail.

into an extra thread to improve the user experience. That way, several frames may be rendered at a lower level of detail until the traversal is finished.

3.2.5 Traversal Priority

The priority used for the priority queue in the traversal thread should reflect the perception of the user. Nodes with a bigger projected size will mostly have a larger impact on the rendered picture than nodes with a smaller projected size. Also, as the awareness of the user is mostly focused on the center of the screen, nodes in the center should have a higher priority than nodes of the same size at the edge of the screen. The priority value is determined like shown in the following equations:

$$slope = \tan(\frac{fov}{2}) \tag{3.1}$$

$$projectedSize = \frac{screenHeight}{2} * \frac{radius}{slope * distance} \tag{3.2}$$

$$angle = \arccos(\mathbf{camToScreenCenter} \cdot \mathbf{camToNodeCenter}) \tag{3.3}$$

$$priority = \frac{projectedSize}{|angle| + 1} \tag{3.4}$$

Equation 3.2 shows how to calculate the projected size of a node. *Radius* refers to the radius of the bounding sphere of the node. *Distance* is the distance from the camera to the center of the nodes bounding box. Equation 3.3 shows how to calculate the angle. The two vectors are the forward vector of the camera and the normalized direction from the camera to the node center. Equation 3.4 shows how the overall priority is calculated. One is added to the absolute value of the angle, to prevent infinitely large priority values as well as divisions by zero.

The priority queue is implemented using a max heap.

3.2.6 The Loading Thread

The loading thread possesses a loading queue. Each time the traversal thread finds a node that has to be loaded, it inserts it into this queue. The loading thread continuously removes the first node from the queue and loads the points for this node (if they haven't already been loaded in the mean time since inserting it into the queue).

3.2.7 LRU Cache

A Least Recently Used Cache was implemented in order to keep the memory usage below a certain threshold. The cache has a maximum number of points it can store and if this threshold is breached, the points of the least recently used node are removed from memory. When a node is loaded in the loading thread it gets inserted into the cache. When Game Objects are created for a node, i.e. while nodes are actively used and being rendered, the node is removed from the cache. When the Game Objects are deleted, it is once again inserted. If the number of points inside the cache exceeds the maximum number, the cache removes the points of the nodes which have been inserted least recently from memory and also removes the nodes from the cache.

The cache is implemented using a linked list for the queuing behavior, as well as a dictionary mapping from the tree nodes to the list nodes, in order to enable efficient removal from the queue.

3.3 Rendering Points

Different rendering techniques for points have been implemented. No illumination models were implemented because the test data sets do not contain normals and because the test data sets were colored by photos that give the model a form of static illumination.

3.3.1 Single-Pixel Point Rendering

The simplest method for rendering points is the points primitive, where every vertex given to the GPU is rendered as a single pixel on the screen. In the vertex shader, each vertex is transformed into screen space coordinates. The fragment shader just returns the color of the vertex.

Figure 3.4 demonstrates rendering with this method. While parts in the distance look dense, objects near the camera, like the hill on the right side of the image, seem almost transparent due to the low point density.

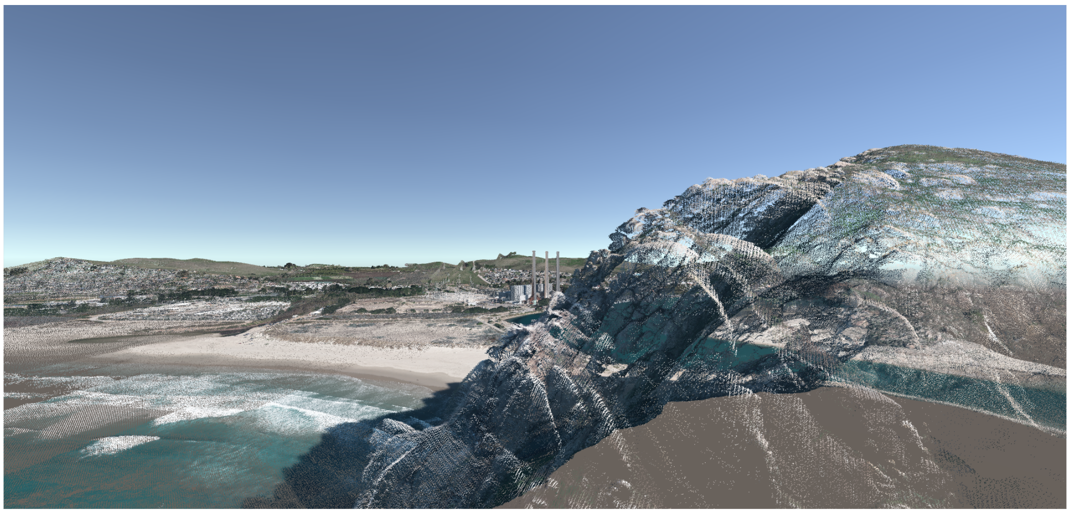


Figure 3.4: Point cloud rendered with single-pixel point rendering and a budget of 5,000,000 points

Many graphic libraries, like OpenGL, also support setting a point size, so that each point automatically gets drawn as a square of a given size on the screen. However, this is not possible in Unity. One reason is that this operation is not supported by newer DirectX-versions and one would have to limit the platforms on which the program could be executed. Thus, to render the points as screen faced squares or circles, other ways had to be explored.

3.3.2 4-Vertex Quad Rendering

A simple method for rendering points as screen faced squares is to use the quads primitive and pass each point four times to the GPU. Additionally, to each of the four points a different offset vector is given. This vector states the direction in which to move the vertex on the screen in order to create a square. The four points have the vectors $\begin{pmatrix}-1 & 1\end{pmatrix}^T$, $\begin{pmatrix}1 & 1\end{pmatrix}^T$, $\begin{pmatrix}1 & -1\end{pmatrix}^T$ and $\begin{pmatrix}-1 & -1\end{pmatrix}^T$.

In the vertex shader, the position of the point is transformed to screen coordinates. Then the offset vector, times the desired point size in pixels, divided by the screen size is added to the position. The code can be seen in listing 3.1. The offset vector is passed via the uv-coordinates. It gets multiplied with the 4th vertex component, which will be undone later during homogenization.

If simple squares are rendered, the fragment shader just returns the color. If circles are desired, each fragment at which the distance to the center is larger than one is discarded

(See listing 3.2).

```
VertexOutput vert(VertexInput v) {
  VertexOutput o;
  o.position = UnityObjectToClipPos(v.position);
  o.position.x += v.uv.x * o.position.w * _PointSize / _ScreenWidth;
  o.position.y += v.uv.y * o.position.w * _PointSize / _ScreenHeight;
  o.color = v.color;
  o.uv = v.uv;
  return o;
}
```

Listing 3.1: Vertex Shader for 4-Vertex Quad Rendering

```
float4 frag(VertexOutput o) : COLOR {
  if (_Circles >= 0.5 && (o.uv.x*o.uv.x + o.uv.y*o.uv.y) > 1) {
    discard;
  }
  return o.color;
}
```

Listing 3.2: Fragment Shader for 4-Vertex Quad Rendering. _Circles is an integer variable used as a boolean-replacement.

One problem with this method is that more meshes may be needed. In Unity, a mesh can contain up to 65,000 vertices. As we store every vertex four times, we can effectively only store up to 16,250 points in a mesh. That means that one Game Object might not be enough for one octree node and that we need multiple Game Objects to represent a single point cloud octree node.

Figure 3.5 shows the results. The hill does not seem as transparent as in the single-pixel approach. In areas with a lot of points - like the ocean in the bottom left corner - the circles or squares can hardly be recognized as such. However, in regions with low point density - for example on the hill in the front or the details in the background - they are still clearly noticeable.

3.3.3 Geometry Shader Quad Rendering

Another way to render screen faced squares is by using a geometry shader. Each vertex gets sent to the GPU once. In the vertex shader, the position is transformed to screen space. In the geometry shader, for each vertex, four new vertices are build to form a square with the desired size on the screen (see listing 3.3). The fragment shader is the same as described in the section above.

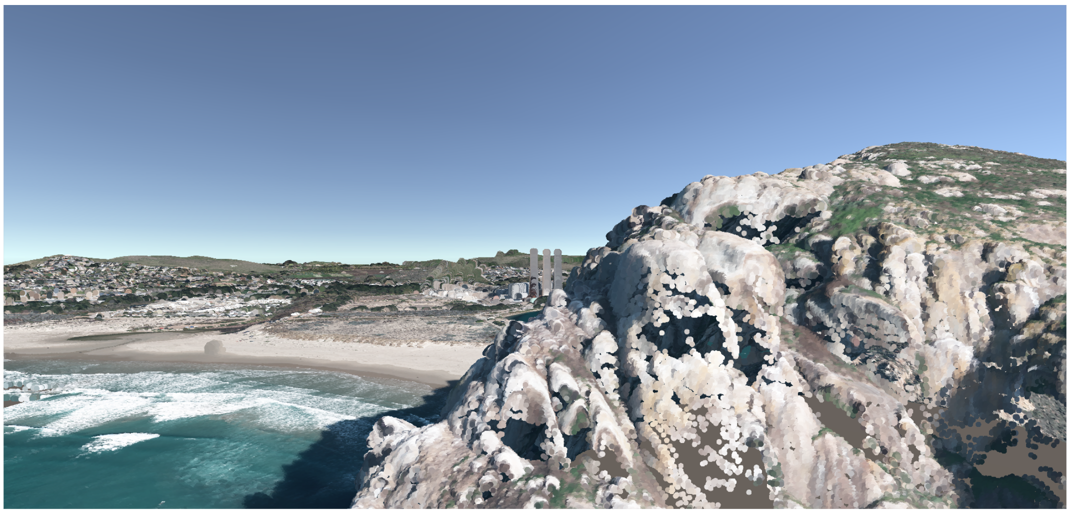


Figure 3.5: Points rendered as circles with a radius in pixels

```
[maxvertexcount(4)]
void geom(point VertexMiddle input[1], inout TriangleStream<VertexOutput>
    outputStream) {
  float xsize = _PointSize / _ScreenWidth;
  float ysize = _PointSize / _ScreenHeight;
  VertexOutput out1;
  out1.position = input[0].position;
  out1.color = input[0].color;
  out1.uv = float2(-1.0f, 1.0f);
  out1.position.x -= out1.position.w * xsize;
  out1.position.y += out1.position.w * ysize;

  //out2, out3 and out4 are calculated similarly, but with other +-
    combinations

  outputStream.Append(out1);
  outputStream.Append(out2);
  outputStream.Append(out4);
  outputStream.Append(out3);
}
```

Listing 3.3: Part of the Geometry Shader for screen faced splats given the desired size in pixels.

Screen faced squares can also be rendered with a given world unit size instead of a pixel size. In this case, an up- and a sideways-vector are calculated which describe the directions of the screen facing square (see listing 3.4). The transformation to screen coordinates happens in the geometry shader after the creation of the four vertices. The four vertices are created by adding or subtracting the up- and sideways-vectors (see listing 3.5).

```
VertexMiddle vert(VertexInput v) {
  VertexMiddle o;
  o.position = v.position;
  o.color = v.color;
  float3 view = normalize(UNITY_MATRIX_IT_MV[2].xyz);
  float3 upvec = normalize(UNITY_MATRIX_IT_MV[1].xyz);
  float3 R = normalize(cross(view, upvec));
  o.U = float4(upvec * _PointSize, 0);
  o.R = float4(R * _PointSize, 0);
  return o;
}
```

Listing 3.4: The vertex shader for screen faced splats for a given world-space point size

```
[maxvertexcount(4)]
void geom(point VertexMiddle input[1], inout TriangleStream<VertexOutput>
    outputStream) {
  VertexOutput out1;
  out1.position = input[0].position;
  out1.color = input[0].color;
  out1.uv = float2(-1.0f, 1.0f);
  out1.position += (-input[0].R + input[0].U);
  out1.position = UnityObjectToClipPos(out1.position);

  //out2, out3 and out4 are calculated similarly, but with other +-
    combinations

  outputStream.Append(out1);
  outputStream.Append(out2);
  outputStream.Append(out4);
  outputStream.Append(out3);
}
```

Listing 3.5: Part of the Geometry Shader for screen faced splats for a given world-space point size

Figures 3.6 and 3.7 demonstrate the rendering with a given point radius in world units. Points that are closer to the camera have a larger pixel size than more distant points. By choosing a bigger size, holes in the front can be covered without creating annoyingly prominent points in the background. However, the used shapes are very recognizable near the camera.

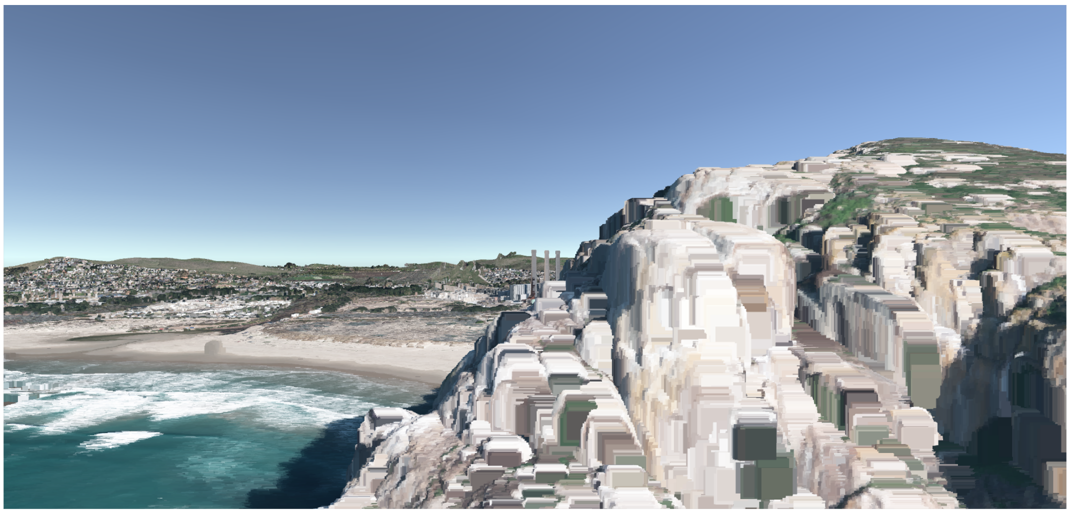


Figure 3.6: Points rendered as squares with a world size radius

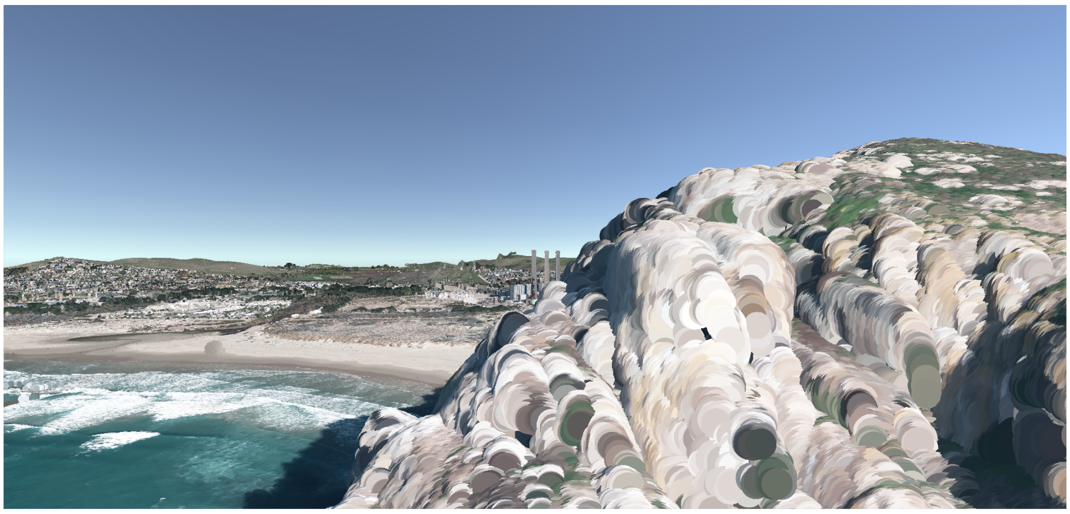


Figure 3.7: Points rendered as circles with a world size radius

3.3.4 Interpolation

The previously shown methods have the disadvantage that the points occlude each other, so that many visual details are lost. Different methods exist to increase the quality of point rendering. Near points can be blended together in order to create a more realistic result image, where the squares or circles themselves are not that prominent anymore and details and structures are preserved better[BHZK05]. Schütz and Wimmer [SW15] developed a method which creates a nearest-neighbor-like interpolation of the points. This is done by rendering the points as screen faced 3d shapes, such as cones, spheres or paraboloids, instead of simple squares. In this work, interpolations with cones and paraboloids were implemented. These were implemented in two different ways. Both variants are an extension of the geometry shader approach described above.

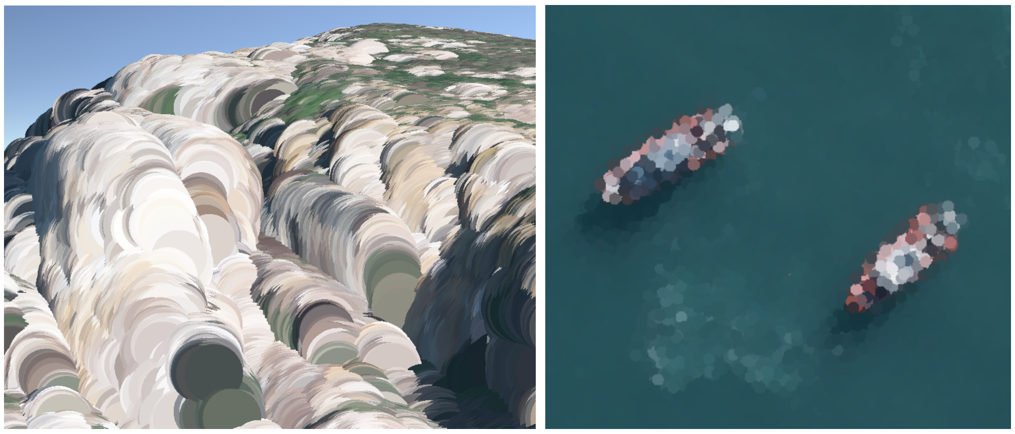
Shapes in the Fragment Shader

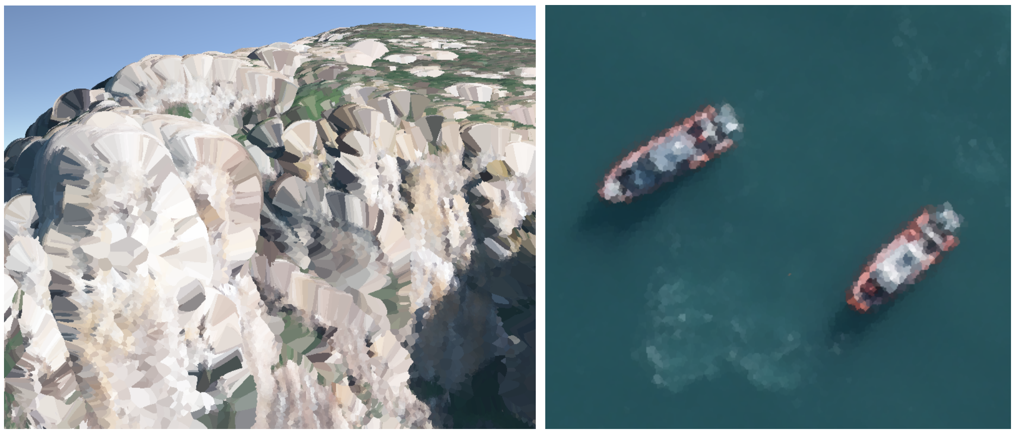
One way to create cones or paraboloids is to adjust the depth value of the screen aligned squares inside the fragment shader. To do so, the position of each vertex in view space is calculated in the geometry shader. In the fragment shader, the z value of the view position is adapted to create the shape. The resulting position is then multiplied by the projection matrix to get the correct depth value. Listing 3.6 shows the fragment shader for rendering paraboloids at a given world size. For a given size in screen pixels, the world size has to be calculated from the screen size first. This can be calculated by rearranging equation 3.2.

```
FragmentOutput frag(VertexOutput o) {
  FragmentOutput fragout;
  float uvlen = o.uv.x*o.uv.x + o.uv.y*o.uv.y;
  if (_Circles >= 0.5 && uvlen > 1) {
    discard;
  }
  if (_Cones < 0.5) {
    o.viewposition.z += (1 - uvlen) * _PointSize;
  }
  else {
    o.viewposition.z += (1 - sqrt(uvlen)) * _PointSize;
  }
  float4 pos = mul(UNITY_MATRIX_P, o.viewposition);
  pos /= pos.w;
  fragout.depth = pos.z;
  fragout.color = o.color;
  return fragout;
}
```

Listing 3.6: Creating paraboloids or cones in the fragment shader. _Cones is an integer used as a boolean

The interpolation improves the results highly. Figure 3.8 shows the results of rendering a scene with interpolation. The improvement can especially be seen on fine details, e.g. on the ships in said Figure. When using high point sizes, cones preserve the texture of surfaces better than paraboloids. However, edges of objects stand out more prominently and the image therefore looks more inconsistent, which can be seen in Figure 3.9. A disadvantage of this method is, that by changing the depth value in the fragment shader, early depth tests cannot be done by the GPU which leads to a reduced performance.


(a) No interpolation


(b) Cones

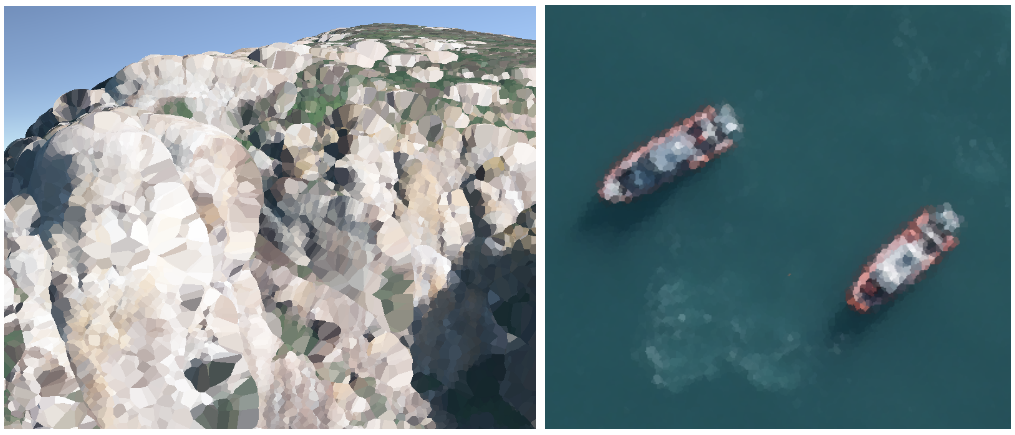
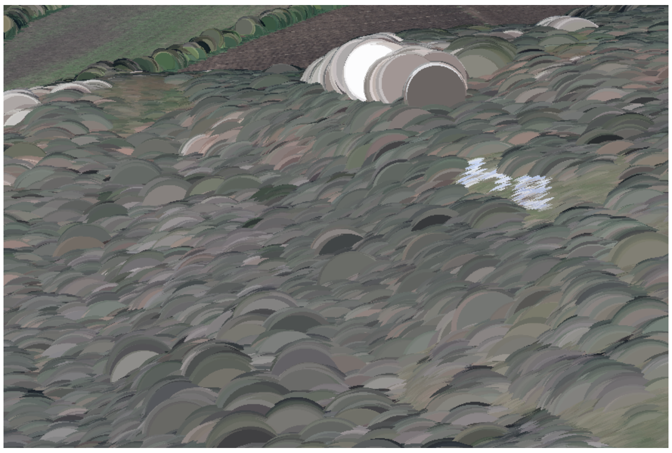
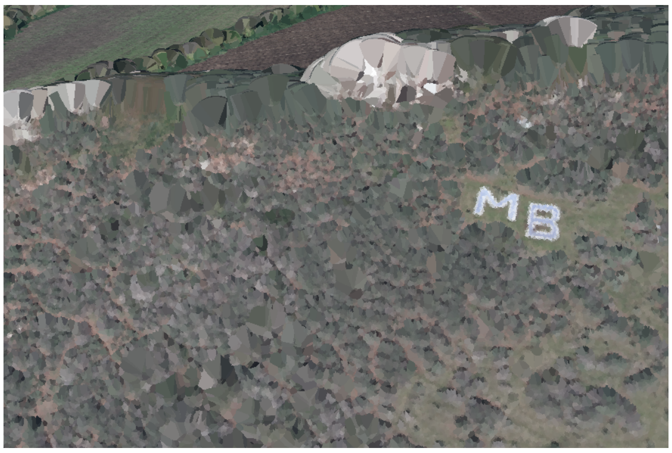

(c) Paraboloids

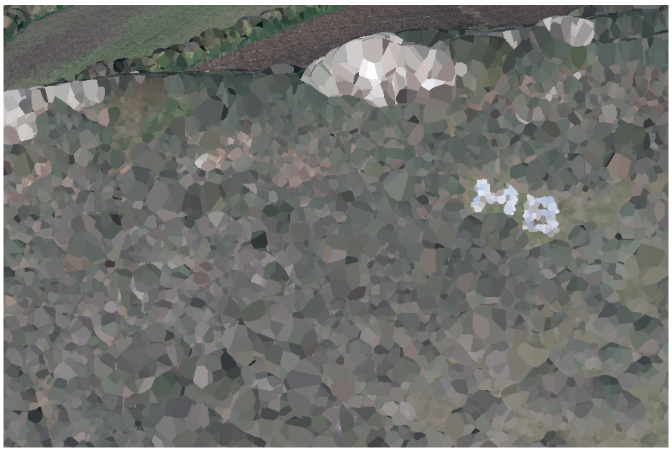
Figure 3.8: Results of rendering with different interpolation techniques



(a) Without interpolation



(b) With cones



(c) With paraboloids

Figure 3.9: Details in the point cloud with different interpolation modes and big point sizes

Approximated Shapes in the Geometry Shader

Another way to create paraboloids is to create a paraboloid-polygon-approximation instead of a square in the geometry shader. Four different paraboloid approximations have been implemented. The paraboloid is approximated either by 8, 16, 32 or by 48 triangles. The 48-triangle structure can be seen in Figure 3.10. All points along one "ring" have the same z value. For each vertex, the depth is calculated and the z value is set accordingly. The resulting three-dimensional paraboloids are depicted in Figure 3.11. The 8-triangle-approximation is closer to a cone than a paraboloid and can therefore be used as a cone-approximation.

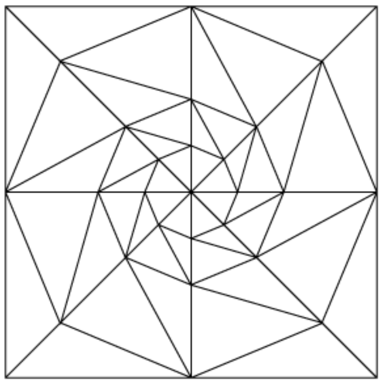
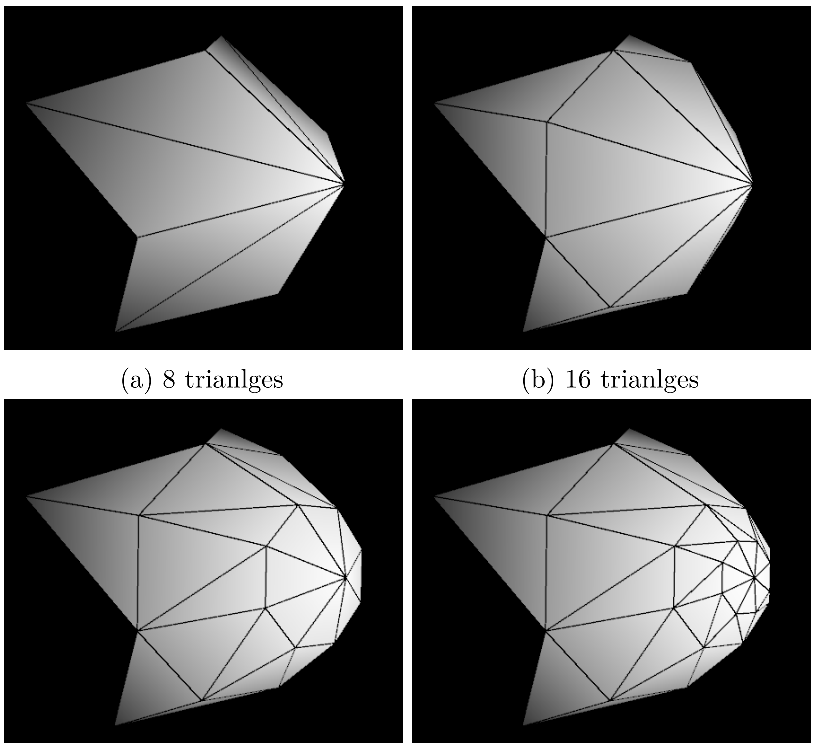


Figure 3.10: The 48-triangle structure for a paraboloid

Creating this for a given point size in world units is simple. In the vertex shader, an up-vector and a sideways-vector are calculated. The point position is only transformed to world space. Then the single points are calculated in the geometry shader from the uv-coordinates (ranging from -1 to +1 in the corners) of the current vertex by the method depicted in listing 3.7. For each vertex, the distance offset is calculated and moved accordingly nearer or farther from the camera. These points are then transformed to screen space. From the created vertices the triangles are created.



(a) 8 trianlges (b) 16 trianlges

(c) 32 trianlges (d) 48 trianlges

Figure 3.11: The paraboloid approximations

```
VertexOutput createParaboloidPoint(VertexMiddle input, float u, float v) {
  VertexOutput nPoint;
  nPoint.position = input.position;
  nPoint.position += u*input.R;
  nPoint.position += v*input.U;
  float4 N = -float4(normalize(float3(nPoint.position - _WorldSpaceCameraPos)
    )*_PointSize, 0);
  nPoint.position += (1 - (u*u + v*v))*N;
  nPoint.position = mul(UNITY_MATRIX_VP, nPoint.position);
  nPoint.color = input.color;
  nPoint.uv = float2(u, v);
  return nPoint;
}
```

Listing 3.7: Creating paraboloid points for a given size in world units. This function is called from the geometry shader.

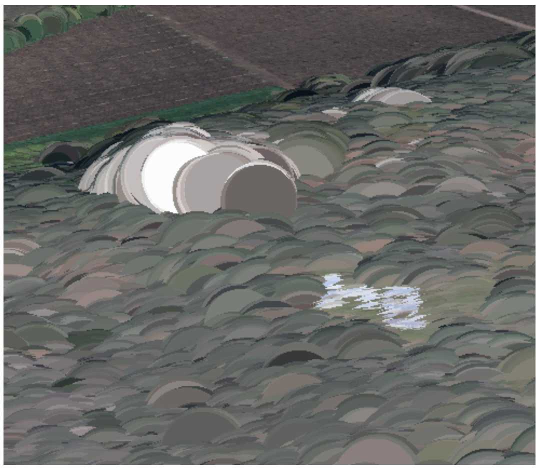
The process for rendering at a given point size in pixels is slightly more complicated. The point position is first transformed to screen coordinates in the vertex shader. In the geometry shader, the points are created by the method in listing 3.8. The x and y coordinates are adjusted by the desired point size (xsize is the point size divided by the screen width and ysize is the point size divided by the screen height). The so resulting point is then projected back to view space by multiplying with the inverse projection matrix. Each vertex is moved closer to or away from the camera according to the calculated paraboloid-value. The variable zsize is the radius of the splat in world coordinates, once again calculated from rearranging equation 3.2.

```
VertexOutput createParaboloidPoint(VertexMiddle input, float xsize, float
    ysize, float zsize, float u, float v) {
  VertexOutput nPoint;
  nPoint.position = input.position;
  nPoint.position.x += u*xsize*input.position.w;
  nPoint.position.y += v*ysize*input.position.w;
  nPoint.position /= nPoint.position.w;
  float4 viewposition = mul(_InverseProjMatrix, nPoint.position);
  viewposition /= viewposition.w;
  float4 vpn = float4(normalize(float3(viewposition.x, viewposition.y,
    viewposition.z)),0);
  viewposition += (1 - (u*u + v*v))*vpn*zsize;
  viewposition = mul(UNITY_MATRIX_P, viewposition);
  viewposition /= viewposition.w;
  nPoint.position = viewposition;
  nPoint.color = input.color;
  nPoint.uv = float2(u, v);
  return nPoint;
}
```

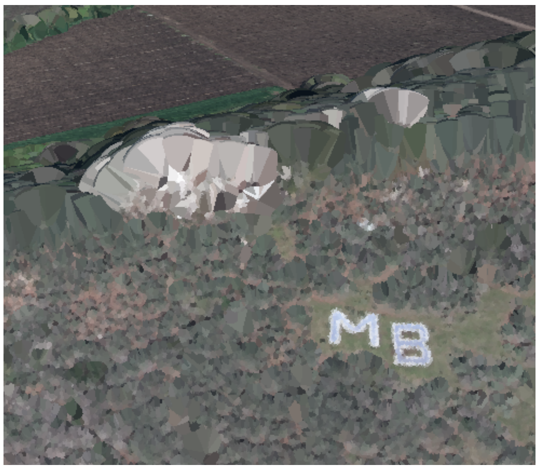
Listing 3.8: Creating paraboloid points for a given size in pixels. This function is called from the geometry shader.

Even though the paraboloids created by the geometry shader are only approximated, the results are still very satisfying. Figure 3.12 compares some of the shown rendering techniques. The difference between geometry shader approximations and fragment shader paraboloids are hardly noticeable.

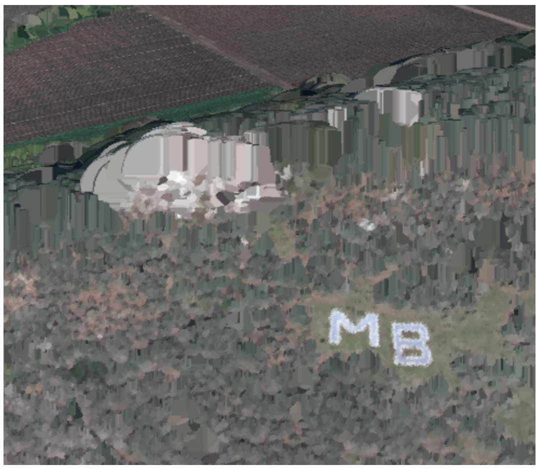
Even though this method does not skip the early z-testing, it is still not as efficient as the fragment shader approach, as will be shown in section 4.



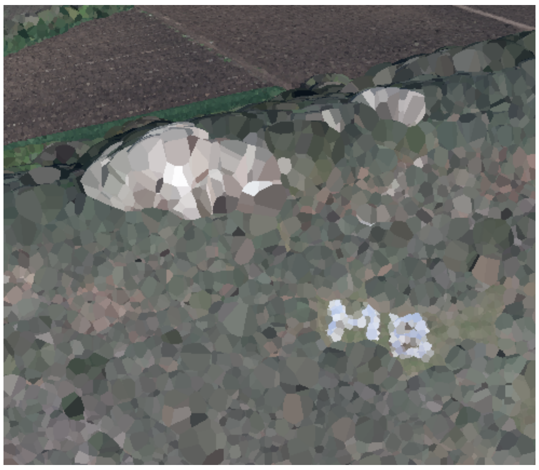
(a) No interpolation



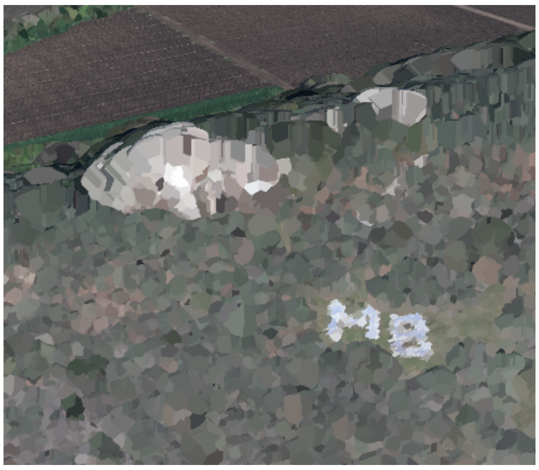
(b) Fragment-Shader cones



(c) 8-triangle-approximation



(d) Fragment-Shader paraboloids



(e) 48-triangle-approximation

Figure 3.12: A detail in the scene rendered with different techniques.

CHAPTER 4

Evaluation

4.1 Methodology

In order to compare the different rendering techniques, the camera was set to a fixed position and the application was executed for ten seconds. During these ten seconds, the point cloud is loaded and rendered. As point cloud, a part of the San Simeon point cloud by OpenTopography[san] was used. This point cloud has around 230 million points. A display resolution of 1366x768 was used, because this was the highest resolution available on both test devices. The application was built as an executable and executed outside of the Unity editor. During the rendering, the duration of each frame (given by *Time.deltaTime* by Unity) was logged. From these values, the average number of frames per second (FPS) was calculated. The average FPS is equal to the inverse of the average deltaT-values and can be calculated as seen in equation 4.1.

$$\text{fps} = \frac{n}{\sum_{i=1}^{n} \Delta \text{t}_i} \tag{4.1}$$

The Memory Profiler was used to measure the maximum memory usage. Tests using the profiler were done separately from the FPS-measurements described above, because profiling can only be done in the editor, where the FPS are usually lower.

The measurements were done on two PCs. PC A is a gaming PC, while PC B is an average laptop.

	PC A	PC B
Processor	Intel Core i7-6700, 4x 3.40GHz	Intel Core i7-3632QM, 4x 2.20GHz
RAM	16GB	8 GB
Operating System	Windows 10 Pro, 64-bit	Windows 10 Pro, 64-bit
GPU	NVIDIA GeForce GTX 1060	Intel HD Graphics 4000

4.2 Point-Pixel Rendering

In this section, three different approaches are compared for rendering points as single pixels: Single-pixel point rendering (see section 3.3.1), 4-vertex quad rendering (see section 3.3.2) with a radius of 1 pixel and geometry shader quad rendering (see section 3.3.3), also with a radius of 1 pixel. All methods render to a single pixel in order to evaluate the overhead of the 4-vertex and geometry shader approaches against the single-pixel point rendering method that is included in Unity.

As point budget 1,000,000 was chosen and the minimum node size was 10. Table 4.1 shows the results.

	∅ FPS PC A	∅ FPS PC B
Single-Pixel Point Rendering	1053.49	91.71
4-Vertex Quad Rendering	601.48	45.17
Geometry Shader Quad Rendering	650.33	50.75

Table 4.1: Average FPS for different rendering methods for drawing each point as a single pixel.

The results show that 4-vertex quad rendering is not a good option, as it has the lowest FPS of the three techniques. It also increases memory usage by a factor of 4. The geometry shader approach does not reach the speed of the single-pixel point rendering, but it still provides good results.

4.3 Point-Quad Rendering

Two techniques have been implemented to render points as screen faced squares or circles with a given size in pixels. There is the 4-vertex quad rendering and the geometry shader quad rendering. The last one also provides various interpolation techniques.

For the evaluation, a point budget of 1,000,000 was chosen. The point size was set to 5 pixels and the points were drawn as squares. The minimum node size was once again 10. The results are shown in table 4.2.

While the simpler geometry-shader interpolations still work fine on PC A, they are hardly usable on PC B due to the low frame rates. The fragment shader approach for interpolation is faster and generates mostly better graphical results as well.

4.4 Point-Budget Influence

To find out how the point budget influences the performance, the test scene was evaluated with different point budgets, once using single-pixel point rendering and once using screen-space-sized point rendering with fragment-shader-paraboloids. Figure 4.1 shows the measurements of the frames per seconds. The measurements were done with 500,000,

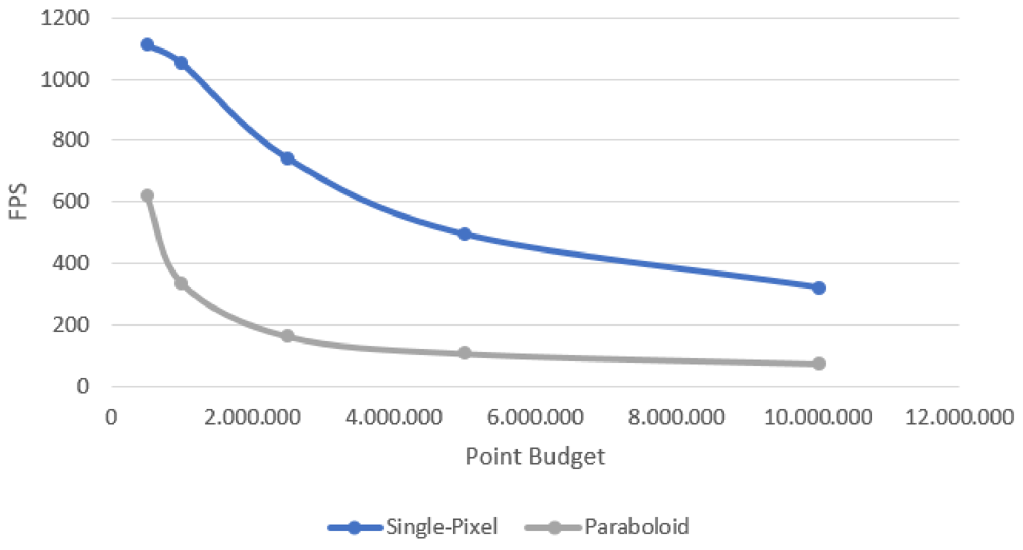
	∅ FPS PC A	∅ FPS PC B
4-Vertex Quad Rendering	560.42	26.71
Geometry Shader: No interpolation	568.57	31.36
Geometry Shader: 8 triangle-approximation	201.4	17.03
Geometry Shader: 16 triangle-approximation	120.03	16.06
Geometry Shader: 32 triangle-approximation	73.51	16.85
Geometry Shader: 48 triangle-approximation	70.38	13.76
Geometry Shader: Fragment-Shader-cones	339.69	17.98
Geometry Shader: Fragment-Shader-paraboloids	322.21	18.18

Table 4.2: Average FPS for different rendering methods for drawing each point as a screen faced circle with a point budget of 1,000,000.

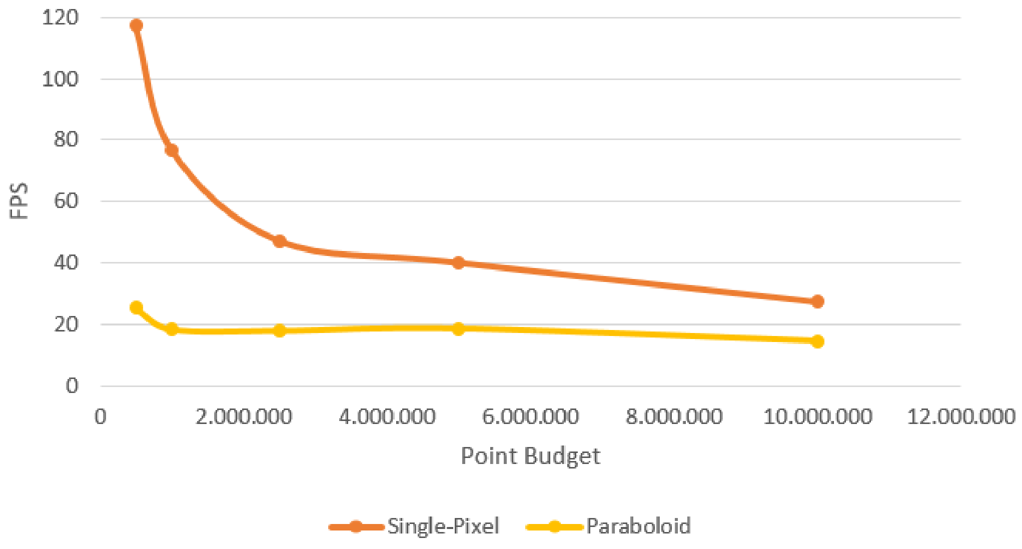
1,000,000, 2,500,000, 5,000,000 and 10,000,000 points. Please note, that at the start of the ten-second measurement, the points are not loaded yet, but instead they are loaded during the first seconds, which means that the FPS are higher in the beginning - because fewer points to render are loaded - than at the end of the measurement.

Single-pixel point rendering always works fine on PC A. The FPS are always above 300. On PC B the performance is generally lower, however, for point budgets below 2,500,000, it is mostly still usable.

With the paraboloid method, the average FPS value of PC A is above 600 for 500,000 points but starts to drop noticeably from 1,000,000 points onwards. However, it still remains usable with FPS above 60. The performance of PC B is already very low from the beginning.



(a) PC A



(b) PC B

Figure 4.1: FPS-measurements for different point budgets

4.5 Memory Consumption

The size of the test point cloud's files is 3.49 GB. The amount of memory used by the application depends mainly on the point budget and the size of the LRU cache. Figure 4.2 shows the maximum memory usage by point budget for two different cache sizes. This was measured while navigating through the scene on a predefined path for 40 seconds. Single-pixel point rendering was used as rendering method. The second cache size, 300,000,000 points, exceeds the number of points in the point cloud, so it is an unlimited cache, that never deletes points from memory. As expected, the memory consumption increases linearly with the point budget and using the cache makes a significant difference.

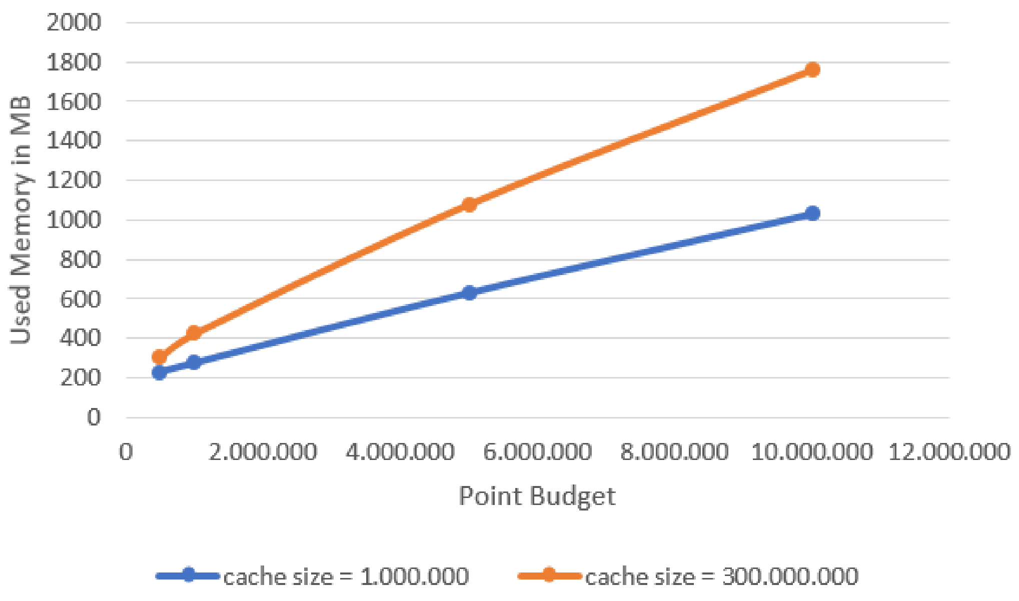


Figure 4.2: Maximum memory usage in MB per point budget

4.6 Comparision to Potree

To compare the performance of the Unity renderer with Potree, the same test point cloud was rendered with Potree from the same camera position as in Unity. Potree was run in the Chromium browser [chr], because it is one of the few browsers that support disabling vertical synchronization (VSync). It was tested on PC A with a resolution of 1920x1080. Potree was executed and the frame durations were recorded for ten seconds. Fixed point sizing and square-rendering were used. A minimum node size of 10 was chosen. The other settings varied through the different tests. Each test configuration was tested in Potree as well as in Unity. In all the Unity tests, Geometry Shader Quad Rendering was used. The results can be seen in table 4.3.

Point Budget	Interpolation	Point Size	∅ FPS Potree	∅ FPS Unity
1,000,000	No	1px	382.85	563.32
10,000,000	No	1px	97.08	114.19
1,000,000	No	5px	350.75	503.97
10,000,000	No	5px	88.65	105.25
1,000,000	Yes	5px	358.55	312.37
10,000,000	Yes	5px	74.45	71.33

Table 4.3: Average FPS in Potree and Unity

The Unity renderer usually has more FPS than Potree. Only when interpolation is used, Potree has a slightly better performance.

CHAPTER 5

Conclusion

A point cloud rendering system for Unity was presented. It renders point clouds in the Potree-data format in real time by continuously checking which parts of the point cloud are visible and loading only these. The number of loaded and displayed points are limited by a point budget and a minimum node size. Various different rendering approaches have been implemented, including fixed point sizes in world units or screen units as well as the possibility to use paraboloid- or cone-interpolations.

For systems with lower computing power and less powerful graphic devices, the settings can be adjusted to reduce the quality in order to still get an acceptable number of frames per seconds.

Possible future work includes the implementation of adaptive point sizes, so that in areas with lower point density the points are displayed with a bigger size than in areas with higher point density. Also, Eye-Dome-Lighting[Bou09] could be implemented to create outlines along silhouettes in order to improve the visual quality.

Bibliography

[BHZK05] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's GPUs. In *Point-Based Graphics, 2005. Eurographics/IEEE VGTC Symposium Proceedings*, pages 17–141. IEEE, 2005.

[Bou09] Christian Boucheny. *Visualisation scientifique de grands volumes de données: Pour une approche perceptive.* PhD thesis, Université Joseph-Fourier-Grenoble I, 2009.

[chr] The chromium projects. `https://www.chromium.org/`. Accessed: 2017-08-30.

[Fra] Simon Fraiss. GitHub Repository BA_PointCloud. `https://github.com/SFraissTU/BA_PointCloud`. Accessed: 2017-08-04.

[GM04] Enrico Gobbetti and Fabio Marton. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(6):815–826, 2004.

[LW85] Marc Levoy and Turner Whitted. *The use of points as a display primitive.* University of North Carolina, Department of Computer Science, 1985.

[ope] OpenTopography. `http://www.opentopography.org/`. Accessed: 2017-07-23.

[pcv] Unity Asset Store: Point Cloud Free Viewer. `https://www.assetstore.unity3d.com/en/content/19811`. Accessed: 2017-07-23.

[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.

[san] OpenTopography: PG&E Diablo Canyon Power Plant (DCPP): San Simeon, CA Central Coast. `http://opentopo.sdsc.edu/lidarDataset?opentopoID=OTLAS.032013.26910.2`. Accessed: 2017-07-23.

[Sch] Markus Schütz. Potree website. `http://potree.org/`. Accessed: 2017-07-23.

[Sch14] Claus Scheiblauer. *Interactions with Gigantic Point Clouds.* PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2014.

[Sch16] Markus Schütz. Potree: Rendering large point clouds in web browsers. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, September 2016.

[SW15] Markus Schütz and Michael Wimmer. High-quality point based rendering using fast single pass interpolation. In *Proceedings of Digital Heritage 2015 Short Papers*, pages 369–372, September 2015.

[WS06] Michael Wimmer and Claus Scheiblauer. Instant points. In *Proceedings Symposium on Point-Based Graphics 2006*, pages 129–136. Eurographics, Eurographics Association, July 2006.