Universität Bremen*
*EXCELLENT.

Artificial Intelligence

Fachbereich 3: Mathematics and Computer Science
Institute of Computer Graphics and Virtual Reality

Master Thesis

Efficient rendering of massive and dynamic point cloud data in state-of-the-art graphics engines

On the example of the Unreal Engine

Valentin Kraft
Digital Media

Matriculation No. 303 274 2

August 29, 2018

Primary Examiner: Prof. Dr.-Ing. Gabriel Zachmann
Secondary Examiner: Prof. Michael Beetz Ph.D.

Valentin Kraft
Digital Media

Efficient rendering of massive and dynamic point cloud data in state-of-the-art graphics engines

On the example of the Unreal Engine

Master Thesis, Fachbereich 3: Mathematics and Computer Science
Institute of Computer Graphics and Virtual Reality

Universität Bremen, August 2018

Universität Bremen

Nachname **Kraft** Matrikelnr. **3032742**

Vorname/n **Valentin**

Diese Erklärungen sind in jedes Exemplar der Bachelor- bzw. Masterarbeit mit einzubinden.

Urheberrechtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Alle Stellen, die ich wörtlich oder sinngemäß aus anderen Werken entnommen habe, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Datum Unterschrift

Erklärung zur Veröffentlichung von Abschlussarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten.
Archiviert werden:

1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2) Bachelorarbeiten des jeweils der ersten und letzten Bachelorabschlusses pro Studienfach und Jahr.

- (●) Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ( ) Ich bin damit einverstanden, dass meine Abschlussarbeit nach frühestens 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ( ) Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Datum Unterschrift

Danksagung

Viele Menschen haben mich während meiner Masterarbeit unterstützt und dadurch diese Arbeit erst möglich gemacht, wofür ich mich gerne herzlich bedanken möchte. An erster Stelle danke ich Prof. Gabriel Zachmann, von dem ich viel lernen durfte und der mir, auch wenn er oft sehr beschäftigt war, immer Zeit eingeräumt hat und mir jede meiner zahlreichen Fragen beantwortet hat. Gleiches gilt für seinen Mitarbeiter Christoph Schröder, der mir besonders bei vielen technischen Problemen sehr hilfreich zur Seite stand. Vom Institut für Artificial Intelligence danke ich Prof. Michael Beetz für seine Unterstützung und Betreuung und auch seinen Mitarbeitern, vor allem Patrick Mania, für seine Hilfe während der Testphase meiner Arbeit und die spannenden Einblicke in den Bereich der Künstlichen Intelligenz.

Auch wenn sie wieder einmal nicht sonderlich hilfreich waren, danke ich darüber hinaus Nikolas Jürgensen, Julia Hass und Ramneek Singh für ihre Unterstützung und für die zahlreichen Ablenkungen, die zwischendurch auch mal nötig waren. Vielen Dank auch an Marian Turowski und Bent Neuberger, ohne die mein Leben eventuell anders verlaufen wäre. Außerdem danke ich meiner Mama und meinem Papa, auf deren Hilfe ich immer zählen kann.

Abstract

Point clouds have lately gained much popularity since professional laser scanners and consumer devices like the Microsoft Kinect have become available to a broad audience. Nowadays, point clouds are being used in a multitude of industries, like the 3D industry, architecture, robotics, and so on. At the same time, the industries rely more and more on the popular 3D graphics engines for their Real-Time applications, like Unity3D or Epic's Unreal Engine. However, there is just very few software and research available on how to efficiently include or implement a high-quality point cloud renderer into these polygon-based and complex state-of-the-art engines.

In this thesis, I present an efficient way to implement a GPU-based point cloud renderer that is capable of rendering huge both static and fully dynamic point clouds in high quality and Real-Time inside the Unreal Engine. In doing so, a novel way of Order-Independent Transparency (OIT) is drafted by employing a massively parallel bitonic sorting that is sorting the point cloud via a compute shader in Real-Time.

The presented renderer could be applied in various application fields, such as collaborative virtual environments (CVEs) or dynamic and on-the-fly environment scanning, which is relevant for instance in robotics.

The point cloud renderer will be published as an publicly available, open source plugin for the Unreal Engine.

Contents

Chapter 1

Introduction

1.1 Motivation

One of the most basic problems of computer graphics is the question, how to represent and display virtual objects as efficient and realistic as possible. Since the beginning of Real-Time computer graphics, when it comes to rendering of three-dimensional objects, polygons are being used to build up the virtual object. Thus, the underlying primitives are most often triangles. This way of assembling the object can be seen as a form of discretisation. Yet this is not the only way one could think of how to discretise an object; the concept of rendering via alternative primitives like e.g. points is reasonable and, because of the lack of interconnectivity between the points, way simpler than the polygonal approach. The simplicity and elegance of the point representation seem to hint to certain advantages over polygons which is why it may be worth to examine and maybe reconsider the advantages of rendering with points over rendering with polygons ([Kob+04] for further read on this topic).

Furthermore, driven by an increasing use of laserscanners and photogrammetry in a multitude of industries nowadays and the development of inexpensive and powerful consumer devices like Microsoft's Kinect [Smi+11], the relevance of point clouds and the demands for proper software that can handle point cloud data are quickly increasing. Nowadays, points clouds are being used in a large variety of fields, especially in the 3D industry, architecture, the construction industries and robotics. Robotics in particular have a high demand for point clouds, often as a representation of the environment, where the surrounding objects are getting recognised and semantically labelled [Kop+11] or for autonomous driving [Gei+13] or Simultaneous Localization and Mapping (SLAM) [Whi+10]. Furthermore, telepresence and collaborative virtual environments (CVE) [Ben+01] have lately become an interesting research field, involving the application of dynamic Real-Time point clouds in CVEs [Bru+14] to create realistic virtual avatars.

In that context, it is worth investigating how well point cloud rendering can be implemented into state-of-the-art (polygon-based) rendering workflows. Some years ago, this would have mainly implied the established 3D-softwares (such as Autodesk Maya, 3D Studio Max, etc.), however, the increasing visual quality and

flexibility of current Real-Time graphics engines seem to have created a shift in the industry to rather use Real-Time graphics engines (such as Unity3D or the Unreal Engine) than offline or custom 3D-Rendering software. Hence, the relevance of point cloud rendering solutions for these popular graphics engines is obvious and higher than ever before. While there are numerous point cloud rendering algorithms proposed by the research community, very few of them are actually designed for or easy to implement in one of the current Real-Time graphics engines that are increasingly used by the industry, which is why one has to investigate ways of implementing point cloud rendering in these engines in an easy and efficient manner.

The Unreal Engine, created by Epic Games, is a popular and widely used state-of-the-art 3D game engine. It offers all tools to create graphical applications of almost any kind and supports a multitude of platforms, such as Linux, Windows, macOS, iOS, Android and HTML5. It is written in C++ and completely open source, therefore providing the possibility to even make low-level changes and write highly efficient code. Aside from the game industry, Unreal is already used in a variety of other industries, especially in architecture, and is popular in academics, too. For example, Unreal is used in several academic contexts, particularly in robotics projects like "UnrealCV" [Qiu+16], "UnrealStereo" [Zha+16], or the "RobCog" project [Hai+18] as a simulation and testing environment. It is also used as a virtual training environment for artificial intelligence applications [Ler+16] [Sha+17]. Unreal does not provide a built-in solution for rendering or processing point clouds, however.

1.2 Goal of the thesis

The main goal of this thesis is to investigate how to implement a point cloud renderer in an existing, state-of-the-art, polygon-based graphics engine on the example of the Unreal Engine 4. The concrete requirements that I defined for the point cloud renderer were:

- **Efficiency**: The point cloud renderer should be capable of rendering at least medium-sized static point clouds (approx. 1 Million points) as well as streamed/dynamic point clouds in Real-Time.
- **Quality**: The point cloud renderer should render the point clouds as realistic as possible and therefore either apply some sophisticated rendering techniques or surface estimation methods.
- **Usability**: The point cloud renderer should be easy to use and fully integrated into the engine's pipeline without manipulating it, making it usable in industry-typical scenarios and use cases.

To test the fulfilment of this goals, the point cloud renderer will be evaluated by rendering some industry-standard static point clouds as well as rendering a dynamic point cloud stream coming from a Kinect in an exemplary robotics-related use case where a dynamic scan of the environment is captured. The resulting point cloud is then compared against the ground truth, given by a accurate virtual representation of the real environment inside the Unreal Engine (see chapter 5).

1.3 Thesis structure

The thesis is structured in six general chapters.
After the introduction, which is covered by this chapter, I will first focus on the fundamentals of point clouds in the second chapter, identify problem fields and present and discuss relevant scientific work and general approaches in point cloud rendering and related areas.
Based on a short analysis of the Unreal Engine and its possibilities and constraints, I will subsequently try to identify suitable approaches and draft the conceptual design of my solution and rendering algorithm, thus building the theoretical basement of the point cloud renderer.
After that, I will cover the implementation details and the difficulties on the way and how I solved them in the fourth chapter.
This is then followed by the presentation of the results in the fifth chapter. Here,

the point cloud renderer is getting evaluated in the aforementioned use cases by analysing the results in qualitative and quantitative ways to deduce if and how well the former set goals were met.
The thesis will be finishing with the conclusion chapter by summing up the thesis in general terms, discussing if and how well the initial aims were fulfilled based on the results, illustrating the limitations of the presented approach and embedding the results into the research context as well as giving ideas for future work.

Chapter 2

Fundamentals

2.1 Point Cloud fundamentals

Generally speaking, a point cloud is a set of points in some coordinate system. In most of the cases, this coordinate system is a three-dimensional coordinate system, in which the points are usually defined by spatial X, Y, and Z coordinates. In most of the cases, the points are intended to represent the external surface of one or multiple objects. Often, there is also additional information stored alongside the point positions, such as the individual colours of the points, the normals, etc.

Point clouds may be used for many purposes and have been employed in many fields, such as robotics, 3D modelling, archaeology, architecture, geography and medical imaging, just to name a few. In general, point clouds often come into play when we want to capture and transfer real-life objects or scenes into the computer to further process, analyse or work with the data. 3D point clouds can be captured in a variety of ways, but the arguably most popular approaches are laser scanning and photogrammetry.

3D laser scanners (due to their ”Light Detection and Ranging” sensors often also referred to as ”LiDAR scanners”) are capturing their surrounding environment by taking a distance measurement (usually at every direction) via laser beams and are available as airborne and terrestrial devices. The scanning process yields a large number of independent points (the ”point cloud”), representing the observable surfaces of the surrounding objects, which are then usually saved in a point cloud data file. Typically, the laser scanner measures and stores not only the raw point positions, but saves further measurable data such as for example the reflectivity (often referred to as ”intensity”) or the colour of the scanned surface point. Due to the fact that stationary laser scanners can only fulfil a partial scan of the environment, it is often necessary to later merge the individual point cloud to gain a representation of the whole environment, which can be a difficult computational problem. Due to the fact that some modern laser scanners can capture more than one million points per second and multiple scans are often merged into one large scan, the point clouds often contain millions, sometimes even billions of points or more.

Photogrammetry, in contrast, tries to reconstruct the 3D data from a set of 2D photographs from different viewing angles. The principle behind most of the algorithms is to first find corresponding feature points in the images and then reconstruct their depth (and hence, their spatial position) through triangulation, which will eventually lead to the reconstructed point cloud. A well known and popular algorithm for the reconstruction is called "Patch-based Multi-View Stereo" (PMVS) and is based on the work of Furukawa and Ponce [Fur+10]. Further information on this topic can be found in [Fur+15].

In addition to that, there are numerous other capturing techniques, which would go beyond the scope of this thesis and therefore cannot be covered in detail, however.

2.2 Challenges & problems

Although the point cloud representation leads to several advantages such as simplicity, scalability and easiness of capturing and handling, new problems and challenges are introduced when working with point clouds. The most important ones are arguably:

- **Big data sizes**: Current laser scanners are able to capture up to one million points per second which quickly leads to vast datasets, making point clouds hard to transfer and process and difficult to render in Real-Time, even for state-of-the-art devices.
- **Sparse or false data** (especially for laser-scanned datasets): Since the LiDAR sensor beams are emitted circularly, the density of the sampling is reduced with ongoing distance. Furthermore, due to occlusion not all areas may be covered equally by the laser beams, sometimes creating big holes and gaps or outliers in the dataset. Moreover, scanning transparent, semi-transparent or reflective surfaces can be difficult and may introduce false data or severe noise. The sensor itself introduces noise, too, which is then inherent in the sample data. Additionally, depending on the device and method of gathering and if the point cloud already got processed, the available data per point can

differ a lot as well, sometimes including color information, normals, intensity, etc. Often, a point cloud comprises point positions only, however.

- **Visualisation**: Even for reasonably small and correct datasets, the general rendering of point clouds is not trivial, since point clouds may only contain 3D positions without any connectivity or normals, but most of the time should be rendered as continuous surface and as close to reality as possible.
- **Merging of multiple scans**: Due to the fact that laser scans can only capture one point of view or one distinct area, it is often necessary to merge multiple scans. The merging problem is inherent for the photogrammetry approaches as well.

As one can see, numerous problems are involved in the realistic rendering of point clouds and manual pre- or post-processing usually was an inevitable part of the pipeline. Thus, various research areas have evolved that try to cope with these issues with the goal to make manual or additional processing of the point clouds superfluous and to visualise them as good and efficient as possible:

- Big data sizes problem → Data compression & reduction algorithms, out-of-core solutions, datastructures & LOD, etc. (see 2.3.1)
- Sparse/false data problem & varying data per point → Algorithms for surface estimation, reconstruction of normals, hole-filling, smoothing, de-noising etc. (see e.g. 2.3.2.1)
- The Visualisation problem → General rendering algorithms; Trying to visualise the point cloud data as efficient and close to reality as possible, often combined with surface estimation approaches, especially with applications of advanced acceleration techniques such as GPGPU, Screen-Space computations, etc. (see 2.3.2)
- Merging problem → Registration & matching algorithms

Of course, a variety of other research areas have evolved as well, such as classification, feature extraction and segmentation, to just name a few. The variety of required and complex algorithms led to the development of the open-source Point Cloud Library (PCL) [Rus+11] in 2011, which embodies numerous algorithms for processing of multidimensional point clouds, including filtering, registration, seg-

mentation, surface reconstruction, feature extraction and visualisation.

Since it would not be feasible to adequately cover all point cloud related research areas and the topic of this thesis is clearly rendering-related, the following chapter will mainly focus on the approaches that are relevant in the context of the formerly set goals (see 1.2) of the point cloud renderer, that is, particularly the rendering-related approaches and their basics.

2.3 Previous Work

2.3.1 Point Cloud organisation & data structures

As stated above, the big data sizes of large point clouds are a major problem when one wants to transfer, display or process the data. While the points in a point cloud are inherently unconnected, it is often crucial to be able to search the dataset for certain points, e.g. for adjacent points when one wants to estimate a continuous surface or just render a certain part or subset of the point cloud (like, for example, in Level-Of-Detail approaches) or further process the point cloud like, for example, find matching points between multiple datasets to merge them. For spatial organisation and division of data, hierarchical tree-based data structures, like for example kd-trees [Ben75] or octrees [Mea82], have proven to be beneficial.

2.3.1.1 Level Of Detail & Out-Of-Core

"QSplat" [Rus+00] uses a bounding sphere hierarchy to organise the point cloud data, in which the actual data is contained in the leaf nodes, while the inner nodes contain averages of their children in order to represent the children on a lower level of detail. By that, the algorithm is able to render depending on a given level of detail, defined by a point budget, rendering finer point resolution only where needed. Based on this approximation approach, the QSplat algorithm is

able to both render very large datasets or perform on weak hardware, making it a very flexible system. "XSplat" [Paj+05] uses "Sequential Point Trees" to create an efficient LOD-based out-of-core rendering system that can handle even huge datasets that would otherwise not fit into the system's memory. "Instant points" [Wim+06] further develops the Sequential Point Trees to be memory-optimised and introduces "nested octrees" as an out-of-core data structure, allowing LOD-based rendering on completely unprocessed point clouds. [Wan+07] presents an out-of-core multi-resolution octree with the ability to interactively edit the point cloud in Real-Time. [Sch+15b] develops, mainly inspired by the QSplat approach, the "potree" system, which uses multi-resolution octrees and on-demand queries to realise an extremely flexible LOD-based point cloud renderer that can run in the browser on almost all devices.

2.3.1.2 Approximate Nearest Neighbour search

"Approximate Nearest Neighbour" (or "kNN", coming from "k-nearest neighbour search") algorithms are highly specialised and efficient algorithms that most often employ hierarchical data structures (see 2.3.1) to find a points' adjacent points in an unorganised dataset. The "FLANN" library [Muj+14] is one popular example and is embedded and utilised in the PCL library for filtering, surface estimation, feature extraction and registration of point clouds. "EFANNA" [Fu+16] is a recent alternative library, promising even higher search performance. [Zho+08b] investigates the creation of kd-Trees on the GPU to be able to perform queries (e.g. nearest neighbour searches) faster than on the CPU. [Pre+12] is proposing an unusual GPU-based nearest neighbour search method, where a parallelised distribution- and gathering-pass is executed on the projected points in Screen-Space to quickly estimate a point's neighbours in a certain radius.

2.3.2 Point Cloud rendering

As stated above, the rendering of point clouds is not trivial at all. Of course, it is possible to circumvent the problems of point cloud rendering by simply converting the point cloud into a regular mesh by employing one of the common meshing algorithms like Delaunay Triangulation [Law72], Alpha Shapes [Ede+94] or Marching Cubes [Lor+87] (an extended survey on this topic is given in [Men+97]). But since the process of converting a point cloud to a polygonal mesh often does not produce satisfying results for more complex objects and introduces a lot of additional and computationally expensive steps [Rem04], making Real-Time rendering not feasible, meshing is often not a good choice. Therefore, direct ways of rendering are in demand that render unprocessed point clouds as realistic and fast as possible, utilising only the available data.

Another idea is to create a solid surface not by converting the point cloud into a mesh but by deriving a mathematical surface representation using higher order polynomials, for example by a moving least squares (MLS) regression [Lev98]. This often produces smooth and, when combined with proximity graphs like in [Kle+04b] [Kle+04a], topologically correct surfaces. The major drawback is however the rendering, which again requires sampling, that is, by meshing or raytracing it, thus making implicit surfaces not well suited for Real-Time applications.

In contrast to the approaches of generating solid surfaces from the point cloud which involve additional steps as stated above, there is also the possibility of using the points themselves as a display primitive. One of the first to investigate the use of points as a approximation for continuous surfaces were Levoy and Whitted [Lev+85]); Yet the idea was already used in particle systems for rendering objects without solid surface (such as fire, clouds, smoke, etc.) [Ree83]. But since simple point samples do not lead to a plastic impression for a solid three-dimensional object, Westover [Wes89] further developed the idea of point primitives into the concept of *Splatting*, where a single point gets mapped to multiple pixels on the screen by computing 2D ”footprint functions” with the colour of a pixel being defined by the weighted averages of the adjacent points. Pfister et al. [Pfi+00] then introduced ”surfels”, which basically are elliptically shaped primitives, positioned

on an object's surface in three-dimensional space, with additional and individual orientations, colours and scalings. They also introduced a hierarchical datastructure called "layered depth cube" and implemented and evaluated simple surface reconstruction and hole filling filters in image space. Since then, the idea of splatting was picked up by numerous scientific works, e.g. improving it by applying advanced texture filtering (via a "Elliptical Weighted Average" filter) which essentially results in a blurring per splat and thus finally producing continuous surfaces and smooth objects [Zwi+01]. Since most of the former approaches are completely software-based, thus providing limited performance, [Bot+05] proposed a hardware accelerated, GPU-based splatting with a deferred shading approach. Hence, splatting proved to produce high quality renderings while still being fast and simple to implement and most of the current point cloud rendering approaches today are in some way based on it. Since then, focus has been mainly set on further speed-ups by employing sophisticated and specialised data structures, further utilising the GPU or working in Screen-Space.

2.3.2.1 Surface estimation

In order to create a realistic impression of the underlying objects, it is crucial to be able to display it with continuous surfaces. When using a splatting-/surfel-based approach, this means that for each visible point the normal and the dimensions/extent of each splat have to be known. If the normals of the points are not available in the dataset, numerous normal estimation methods (e.g. [Mit+03], [Liu+12], [Zho+08a]) can be applied to estimate the normals and thus the objects' surface. At the heart of these estimation methods, it is crucial to be able to search the points' proximity for the points' nearest neighbours (see 2.3.1.2). Because this is computationally quite expensive, Real-Time approaches often try to avoid the surface estimation in the three-dimensional object space, but rather try to do it in the two-dimensional Image-Space. [Pre+12] is performing the nearest neighbour search in a parallel fashion utilising the GPU in Screen-Space in order to compute the points' orientation and radii at interactive framerates. [Dob+10] does not try to rely on a nearest neighbour estimation at all, instead using several filter steps

in Image-Space to generate a continuous and smooth surface. Additionally, their approach allows for point cloud visualisation with transparency and soft shadows at interactive framerates by employing monte-carlo integration and depth peeling. [Pin+11] works entirely in Image-Space in a similar fashion, but uses a more sophisticated reconstruction pipeline and an advanced deferred shading with screen-space ambient occlusion for better shape depiction. More information on this vast topic can be found in [Ber+14].

2.3.2.2 Point Cloud rendering in Game Engines

Surprisingly, very few approach try to implement point cloud renderer in current game engines. One reason might be that most of the rendering approaches rely on rather unique rendering techniques or even specialised GPGPU-based pipelines (e.g. [Gün+13]) and thus are not easy to include in current polygon-based pipelines. Still, [Fra17] successfully implements a point cloud rendering solution, which is heavily inspired by the "Potree" approach [Sch+15b], in the Unity3D engine. Similar to the potree approach, it uses an octree to realise a LOD-based rendering and vertex or geometry shader to create and render the individual splats. Besides this, scientific contributions for point cloud rendering in game engines are almost not findable and one finds only various little tools and plugins, mainly for the Unity3D engine.

2.3.2.3 Dynamic Point Cloud rendering

Particularly challenging is the rendering of dynamic and continuously changing point clouds, coming e.g. from a streaming device like Microsoft's Kinect. The fact that the point cloud is changing over time makes classical surface estimation approaches difficult since updating the underlying tree structures is often prohibitively slow. [Pre+12] is therefore moving the nearest neighbour search to the Screen-Space and the GPU, in order to be able to estimate the surfaces and render

multiple dynamic point clouds at interactive frame rates. Surprisingly, Preiner et al. are one of the very few that tackle the problem of rendering a changing point cloud while employing a surface estimation in Real-Time.

2.3.3 Order-Independent Transparency

When rendering semi-transparent splats (as proposed by [Zwi+01]), the correct depth-ordering of the points is crucial to get a visually correct rendering of the point cloud. The algorithms that try to ensure the correct ordering of transparent objects are known under the term "Order-Independent Transparency" (OIT). OIT is still an ongoing research area and relevant recent works on this are for example [Mcg+13] who realise OIT by computing weighted sums of semi-transparent objects and [Sch+15a] who propose an approach for the integration of OIT into a deferred shading pipeline. Surveys on this topic are given in [Mau+11] and [Liu13].

2.3.4 GPU-based Sorting

In order to achieve a correct depth-ordering of the points, one efficient yet rarely used option is to sort the points directly on the GPU via a parallel sorting algorithm. Sorting on the GPU is a well studied research field and among the variety of different massively parallel sorting algorithms, "Radix Sort" and "Bitonic Sort" are arguably the most efficient and most popular ones. A thorough survey about current GPU sorting approaches are given in [Cap+12] and [Ark+17].

2.3.4.1 Parallel Bitonic Sorting

The Bitonic Sorting algorithm was first formulated by Batcher [Bat68] as a sorting network and is a data-independent sorting algorithm and thus well suited for parallelisation. The bitonic sorting network consists of multiple sorting stages with

the so-called "half cleaner" being the fundamental and only building block. The half-cleaner requires an input which forms a bitonic sequence. A bitonic sequence is defined as follows:

Definition 2.1 *A sequence of numbers* $\{a_0, \ldots, a_{n-1}\}$ *is called bitonic when there is an index* i *such that*

- $\{a_0, \ldots, a_i\}$ *is monotonically increasing* ***and***
- $\{a_{i+1}, \ldots, a_{n-1}\}$ *is monotonically decreasing* ***or***
- *if there is a cyclic shift of this sequence such that this is the case.*

With a bitonic sequence as input, a half-cleaner now takes the first half of the input and performs a point-wise min/max operation with the second half of the input. For this two new sequences L_a and U_a the property holds that each element of L_a is less than or equal to each element of U_a. Since the two output sequences are bitonic, too, they can be used as inputs for another half-cleaner in the next stage. Repeating this divide-and-conquer approach, an bitonic input sequence of arbitrary length can be sorted efficiently. This is called a bitonic merger (see green or blue boxes in Fig. 2.1). Since in reality it is unlikely to have a bitonic sequence as an input, the first stages create a bitonic sequence from an arbitrary input. Starting at the finest level of comparing two single values (which, by definition, also form a bitonic sequence), the input is getting sorted in increasing order (blue boxes) or decreasing order (green boxes). The combinations of the boxes, again, result in bitonic sequences, so that in the last stage the full input sequence is "converted" into a bitonic sequence and can be fully sorted by the last bitonic merger. In that way, an arbitrary sequence of length 2^m can be efficiently sorted, yielding a butterfly sorting network.

Since the half-cleaners operate on distinct subsets of the input, the bitonic sorting algorithm is well suited for an implementation on parallel architectures. However, given the fact that the shared memory on current GPUs is still fairly limited, sorting huge datasets that exceed the shared memory is not trivially possible with the bitonic sorting algorithm and requires further modification.

Since the original formulation in 1968, the bitonic sorting algorithm has been extended and altered in numerous approaches, e.g. by lowering the computational

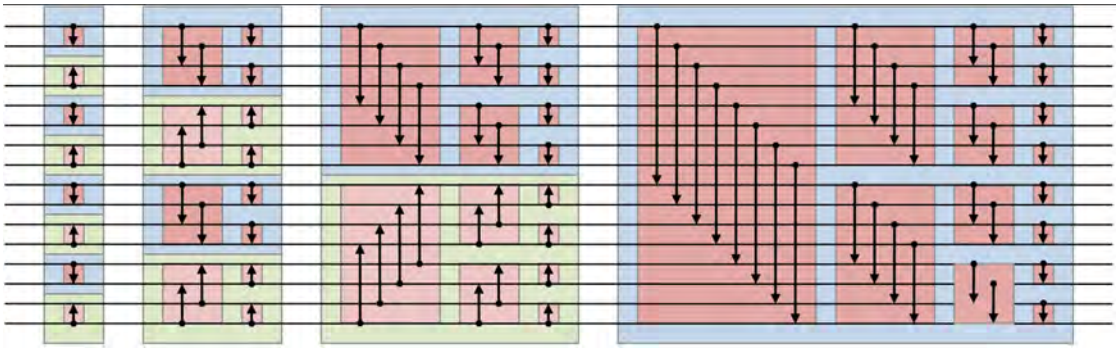

Figure 2.1 An examplary bitonic sorting network for 16 values. The red boxes are the half-cleaners, the blue boxes are sorting in increasing order, the green boxes in decreasing order. Image adapted from Wikipedia [1]

complexity by applying bitonic trees [Gre+06] or minimising the number of kernel launches and access to global memory while allowing arbitrary input lengths [Pet+10].

[1] https://en.wikipedia.org/wiki/Bitonic_sorter

Chapter 3

Concept

3.1 Challenge

The particular challenge in creating and implementing the described point cloud renderer lies in basically two distinct problems. On the one hand, in order to be able to render massive and also dynamic point cloud data in reasonable quality in Real-Time, one has to determine which particular methods and approaches are suitable for this. While there are numerous approaches that tackle these problems, the second problem lies in the fact that most of the approaches rely on additional buffers or more or less special rendering techniques, like e.g. Image-Space rendering, that make them not well suited for a highly complex game engine like Unreal where the manipulation of the rendering pipeline is quite difficult. In other words, one has to investigate which approaches are applicable at all and how to actually implement them in the context of the Unreal Engine. Maybe one has to even change existing techniques or create new approaches or workarounds.

Thus, it is crucial to early identify possible restriction imposed by the Unreal Engine in order to be able to design the general approach of my Point Cloud Renderer and conceptualise the underlying algorithms. Therefore it is necessary to analyse and understand Unreal's rendering pipeline and the involved steps, which I will do in the following.

3.2 Analysis of the Unreal Engine

The Unreal Engine is a state-of-the-art rendering engine and offers all basic functionality to create a full modern game, including physically based rendering, audio, artificial intelligence, collision detection, physics, particle simulation, etc. Since the Unreal Engine is open source, the rendering pipeline can theoretically be analysed in detail. However, given the lack of proper documentation and the high complexity of the code base, I will try to break down the essential concepts to a very short high-level analysis in the following. Since my Point Cloud Renderer will be developed for Desktop platforms and the analysis for all available platforms

would be not feasible in this context, I will focus on the corresponding facts for Desktop platforms.

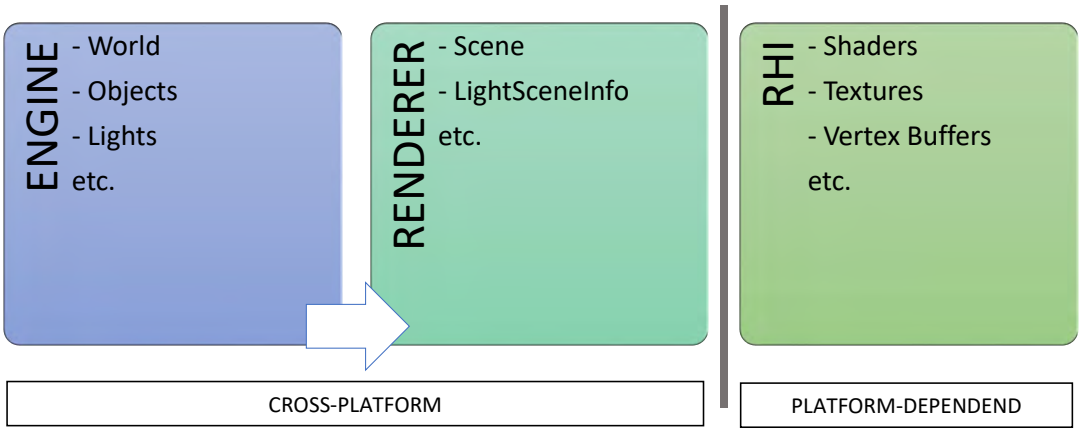

Figure 3.1 High-level view on the rendering part of the Unreal Engine.

Unreal's main rendering pipeline for Desktop platforms is based on a deferred shading approach. The deferred renderer is executed on a separate render thread, running concurrent to Unreal's game thread (for a thorough survey including the listing of all the individual rendering steps, I refer the reader to [Ana17]). While functions on the render thread can be called through various macros in the game thread, the renderer itself uses an abstract interface called the "Renderer Hardware Interface" (RHI) to access the platform-specific graphics API functions. Since the Unreal Engine supports a multitude of different platforms, this RHI has to account for the individual platform specialities and is therefore platform-dependent, while the high-level modules like the renderer and the engine itself can be platform-independent (see fig. 3.1). For Windows Desktop platforms, Unreal supports both Direct3D and OpenGL, using a Cross-Compiler to convert HLSL shader code to GLSL shader code. Since all hardware-related code has to use the RHI, not necessarily all OpenGL/DirectX functions are available.

In the consequence, the rendered point cloud has to be correctly embedded into the deferred renderer's G-Buffer to ensure correct lighting and rendering inside the engine. But given the complexity of the rendering pipeline [Ana17] and the

Figure 3.2 The G-Buffer of Unreal's deferred renderer. Image adapted from [Hof17]

number of buffers (see fig. 3.2), the manipulation of image buffers might in theory be possible, yet practically not feasible. This renders Screen-Space approaches and those that manipulate the frame buffer as basically not applicable. A separate rendering pipeline as proposed by [Gün+13] is therefore not practicable as well.

3.3 Implementation concept

Based on the examined restrictions imposed by the Unreal Engine and the declared aims (see section 1.2), I will now draft a final concept for the rendering approach of the point cloud renderer. The analysis given above concludes that the most promising approach seems to embed the point cloud into the standard rendering pipeline by using Unreal's internal workflow and objects (that is, e.g. meshes, particles, etc.) rather than directly working on the Image buffers. This also has the advantage that already built-in functionality of the Unreal Editor can be used that allows e.g. the easy manipulation of possible rendering or point cloud parameters. Moreover, this leads to the possibility of using Unreal's material shader, which provide a node-based system that allows for easy manipulation of and access to

shader-related functionality and ensures the correct lighting and shading of the point cloud.

To get properly included into Unreal's rendering pipeline, there are now several options. One reasonable option is to try to use Unreal's particle system to render and dynamically change the point clouds. Unfortunately, the current particle system does not trivially allow custom position setting of the points, since they are determined by the internal particle simulation. The remaining and most promising option is now to use a proxy mesh that, in combination with a material shader, builds up the point cloud and handles its rendering. To enable the rendering of dynamic point clouds in Real-Time, the creation and/or update of the individual points should be able to be performed in Real-Time, too.

3.4 Algorithm concept

Now that the restrictions are identified and the basic implementation concept is drafted, the underlying algorithm design can be determined. In order to account for the former set aims (see 1.2), the point cloud renderer will be comprised of several individual blocks that will be precisely defined in the following:

- As splatting/surfels evolved as the quasi-standard for point cloud rendering (as identified in 2.3.2), the point clouds should be also rendered based on, ideally, soft-edged/filtered, non-uniform splats, similar to [Zwi+01].
- To ensure a high visual quality and to handle possible holes in the rendered cloud, a surface estimation, similar to the one in [Pre+12], will be employed that produces continuous surfaces. Since the point clouds might be dynamic, this step should ideally be computable in Real-Time every frame.
- Since the correct depth-ordering of the points has to be ensured and most probable manually computed, a massively parallel GPGPU-based sorting routine will sort the point cloud data according to the distance to the camera. For that, a massively parallel formulation of the bitonic sort, based on the implementation of [Wal15], will be applied.

3.4.1 Splatting

Since the Point Cloud renderer will be included into Unreal's rendering pipeline using the built-in functionality, the basic rendering will be provided by Unreal. Therefore, no special shading or rendering algorithm is needed, which also means that the rendering techniques of the original splatting approaches [Zwi+01] [Pfi+00] [Bot+05] cannot be applied. Yet it is evident that unfiltered rendering primitives do not provide sufficient visual quality (see fig. 3.3), which is why a substitution of the Elliptical Weighted Average (EWA)-filtering has to be found that can be implemented in an Unreal material shader. Since [Zwi+01] proved that the EWA filtering can be simplified to one Gaussian filter step in Screen-Space, the most obvious approximation is to use a blurred ellipsoid as a rendering primitive. This can easily be implemented inside the shader using a simple equation determining the opacity O of a point $\boldsymbol{p_{uv}}$ in texture space (= UV coordinate space) per primitive (with $\frac{\sqrt{3}}{6}$ being the maximum radius of a circle that fits into a equilateral triangle with side length one since the proxy mesh will be comprised of triangle primitives and x determining the strength of the blurring):

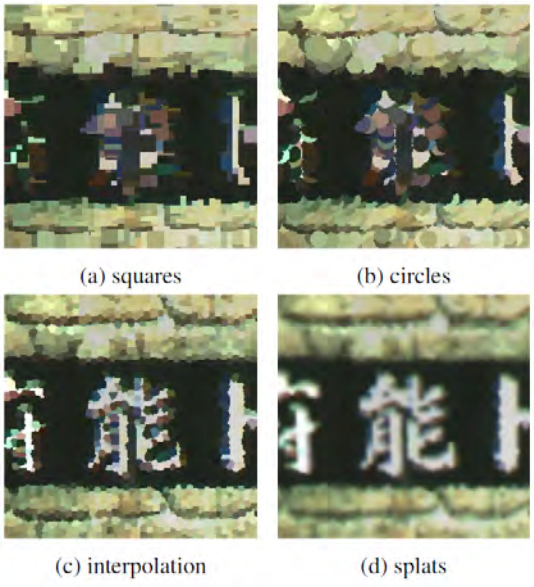

Figure 3.3 Several primitives for point-based rendering. Image adapted from [Sch+15b].

$$O(\boldsymbol{p_{uv}}) = 1 - \left(\frac{||\boldsymbol{p_{uv}}||}{\frac{\sqrt{3}}{6}} \right)^{x} \tag{3.1}$$

For point clouds without available or estimated normal information the splats should also be rendered as camera-oriented billboards. This can be ensured by rotating each triangle primitive to face the camera position by the shader.

3.4.1.1 Depth Sorting

As described in the above section, the EWA-like filtering produces splats with smooth edges and thus semi-transparent areas for every splat. Since the point cloud and the camera position will be fully dynamic and the points of the point cloud will be set by a shader directly on the GPU, it is likely that the depth-ordering of the points will be wrong. The correct ordering of semi-transparent objects is a known problem and several approaches try to cope with this problem (see section 2.3.3). As these approaches mostly use additional buffers or directly manipulate the frame buffer, they are mostly not well suited for the current context of the Unreal Engine and another solution has to be found. I propose to use a sorting algorithm that sorts the points based on the distance to the camera to create a correct depth ordering of the points. Considering the amount of points (which likely will be multi-million), a CPU-based sorting will not be efficient enough to ensure Real-Time performance. Therefore, the sorting will be realised by employing a massively parallel GPU-based sorting algorithm, namely parallel bitonic sort, based on the ideas formulated in [Wal15].

In order to create a sorted sequence of length n from two sorted sequences of length $\frac{n}{2}$, $\log(n)$ comparator stages are required in the bitonic sorting network. The number of comparator stages $T(n)$ of the whole sorting network is given by the recursive function:

$$T(n) = \log(n) + T(\frac{n}{2}) \tag{3.2}$$

With the solution being

$$T(n) = \log(n) + \log(n) - 1 + \log(n) - 2 + ... + 1 = \log(n)\frac{(\log(n) + 1)}{2} \tag{3.3}$$

Since each stage of the sorting network consists of $\frac{n}{2}$ comparators, this yields a number of comparators and thus a work complexity of

$$\mathcal{O}_w(n \log^2 n) \tag{3.4}$$

and a depth and parallel time complexity of

$$\mathcal{O}_d(\log^2 n) \tag{3.5}$$

As stated above, the implemented bitonic sorting algorithm will be similar to the one Walbourn et al. proposed [Wal15], which is a fairly unusual implementation as it is divided into a sorting routine and a matrix transpose operation. The reasons and details of this will be given in the implementation section (4.2.2.2). The general layout of the algorithm is separated into an outer procedure and an inner sorting kernel as can be seen in algorithms 3 and 1.

3.4.2 Surface estimation

In order to create continuous surfaces from the unconnected points of the point cloud, a surface estimation has to be performed. This basically comes down to a neighbourhood estimation, which is a computationally intense task since the neighbours for every point have to be determined. For this task, I perform a surface estimation based on the proposed procedure in [Pre+12], where the surface estimation is performed in Real-Time. The substantial difference is that Preiner et al. perform all the essential steps in Image-Space, whereas I decided to compute the surface estimation for the whole point cloud and on the CPU since the point clouds that were streamed from the kinect were sufficiently small to justify a CPU-based implementation. The basic steps of the surface estimation according to Preiner et al. are the following (per splat):

1. Find the k nearest euclidean neighbours $\boldsymbol{p}_i$ with $\{\boldsymbol{p}_i | i = 1 \dots k\}$ of the current point $\boldsymbol{p}$

2. Compute the mean $\bar{\boldsymbol{p}}$ of the k nearest neighbours:

$$\bar{\boldsymbol{p}} = \frac{1}{k} \sum_i \boldsymbol{p}_i \tag{3.6}$$

3. Compute a fitting plane by computing the covariance matrix $\boldsymbol{M}_{cov}$ of the current point and its k neighbours:

$$\boldsymbol{M}_{cov} = \sum_i (\boldsymbol{p}_i - \bar{\boldsymbol{p}})(\boldsymbol{p}_i - \bar{\boldsymbol{p}})^T \tag{3.7}$$

4. Compute the eigenvector $\boldsymbol{n}$ with the least eigenvalue of the covariance matrix, representing the normal of the current point
5. Compute the radius of the current splat with r_k being the smallest radius that encloses all k nearest neighbours:

$$\bar{r}_{splat} = 2\sqrt{\frac{r_k^2}{k}} \tag{3.8}$$

With complexity (according to [Liu+12]):

$$\mathcal{O}(n \log n) + n \times (\mathcal{O}(k \log n) \times \mathcal{O}(c_3)) \tag{3.9}$$

with c_3 being the complexity of the eigen decomposition of a 3×3 matrix and k being the number of considered neighbours.

For this computation I will employ the PCL library and particularly the included "Eigen" library for the Eigenvector computations and the "FLANN" library [Muj+14] for the nearest neighbour search.

Algorithm 1: ParallelBitonicSortKernel(A, levelMask, level, DTid, GI)

Data: A: an unsorted array with size 2^m,
level: current level of the sorting network,
levelMask: current levelMask,
DTid: Dispatch Thread ID (global Thread ID),
GI: Group Index (local Thread ID within a group)
Result: A: an sorted array

```
// move the data for the current row to shared memory
```
sharedData ← A[CurrentRow]

barrier sync

for *j = iLevel ≫ 1 ; j > 0 ; j ≫= 1* **do in parallel**
  pos1 ← sharedData[GI & ∼j]
  pos2 ← sharedData[GI | j]
  if *(pos1 <= pos2) == (iLevelMask & DTid)* **then**
    result ← sharedData[GI ∧ j]
  else
    result ← sharedData[GI]
  end
  barrier sync
  sharedData[GI] ← result
  barrier sync
end
A[CurrentRow] ← sharedData[GI]

Figure 3.4 Pseudocode of the parallel bitonic sorting kernel for sorting values as proposed by [Wal15].

Algorithm 2: MatrixTransposeKernel(A)

Data: A: a squared matrix with dimension n or an array with size n^2
Result: A: the transposed matrix

sharedData ← A

barrier sync

A[XY] ← sharedData[YX]

Figure 3.5 Pseudocode of the (simplified) matrix transpose kernel for transposing the image matrix as designed by [Wal15].

Algorithm 3: BitonicSortNetworkProcedure(Array A)

Data: A: an unsorted array with size 2^m,
n: the number of elements,
b: the bitonic block size,
t: the transpose block size,
w: the matrix width,
h: the matrix height
Result: A: an sorted array

```
// First sort the rows for the levels <= to the block size
for level = 2 ; level <= b ; level = level * 2 do
    // Sort the row data
    SetOutputBuffer(Buffer1)
    Dispatch[n/b, 1, 1] ParallelBitonicSortKernel(A, level, level)
end

// Then sort the rows/columns for the levels > the block size
// Transpose. Sort the Columns. Transpose. Sort the Rows.
for level = (b * 2) ; level <= n ; level = level * 2 do
    // Transpose the data from buffer 1 into buffer 2
    SetOutputBuffer(Buffer2)
    Dispatch[w/t, h/t, 1] MatrixTransposeKernel(A)

    // Sort the transposed column data
    Dispatch[n/b, 1, 1] ParallelBitonicSortKernel(A, (level / b), (level & ~n)
     / b)

    // Transpose the data from buffer 2 back into buffer 1
    SetOutputBuffer(Buffer1)
    Dispatch[h/t, w/t, 1] MatrixTransposeKernel(A)

    // Sort the row data
    Dispatch[n/b, 1, 1] ParallelBitonicSortKernel(A, b, level)
end
```

Figure 3.6 Pseudocode of the ”outer loop”, forming the bitonic sorting network on the CPU side, dispatching the bitonic sorting kernels as described in alg. 1.

Chapter 4

Implementation

4.1 Basic architecture

One of the basic aspects of Unreal's basic software architecture is it's modularity. The engine is split into and comprised of numerous distinct modules for the different functionalities. If one wants to extend the functionality of the engine, one has to create an own module. For larger and distinct enterprises like my point cloud renderer, it is recommended to create an own plugin. A plugin can contain several modules and is completely separated from the engine code, thus making it easy to distribute and maintain. Unreal has a custom tool called "Unreal Build Tool" that manages the process of building the engine's source code across a variety of build configurations. Therefore, each module has a corresponding build file (with the suffix ".Build.cs") in which the includes and dependencies of the module are specified, which then will be set in the generated Visual Studio project files accordingly by the Unreal Build Tool. This dependencies can be both external libraries and includes and other Unreal modules. The implementation logic can be written in C++ or in so-called "Blueprints" that utilise the Unreal visual node system.

I decided to develop the point cloud renderer as an individual plugin, comprised of three modules, with the main target framework being the Unreal Editor. The first module is the "Third Party" module, that combines and handles all external third party libraries and includes, like PCL, boost, Eigen and the FLANN libraries. Since the external libraries are using custom and standard data types, while Unreal relies completely on own data types, it is important that this module is clearly separated from the Unreal code base. Furthermore, the boost library requires exception handling and Runtime Type Information (RTTI) which were enabled in the Build.cs file by setting the according flags. While Eigen and FLANN are header-only libraries, I built and included PCL and boost as static libraries, because the inclusion of dynamic libraries did not work in an obvious way. The other two modules are the "PointCloudRenderer", providing the "backend" environment for the logic and algorithms with access to the Third Party module and functions, and the "PointCloudRendererEditor" being the public interface to the Unreal Editor, which exhibits the available functionality to public C++ functions and Blueprints.

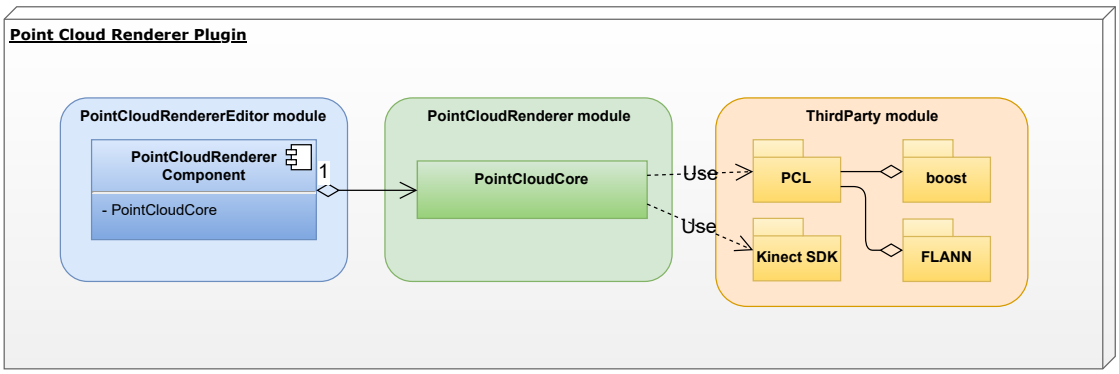

Figure 4.1 High-Level view on the class hierarchy inside the point cloud renderer plugin.

4.2 Rendering Point Clouds in Unreal

As stated in section 3.3, the most promising way to render a point cloud in Unreal is to create a proxy mesh with a corresponding shader. While the creation and/or update of the point cloud should happen in Real-Time, one reasonable option to achieve this might be using geometry shaders. The advantage of using these would lie in the easy and dynamic creation of geometry. Unfortunately, geometry shaders are known to be fairly slow, which renders them inapplicable for a high-performance point cloud renderer for huge and dynamic datasets as in the context of this thesis.

Another idea would be to create the mesh in an ”ordinary” way and try to somehow change the point positions dynamically. In that context, the built-in but rarely used and still experimental ”PaperGroupedSpriteComponent” becomes interesting. This component creates an arbitrary amount of primitives, rendered with custom sprite textures and using instancing. While the rendering of static point clouds is therefore possible and even quite performant (see results chapter), the dynamic update of the points is only possible via individual CPU calls of the built-in methods. Thus, when one wants to update the whole point cloud (which likely will be the predominant case), the update becomes prohibitively slow (see table). This renders the ”PaperGroupedSpriteComponent” approach unsuitable for dynamic point clouds, yet it might be usable for purely static ones.

Point count	**Transform update time (rotation only)**
19k	3 ms
300k	40 ms

Table 4.1 Timings for updating the point transforms in the PaperGroupedSpriteComponent.

Other approaches I tested were the mesh generation via a ”Runtime Mesh Component” and via the ”Primitive Draw Interface”. Unfortunately, both ways proved to be too slow in my tests to properly create or update large meshes during Run-Time.

4.2.1 Rendering dynamic Point Clouds

Hence, another way has to be found to be able to dynamically update the individual points of the point cloud. This is where the ”World Position Offset” input of the Unreal materials becomes interesting. The World Position Offset input allows for the vertices of a mesh to be manipulated in world space by the material. While this is ”traditionally” rather used for ambient animations and similar effects, the world position offset can be also used for further manipulations, even to the degree where whole point clouds can be built up by it. The advantage is here that the data resides on the GPU and also the manipulation of the point data happens directly on the GPU - allowing for high performant rendering of dynamically changing point clouds.

This requires to build up a custom mesh in the first step. I realised this by creating a ”PointCloudMeshBuilder” class that inherits from the ”UInstancedStaticMeshComponent” class which is a component that efficiently renders multiple instances of the same base mesh using instancing. Since this component uses instancing, the whole point cloud is comprised of instances of the base mesh and thus can be rendered using only one drawcall, because the point-specific trans-

forms are stored and manipulated directly on the GPU. The disadvantage of an instanced static mesh being, as the name suggest, static, can thus be avoided. Yet the mesh is restricted to consist of a constant number of elements, because the recreation of the whole mesh is quite expensive. The basic element of the mesh and therefore the shape of one individual point of the point cloud can now be any kind of mesh, theoretically. While quads are often used as primitives (e.g. in the "Potree" system), the point cloud renderer should render points as filtered points (as stated in section 3.4). Therefore I decided on using equilateral triangles as the simplest and thus most efficient primitive and setting the final shape in the corresponding shader/material. The vertex positions are determined as follows to create equilateral triangles:

$$\boldsymbol{v_1} = \begin{pmatrix} x_1 - \frac{a}{2} \\ y_1 - r \\ z_1 \end{pmatrix} \boldsymbol{v_2} = \begin{pmatrix} x_2 + \frac{a}{2} \\ y_2 - r \\ z_2 \end{pmatrix} \boldsymbol{v_3} = \begin{pmatrix} x_3 \\ y_3 + \frac{a}{\sqrt{3}} \\ z_3 \end{pmatrix} \tag{4.1}$$

With a being the triangle side length (was set to 1 for my purposes) and r being the radius of the inscribed circle: $r = \frac{\sqrt{3}}{6} * a$.

Now that the properties of the basic mesh are defined, the corresponding shader that will be assigned to the mesh can be designed. There are now several things that the shader should definitely take care of. These are at least:

- Moving each individual primitive to its proper position in order to build up the whole point cloud
- Defining the final shape, appearance and scaling of the points (e.g. filtered/soft-edged circles, according to eq. (3.1))
- Defining the general rendering properties (Lit/unlit, shadows enabled/disabled, etc.)
- Exposing relevant properties to the user (such as e.g. point size, general scaling, etc.)

However, in order to be able to render the point cloud by manipulating the base

mesh, first the underlying point cloud data has to be transferred to the GPU and made accessible to the shader. I realised this by encoding the point position data into textures which then are getting exposed to the shader. The texture size in Unreal is limited to 8192 by 8192 pixels, restricting the point cloud to have approximately 67 million points at maximum. The underlying texture has to provide four channels of 32bit-float precision in order to properly represent the point cloud data (see fig. 4.2). To now properly encode the data into the textures, the point data has to be available as an array of FLinearColor objects and the point colors have to be encoded as an array of unsigned integers. In my implementation, several public input methods are handling the proper conversion of the input data into the texture-friendly data representations. The data is therefore available twice - in respective buffers on the CPU and encoded into textures on the GPU.

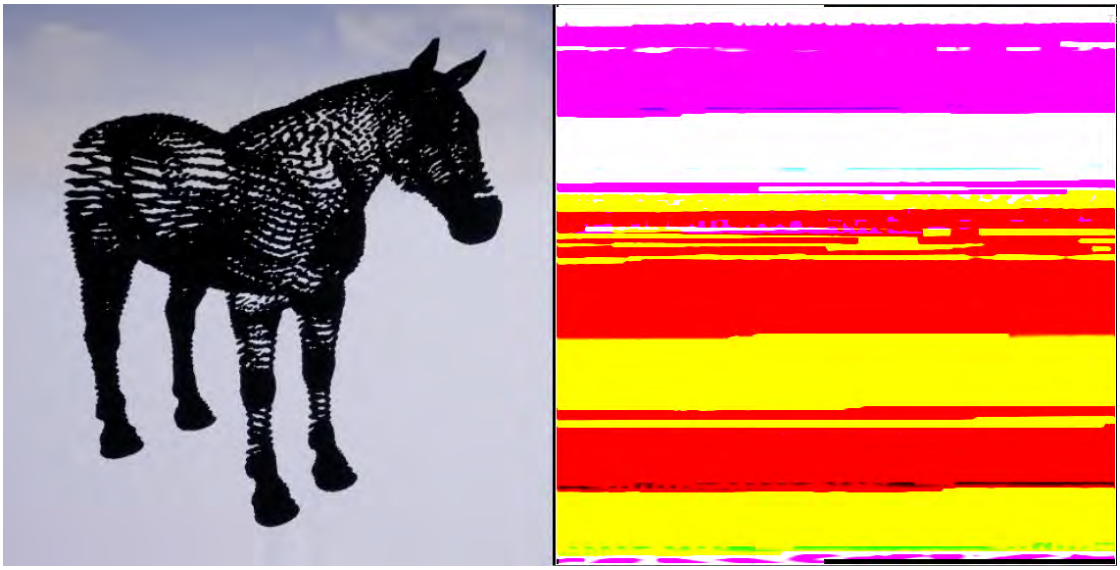

Figure 4.2 A point cloud and its point positions encoded into a 32bit HDR texture. In the texture visualisation, the colours are clamped to be displayable by the screen, thus appearing as fully saturated colours.

Now the shader only has to determine the instance ID of the current instance and fetch the corresponding point position from the according texture. Unfortunately, the "UInstancedStaticMeshComponent" assigns the instance IDs in a random fashion instead of regular numbers. Therefore another mechanic for transferring the instance ID to the shader has to be found. Since the point positions will be

manipulated by the shader anyway, I decided to encode the instance ID directly in the vertex positions of the primitives. This is an easy and efficient way to be able to then fetch the instance ID and also the vertex ID inside the shader. The vertex Z positions for every primitive i are therefore:

$$\boldsymbol{v}_{\boldsymbol{z_1},\boldsymbol{z_2},\boldsymbol{z_3}} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} + \begin{pmatrix} \frac{i}{10} \\ \frac{i}{10} \\ \frac{i}{10} \end{pmatrix} \tag{4.2}$$

The base mesh is therefore built up as a stack of triangles with the vertices according to eq. (4.1) and the z-values according to eq. (4.2). Thus, the vertex and – far more important – the current triangle/instance ID can easily be determined in the shader by simply computing the floor of $v_z * 10$ for every primitive i. It is important to mention that these index helpers must not be too large, because for large values (which will be introduced when one wants to render huge multi-million point clouds) the precision of floating point numbers decreases which results in rounding errors (see fig. 4.3). It is evident that the low index numbers (presumably right area of the image) produce quite exact renderings while the high index numbers (presumably left area of the image) induce severe inaccuracies.

Now that the point position for every triangle can be determined, the shader has to ensure that the triangle is moved to its corresponding position. For this, the triangle is first moved to the given point position. In order to be able to move and scale the point cloud in the editor, the points are then being transformed from local space to world space by applying the object's transform matrix. This transform matrix is being read by the CPU-side and is getting updated and transferred every frame to the shader. Finally, the manipulated Z-indices are reverted to its correct value by subtracting the Z-indices as determined by eq. (4.2) (see fig. 4.4).

In addition to the key process of building up the point cloud by determining the final position for each vertex and moving the vertices to that position, the shader has to account for other things as well (see listing above). This is mainly related to the visual appearance of the points, namely the scaling, shape, colour and

Figure 4.3 Wrong rendering caused by precision loss due to large index helper values.

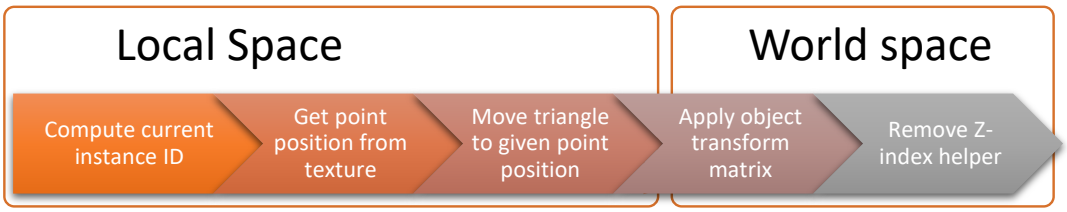

Figure 4.4 The basic transformation process of the triangles to build up the point cloud

orientation. The shape, that is, the soft-edged circle shape, is created by basically applying eq. (3.1) to every pixel of the current triangle and treating this value as the opacity for the current pixel. Since eq. (3.1) ensures that the maximum radius never exceeds the ”physical” limits of the triangle, the points are always rendered as circle-shaped. In doing so, the x parameter is exposed to the user to give control over the ”softness” of the point.

The point colour is now derived by reading from a texture which is encoding the point colours in a similar fashion as the point positions. But in contrast to the

point position texture, a simple 8-Bit RGBA-texture is sufficient to accumulate all point colours. Since the dimensions of the texture matches the point position texture, the same instance/triangle ID is valid for both textures. Alternatively, for example when no distinct point colours are available in the dataset, the colours can be set to a user-defined colour or colormap. Furthermore, the shader ensures that the triangles are always facing the camera (often related to as "billboard rendering") by rotating every triangle towards it.

Finally, the individual size of each point is given by the distance to the camera on the one hand and a global scaling parameter that is exposed to the user on the other hand. Additional user parameters are controlling also the distance-scaling relation, for example the start distance after which the distance-related scaling starts and the falloff of the scaling (from linear to quadratic). The combination of this user parameters exposes a reasonable amount of control over the visual appearance to the user.

For the full schematic shader architecture, see fig. 4.5 and fig. 4.6 for an actual screenshot of the network.

While the shader works well for fully opaque materials, the transparency, which is introduced by eq. (3.1), brings up depth ordering problems, as depicted in fig. 5.15. While transparency ordering is a known and solved problem in rendering, most of the standard approaches - such as depth peeling for example - are using additional buffers (see Order-Independent Transparency in section 2.3.3). But since the manipulation of Unreal's internal Z-buffer is not trivial and may introduce several side effects, another technique has to be found that is not involving further buffer creation or manipulation. As the data sizes likely are very large and the point data is stored as textures on the GPU memory anyway, one very reasonable and promising approach would be to rely on a GPU-based massively parallel sorting routine to reorder the point positions in the according texture according to their distance to the current camera position.

However, before implementing a GPU-based transparency sorting, I had to analyse Unreal's internal sorting procedure of the mesh primitives, because the sorting routine depends on the "original" ordering of the meshes. I tested this by rende-

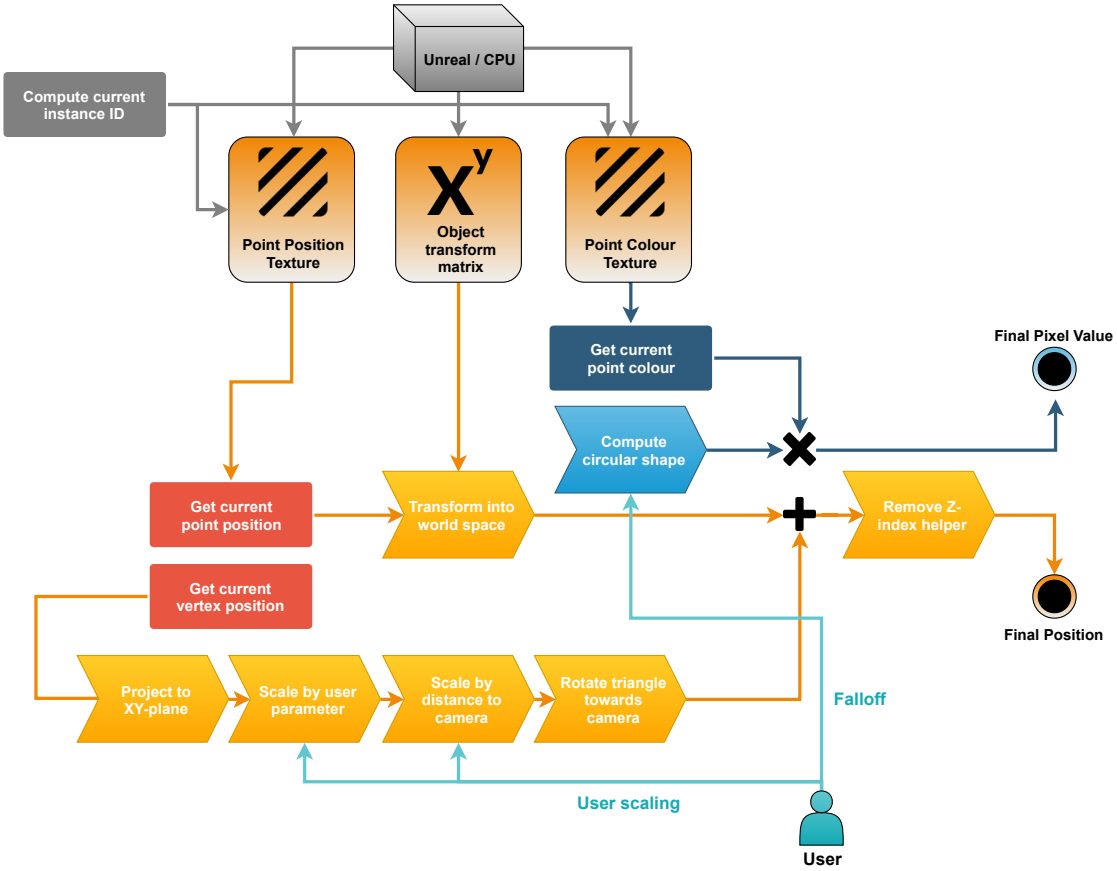

Figure 4.5 The schematic network of the material shader that handles the point cloud rendering.

ring the unmodified triangle stack with a simple transparent material. It turned out that all primitives were sorted correctly which led to the conclusion that the ”World Position Offset” input of the shader introduced the depth ordering problems. Since the triangle stack base mesh is always created along the positive Z-axis (as determined by eq. (4.2)), a constant space was introduced to ensure that the bottommost primitives were the ones that got rendered as nearest to the camera while the topmost are the ones that are farest away. Consequently, the point position texture had always to be sorted in the same manner, with the nearer point positions associated to the lower primitive IDs and vice versa. This leads to the fact that the primitive ID has to be simply remapped to the twodimensional index of the point position texture, starting at the upper left corner of the texture.

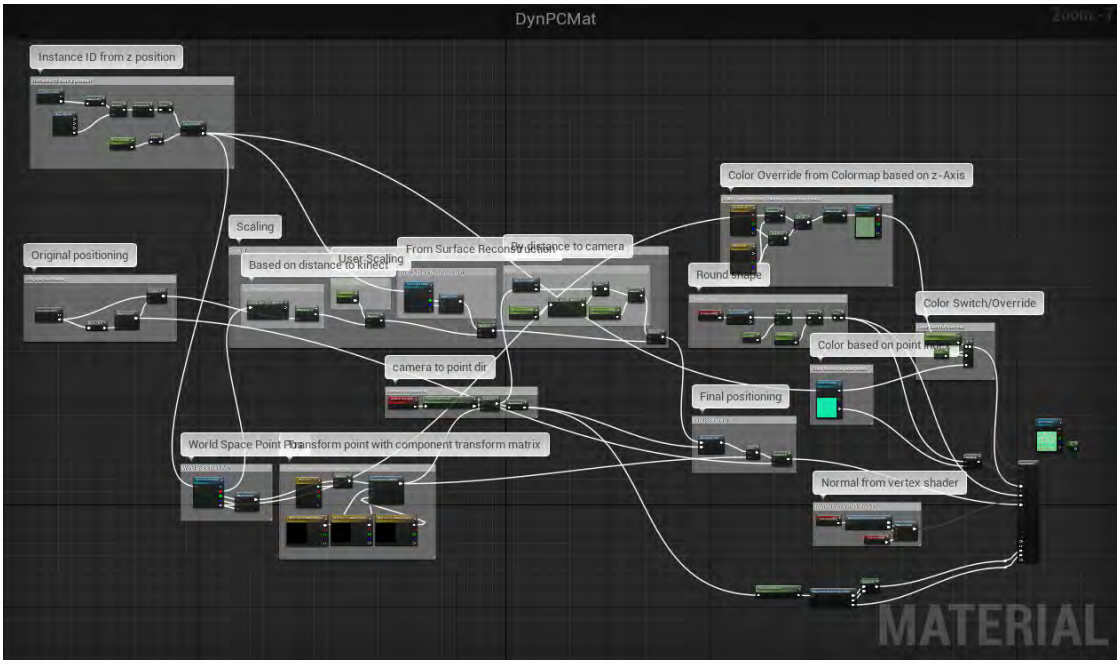

Figure 4.6 The actual network of the material shader.

4.2.2 GPGPU Depth Sorting

Now that it was proved that the massively parallel sorting routine was applicable to the problem, the implementation details has to be sort out. Parallel algorithms can be implemented via numerous different frameworks. The most popular ones are shortly described in the following while evaluating their applicability in Unreal.

4.2.2.1 GPGPU architectures

Parallel programming routines can be implemented via various, also vendor-dependent frameworks. These frameworks offer the possibility to use the GPU's highly parallel architecture to perform also general purpose computations (therefore often also called *GPGPU* for "General Purpose Computation on Graphics Processing Unit"). The key principle is to break down a large computational problem into smaller pieces that can be computed as individual "kernels" in parallel.

Hence, vast speedups compared to a CPU-implementation can be achieved when the underlying problem is well suited for parallelisation. The most well-known and popular frameworks are arguably:

- Nvidia CUDA
- Microsoft DirectCompute
- OpenCL
- C++ AMP

Nvidia CUDA is probably the most popular parallel programming framework with a large base of users, examples and applications, but restricted to CUDA-enabled Nvidia graphics cards. OpenCL represents a very similar approach but generalises over multiple platforms (such as Nvidia graphics cards, AMD graphics cards, but also CPUs and other processor types). While there is a special branch of the Unreal engine called "Gameworks", which includes CUDA in the Unreal engine, I decided against using it, because the textures I used for the point positions were standard Unreal textures and were likely to be not directly usable in CUDA. Vice versa, the proper usage of textures created by CUDA in Unreal shader and materials is very unlikely, too.

While the same applies to OpenCL, C++ AMP was an reasonable option for implementing the parallel sorting algorithm. C++ AMP is an DirectX-based open source library from Microsoft for implementing algorithms on data-parallel hardware directly in C++. It provides easy ways of parallel programming without the need to include external libraries or frameworks. Indeed, there are implementations of parallel sorting algorithms available (see timings in chapter 5). However, the main drawback is that the data has to be available on the CPU, will then be transferred to the GPU and, once the computations are finished, transferred back to the CPU. The employment of C++ AMP would therefore imply two unneccesary copy actions (to the GPU and back), because the processed data on the GPU cannot be directly assigned to the Unreal texture. Since the datasets will likely be very large and copying from CPU to GPU is quite expensive, this double copying (which would have to happen every frame since the camera is completely interactive) would affect the performance severely, which is why I decided to not

use the C++ AMP library.

The most promising option was now to use Microsoft's DirectCompute. DirectCompute is a computing platform which is quite similar to and largely influenced by CUDA and OpenCL with the big difference that it relies on Direct3D and thus is usable by all platforms that are capable of using DirectX 10 at least. DirectCompute now exposes the compute functionality of the GPU as a new type of shader: the compute shader. The contained kernels can be called in parallel by dispatching a multitude of individual threads which are organised in thread groups and dispatches, forming a three-dimensional grid of thread groups. While it is not specifically attached to any stage of the graphics pipeline, it can be used for general purpose computations and, very importantly, has a full inter-operability with all D3D resources. Thus, Unreal-created textures, which are D3D resource on desktop platforms, can be used in Compute Shaders without problems. Consequently, DirectCompute Compute Shaders should be a perfect base for implementing the bitonic sorting in Unreal.

4.2.2.2 Bitonic Sorting in Unreal

In Unreal, the RHI provides access to the handling and creation of compute shaders through various functions. The methods are largely consistent with the standard methods in the Direct3D API. It was therefore possible to translate some publicly available implementations of parallel sorting algorithms to Unreal in a fairly straight-forward manner.

At first, I aimed for implementing radix sort because it is generally considered as the fastest parallel sorting algorithm. While radix sort is indeed quite fast and easy to implement for small datasets, some tricks are involved to adapt the algorithm to larger datasets. After implementing radix sort in a compute shader, I realised that the sorting worked fine within the thread groups, creating sorted chunks/subsets of the dataset, but the merging was the harder problem. As there was no straight-forward solution to this problem, one solution was to employ a bitonic merger. Based on this insight I decided to completely rely on the bitonic

sorting algorithm to have a cleaner solution in the end.

Therefore I adapted Microsoft's publicly available example for bitonic sorting [Wal15] to be usable in Unreal. The bitonic sorting routine is here slightly different from the standard procedures as described in 2.3.4.1, insofar as it is composed of two routines or kernels: a bitonic sorting kernel and a matrix transpose kernel. This is due to the fact that standard bitonic sorting routines have a distinct size limit. In parallel implementations, this is often defined by the size of the available shared memory. In current architectures, the shared memory is still quite limited (for instance, 32 kilobytes per thread group for Shader Model 5.0 in DirectCompute), giving space for 8192 floating point numbers or 2048 float4 vectors per thread group. Because the extensive usage of the very fast shared memory is key to an efficient implementation, the algorithm has to be able to run with this limited amount of memory. The answer to this limitation is the division into this two kernels, or, more precisely, the introduction of the matrix transpose step. The idea behind this is that the bitonic sorting is first only executed on the rows of the texture with one thread group per row. In the second step, the texture (in principle, being nothing else than a matrix), is transposed to be able to sort the columns in the same manner. In the last step, the picture matrix is getting transposed and sorted again, yielding the finally sorted matrix (for details see alg. 3). However, the matrix transpose kernels are getting dispatched in a different thread group layout than the sorting kernel; namely in blocks of the matrix width divided by the constant "transpose block size" (which is equal to the dimension of the point position texture and thus equal to the square root of the number of elements). Furthermore, a double buffer layout was used to avoid read-write conflicts during the transpose operation. In that way, even huge datasets can be sorted efficiently. With a shared memory limit of 32 kilobytes, the maximum number of points in a trivial implementation is therefore 2048*2048 = approximately 4 million points. But since the number of threads per group is limited to 1024, the point limit in the current implementation is 1024*1024 = approximately 1 million points. An important consequence of the restriction of the inputs containing a power-of-two number of elements is that the textures also can only be squares with power-of-two dimensions.

However, I had to modify the original bitonic sorting algorithm from [Wal15] in another way. The original sorting algorithm was designed to sort unsigned integer values, which had to be altered to run on float values, more precisely on three-dimensional vectors with floating point precision (float4). Since the sorting should be based on the distance of the points to the camera, the current camera position had to be accessible to the shader and the central comparison operation had to be modified to use the according distances to the camera. It is important to mention that the camera position has to be transferred into the object space beforehand to account for a possible rotation, translation or scaling of the object that should be sorted. The changes can be seen in alg. 4. The outer network loop as described in alg. 3 and the matrix transpose kernel as described in alg. 2 largely stayed the same, however.

The sorting-related variables, like the camera position and the level and level mask are now transferred to the shader by constant buffer objects, that are renewed if a value is changed. The point positions were a bit more difficult to set; First, I tried to use a ”RWTexture2D”, since the point position data had to be stored in a texture anyway in order to be accessible by the Unreal material shader later on. For this, I created an Unordered Access View (UAV) on the texture. UAVs are a special type of buffer access that allow multiple GPU threads to read from or write to the same buffer simultaneously without generating memory conflicts (in contrast to shader resource views). Unfortunately, it is apparently not possible in Unreal to perform a read operation of a float4 value from an UAV (called a ”Typed UAV Load”), although since DirectX 12 it is explicitly allowed. Even an option to force Unreal to use DirectX 12 did not bring the success, which is why I decided to use an ”RWStructuredBuffer” instead. RWStructuredBuffer are buffers that basically are just arrays of struct

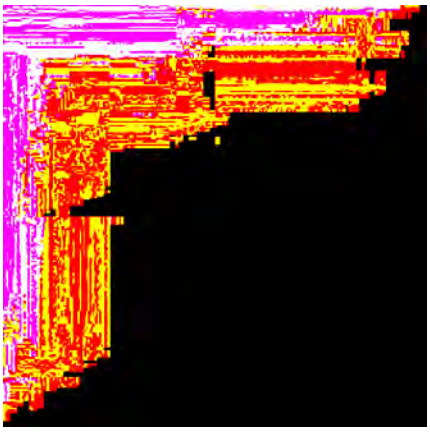

Figure 4.7 Wrong sorting results caused by read-write conflicts during the matrix transpose step.

types. But although the RWStructuredBuffer was also initialised with an UAV, it was still suffering from read-write conflicts during the matrix transpose step, visible as flickering in the final texture and wrong sorting (see fig. 4.7). This was caused because the sync barrier inside the code only synchronises the threads within a group, but not all thread groups globally. But even the declaration of the buffer as "globallycoherent" did not work out. Thus, I had to implement a similar double buffer solution as in the original approach, where the result of the sorting is stored into two buffers simultaneously. The transpose kernel now reads from the second buffer and stores the result in the first one, which is again used as input for the next sorting stage. The thread group layout, however, remained the same by treating the structured buffer similar to the texture but with the according index corrections. Thus, the whole computation happens in shared memory and the buffer accesses, which are stored in global memory, are mainly performed as coalesced memory access (only the transpose kernels are dispatched in a block-wise fashion). Still, the (doubled) buffer accesses and the transfer of the point data every frame to the structured buffer are likely to be the bottleneck of the sorting routine, although the latter point obviously cannot be improved for dynamic point clouds.

Because raw texture objects cannot be directly assigned to Unreal material shader, a simple pixel shader is needed to "convert" the raw texture reference (a FTexture2DRHIRef object) into an usable format (an UTexture) in the end. This is realised by simply copying the values from the output texture of the compute shader to a standard Unreal texture, which is then getting exposed to the material shader that, in turn, gets assigned to the former created base mesh. Both the PixelShader and ComputeShader plugin are based on a template from the community[1].

[1] https://github.com/Temaran/UE4ShaderPluginDemo

Algorithm 4: BitonicSortKernel(A, levelMask, level, DTid, GI)

Data: A: unsorted Array of all the point positions with size 2^m,
level: current level of the sorting network,
levelMask: current levelMask,
DTid: Dispatch Thread ID (global Thread ID),
GI: Group Index ("flattened" index of a thread within a group)

Result: A: sorted Array, according to the distance of each point to the camera

```
// get the current camera position in object space from Unreal
```
camPos ← CurrentCamPos

```
// move the data for the current row to shared memory
```
sharedData ← A[CurrentRow]

barrier sync

for *j = iLevel ≫ 1 ; j > 0 ; j ≫= 1* **do in parallel**
  pos1 ← sharedData[GI & ∼j]
  pos2 ← sharedData[GI | j]
  dist1 ← distance(pos1, camPos)
  dist2 ← distance(pos2, camPos)
  `// Put invalid point data at the end`
  if *pos1 is invalid* **then**
    dist1 ← −dist1
  end
  if *pos2 is invalid* **then**
    dist2 ← −dist2
  end
  if *(pos1 <= pos2) == (iLevelMask & DTid)* **then**
    result ← sharedData[GI ∧ j]
  else
    result ← sharedData[GI]
  end
  barrier sync
  sharedData[GI] ← result
  barrier sync
end
A[CurrentRow] ← sharedData[GI]

Figure 4.8 Pseudocode of the modified parallel bitonic sorting kernel for sorting points according to the distance to the camera, based on the original algorithm design by [Wal15]; see alg. 1.

4.3 Point cloud processing

Now that the rendering part of the plugin is clear, the basic point cloud processing module has to be designed in order to be able to provide point cloud data (from files or from the Kinect) and process them, for example by performing a surface estimation. As stated before, I employed the PCL library and the included FLANN, Eigen and boost libraries for this. All basic processing is handled by and implemented in the "PointCloudCore" class.

I decided on separating the point cloud renderer into two plugins: the one which mainly handles the rendering of the point clouds (called the "GPUPointCloudRenderer" plugin) and another one which provides the input data to the rendering plugin with the help of PCL and the Kinect SDK (called the "PointCloudRenderer" plugin). I decided on this to decouple the processing and the rendering part – on the one hand, to generalise the rendering, because it works as well with point cloud data from other sources than a file or the Kinect (for instance, from procedurally generated content), as long as the provided datatypes are correct. On the other hand, because the processing part was quite heavy as it utilises PCL, which in turn uses FLANN and boost. This results in more than ten thousand source files. Furthermore, installing can be tricky, because the static PCL libraries are dependent on the system, more precisely they are bound to a certain Visual Studio and Windows SDK version which makes the plugin not very portable. To further support modularity and flexibility, also the compute shader, that handles the parallel bitonic sorting, was excluded to a separate plugin.

However, I implemented proper interfaces (via the Unreal node system) to easily and efficiently transfer data between the two plugins. The involved workflow is described in detail in section 4.4 and the UML diagrams of the main classes are shown in figures 4.14 and 4.13.

In order to be able to render point clouds even without the GPUPointCloudRenderer plugin, I included a point cloud renderer that only renders static point clouds with the help of the built-in PaperGroupedSpriteComponent (as mentioned in section 4.2). That component renders static point clouds with reasonable perfor-

mance in an easy way. The advantages are that the point clouds can be rendered with arbitrary materials since there is no special shader needed as opposed to the procedure of the GPU-based point cloud renderer as described above. Furthermore, the PaperGroupedSpriteComponent allows to compute collisions with the individual points.

Furthermore, when one wants to render dynamic point clouds from the Kinect, a special "Kinect2Grabber" class is handling the conversion from the raw Kinect buffers to PCL cloud objects. To gain access to the Kinect buffers, the Kinect2 SDK is needed and has to be included in the Unreal project via a DLL. The code of the "Kinect2Grabber" has been adapted from [Sug14]. However, since Unreal is using a different coordinate system than the Kinect, I had to transform the points from one coordinate system to the other. This consists of a rotation of minus 90 degrees on Unreal's Z-Axis and 90 degrees on its X-Axis. Furthermore, the Kinect coordinate system works in units of centimeter, while Unreal is using meter units. The respective transformation and scaling thus evaluates to the following mapping of the axis:

$$x_{unreal} = -z_{kinect} * 100$$
$$y_{unreal} = x_{kinect} * 100$$
$$z_{unreal} = y_{kinect} * 100$$

4.3.1 Surface estimation

As stated in section 3.4, the point cloud renderer should employ a surface estimation algorithm to produce visually continuous surfaces. The chosen surface estimation procedure of Preiner et al. [Pre+12] is implementable with the help of PCL in a quite straight-forward fashion, with the exception that the nearest neighbour search is not performed in parallel on the GPU and in Screen-Space, but via the help of the FLANN library on the CPU and in object space. For the

implementation, see alg. 5.

Algorithm 5: ReconstructSurface(cloud, k, &normalsOut, &scalingsOut)

Data: cloud: a pointer to a PCL point cloud object,
k: the number of nearest neighbours taken into account per point,
kdtree: a PCL kdtree object for the nearest neighbour search
Result: normalsOut: the array of the computed point normals,
scalingsOut: the array of the computed point/splat sizes/radii

```
if !cloud then
    return false
end
if !kdtree then
    ConstructKdtree()
end
foreach point in cloud do
    // Use FLANN for an approximate nearest neighbour search
    nearestNeighbours[0…k], nnDistances[0…k] ← kNNSearch(kdtree, point, k)
    // Use PCL for computing covariance matrix and eigenvectors
    computedNormal ← pcl::computePointNormal(nearestNeighbours[])
    r_k ← maxElement(nnDistances)
    computedRadius ← 2 * sqrtf(pow(r_k, 2) / nearestNeighbours.size())
    // Set a simple maximum radius
    if computedRadius > 10 then
        computedRadius ← 10
    end
    normalsOut[point] ← computedNormal
    scalingsOut[point] ← computedRadius
end
return normalsOut, scalingsOut
```

Figure 4.9 Pseudocode of the surface estimation procedure as used in [Pre+12] (see 3.4.2), with CPU-based nearest neighbour search by the FLANN library instead of the proposed GPU-based search.

4.4 General overview

In this section, the functionality of the individual parts of the point cloud renderer were explained in detail. To now gain a proper and full understanding of the workflow, I will give a final summary by explaining the individual steps that are involved when one wants to render a static point cloud from a file or a dynamic point cloud, for instance coming from a Kinect. This explanations will be supported by screenshots of the corresponding node networks in the Unreal Editor and detailed UML diagrams of the main classes. Screenshots of the resulting point clouds will be given in the results chapter (see chapter 5).

I will first concentrate of the involved steps when one wants to process and render a static point cloud from a file (see fig. 4.10). This involves only the PointCloudRenderer plugin (from now on also referred to as "static point cloud renderer"). First, the general properties for the point cloud are set by the "Set Point Cloud Properties And Update" node. To be able to also change the order of the steps in the workflow, the underlying methods are implemented in a way that allows as much flexibility as possible. Thus, the order of the steps could be changed as well. Nonetheless, after the setting of the properties, a point cloud is read from a file by the PCL library by using the "Render Static Point Cloud From File" node. The resulting point cloud object can now be further processed, for example by performing a surface estimation via the "Reconstruct Point Cloud Surface" node.

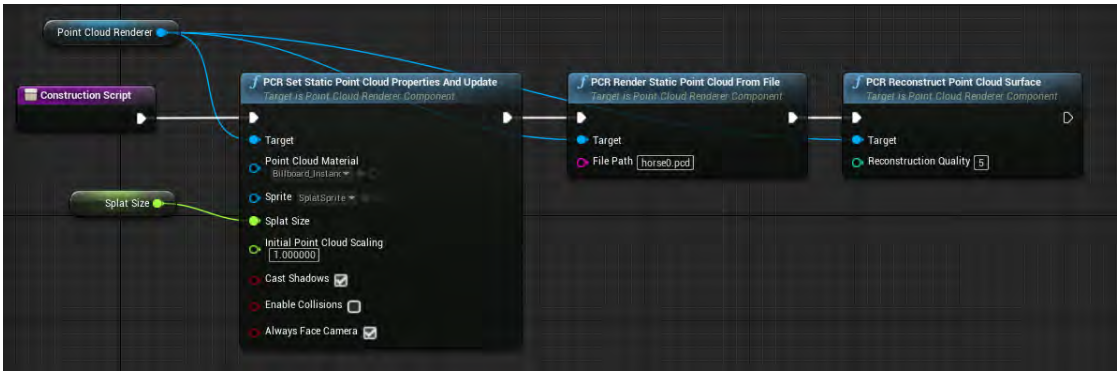

Figure 4.10 The node network to perform a surface reconstruction on a static point cloud from file and to render it with the static point cloud renderer.

It is now also conceivable to render a static point cloud dynamically via the GPUPointCloudRenderer plugin (from now on also referred to as "dynamic point cloud renderer") (see fig. 4.11). To achieve this, the PointCloudRenderer has to read the according data from the file as usual, but instead of rendering it directly, it provides the raw data to the GPUPointCloudRenderer plugin. This can be done via the "Get Current Point Cloud Data / Read From File" node, which provides the data as arrays of LinearColors and uint8, so that the GPUPointCloudRenderer can directly use the data for the textures without further conversion steps via the "Set/Stream Input" node. To avoid unnecessary copying, the methods are taking the variables by reference. In that way, only the data pointers have to be transferred, resulting in high efficiency. However, the user has to ensure that the data the pointer points to is accessible long enough to be encoded into the textures. Since the point cloud itself is static, it is sufficient to provide the input once at the beginning (the "BeginPlay" event), while the parameter setting and the sorting can be performed every frame.

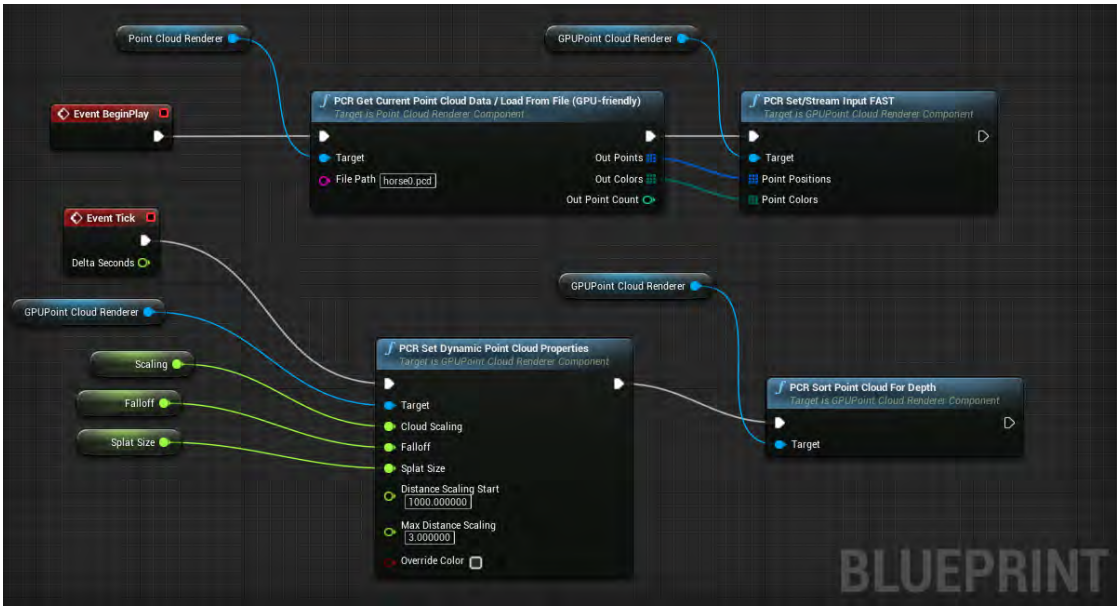

Figure 4.11 The node network for rendering a static point cloud file with the dynamic point cloud renderer.

Figure 4.12 now shows the nodes that are necessary to render a fully dynamic point cloud from the Kinect. Here, the Kinect point cloud, which is getting converted

from the Kinect buffers every frame by the Kinect2Grabber class, is again provided to the ”Set/Stream Input” node in the respective texture-friendly format. Further steps, such as parameter setting or sorting can be appended as well.

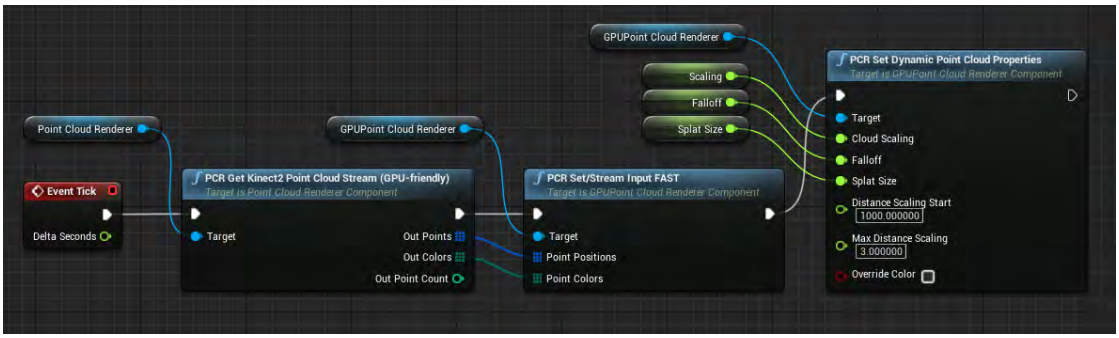

Figure 4.12 The node network for rendering point cloud data coming from a Kinect.

Because the underlying data is represented as PCL clouds and the methods are designed to be as modular as possible, the individual nodes and connected steps can be set very flexible. Nevertheless, the general process or workflow the point cloud data follows until it is getting rendered, is quite straight-forward and can be described as follows: In the first step, the data gets provided from a certain source, for example from a point cloud file, which is getting read by the according PCL method. This yields a PCL cloud object, which is then getting converted to a usable format, in the best case as arrays of FLinearColors and uint8. These are transferred to the GPUPointCloudRenderer plugin by reference and are then getting encoded into the respective textures. Subsequently, a mesh is getting created with a sufficient amount of primitives and an instance of the shader as described above is getting created and assigned to the mesh. The created textures, in turn, are getting assigned to the shader. Optionally, the textures are getting sorted by the compute shader, employing the parallel bitonic sorting. During rendering, the point positions and colors are getting read by the shader, which moves the meshes triangles to their destined positions, building up the whole point cloud.

Alternatively, the rendering can be taken over by the static PaperGroupedSprite-Component renderer, where every point is added as an individual sprite with the

given parameters and transform to the component.

Finally, the UML diagrams of the main classes in the figures 4.14 and 4.13 depict the underlying methods and architecture of the plugins.

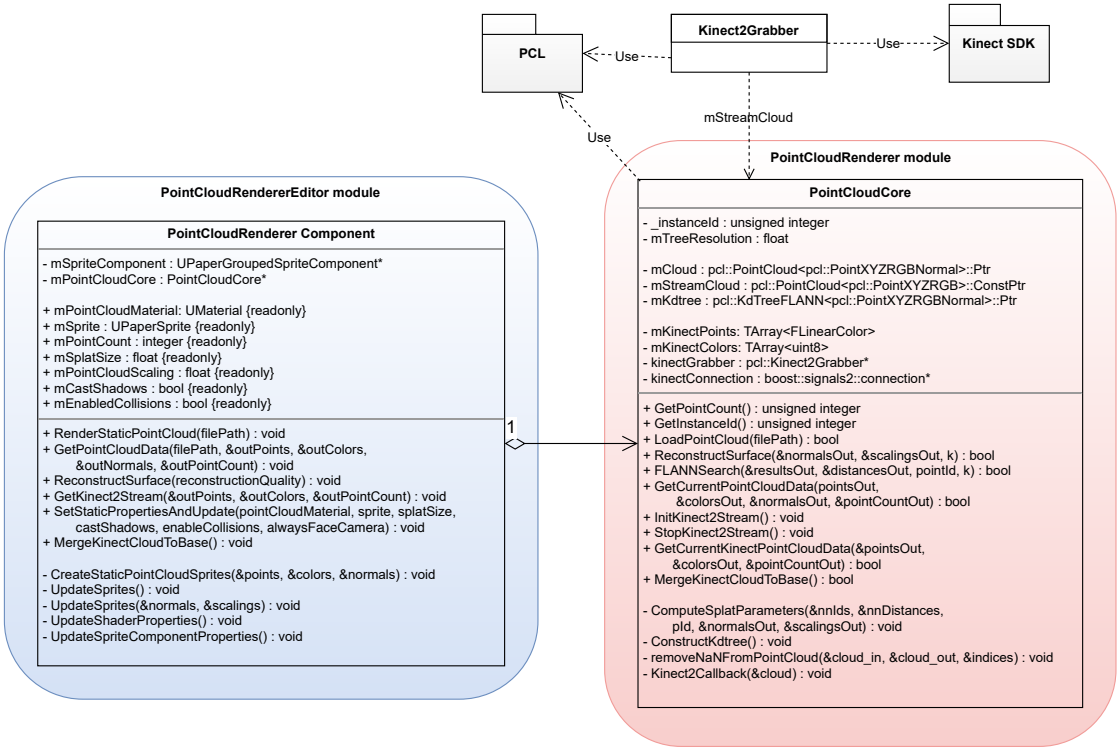

Figure 4.13 Simplified UML class diagram of the Point Cloud Renderer plugin.

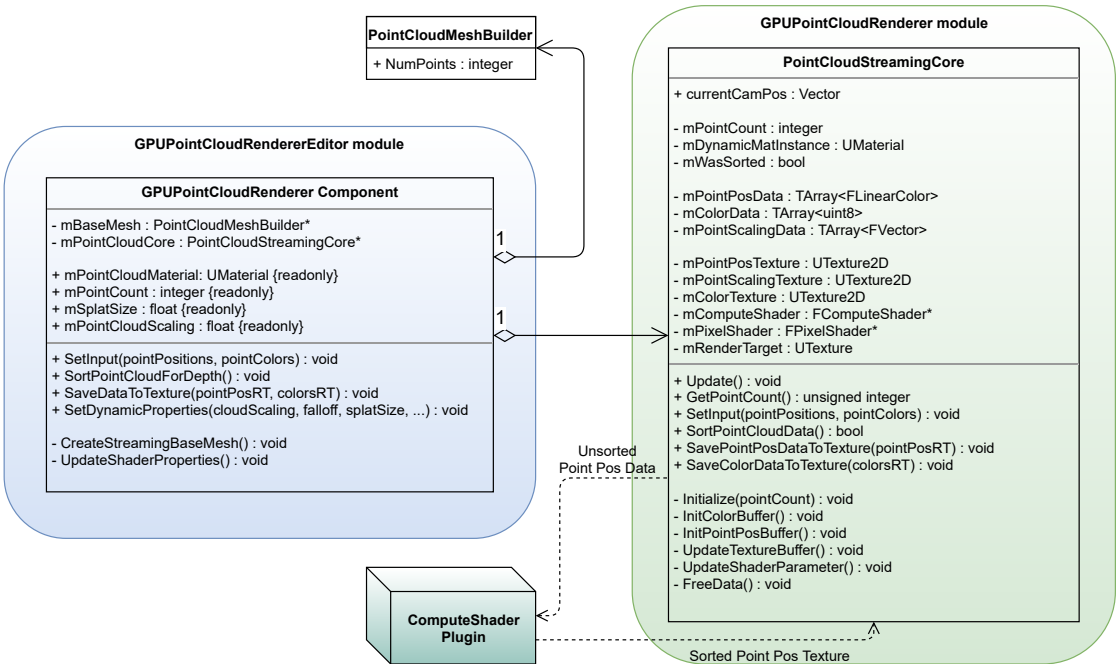

Figure 4.14 Simplified UML class diagram of the GPU Point Cloud Renderer plugin.

Chapter 5

Results

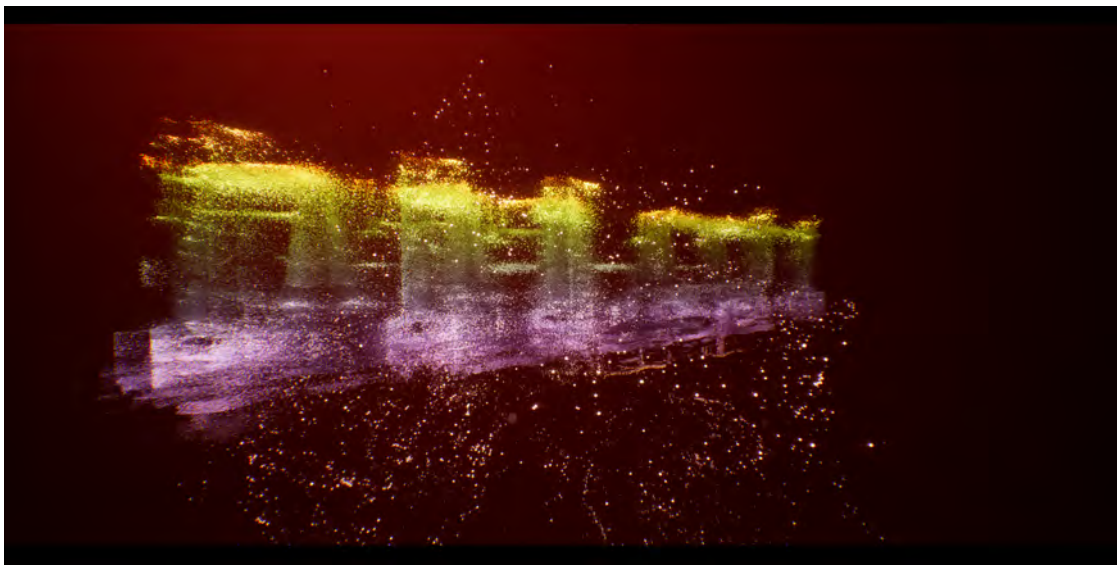

Figure 5.1 Screenshot of a final point cloud rendering of the kettles data set from the Point Cloud Renderer Plugin (with additional image effects and distortion).

Before it is possible to assess the outcome of this thesis by determining if and how good the initial aims of the thesis were met, the results of the implemented approaches have to be presented in detail. Since efficiency and quality were the key principles of the point cloud renderer (see section 1.2), this will happen in both quantitative and qualitative ways. To evaluate the efficiency, timings of all computationally expensive steps will be given, dependent on relevant parameters. These values will be compared to equivalent other approaches, where possible. To evaluate the visual quality of the renderings, numerous screenshots of renderings will be given which will, as well, be set in contrast to comparable other approaches, provided by the Open Source "CloudCompare" software [EDF18]. Since the point cloud renderer is comprised of distinct individual parts, the results will be structured according to these individual parts by adopting the structure of the former chapter.

To do so, the tests were made with different datasets. As static point clouds, I employed five datasets of different sizes, which are illustrated in the following table:

Model name	**Point count**	**File format**	**Colored?**	**Source**
Bunny	35.947	PLY	no	[Lev05]
David	52.566	PCD	no	[Lev09]
Kettles	2.453.890	PCD	yes	private
Trimble	8.484.455	PCD	yes	private
Bremen City	15.896.874	PCD	no	[Nüc+16]

Table 5.1 The different point cloud datasets used for testing the point cloud renderer.

The visual "ground truth" of these models is given by simple splat-based renderings without any further processing by the CloudCompare software (see fig. 5.2).

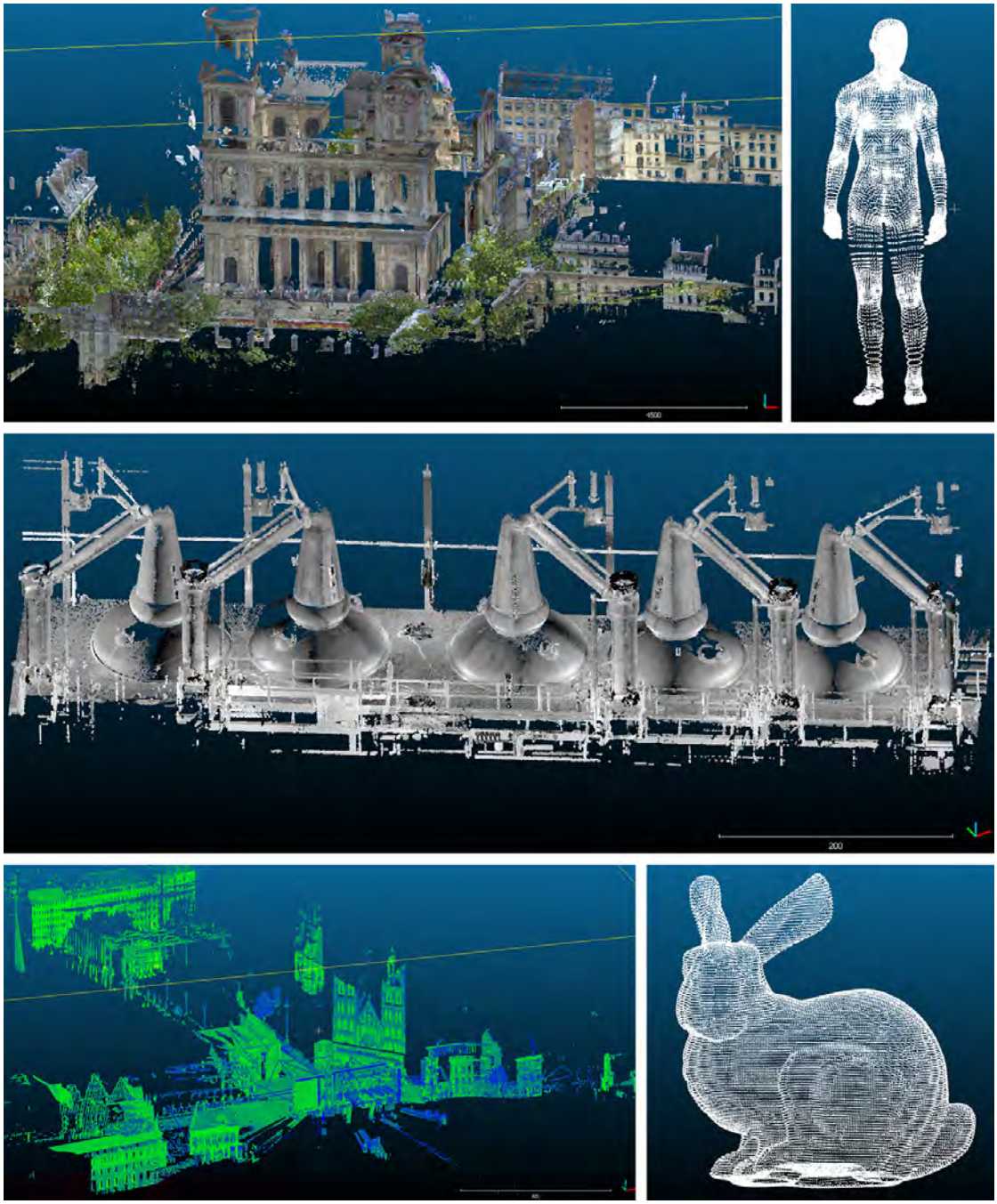

Figure 5.2 The ground truth renderings as produced by the "CloudCompare" software. Datasets from top to bottom, left to right: house, david, kettles, bremen city, bunny.

5.1 Renderings

5.1.1 Static Point Cloud Renderer

The renderings of the point clouds as produced by the PointCloudRenderer for static point clouds are depicted in the following pictures:

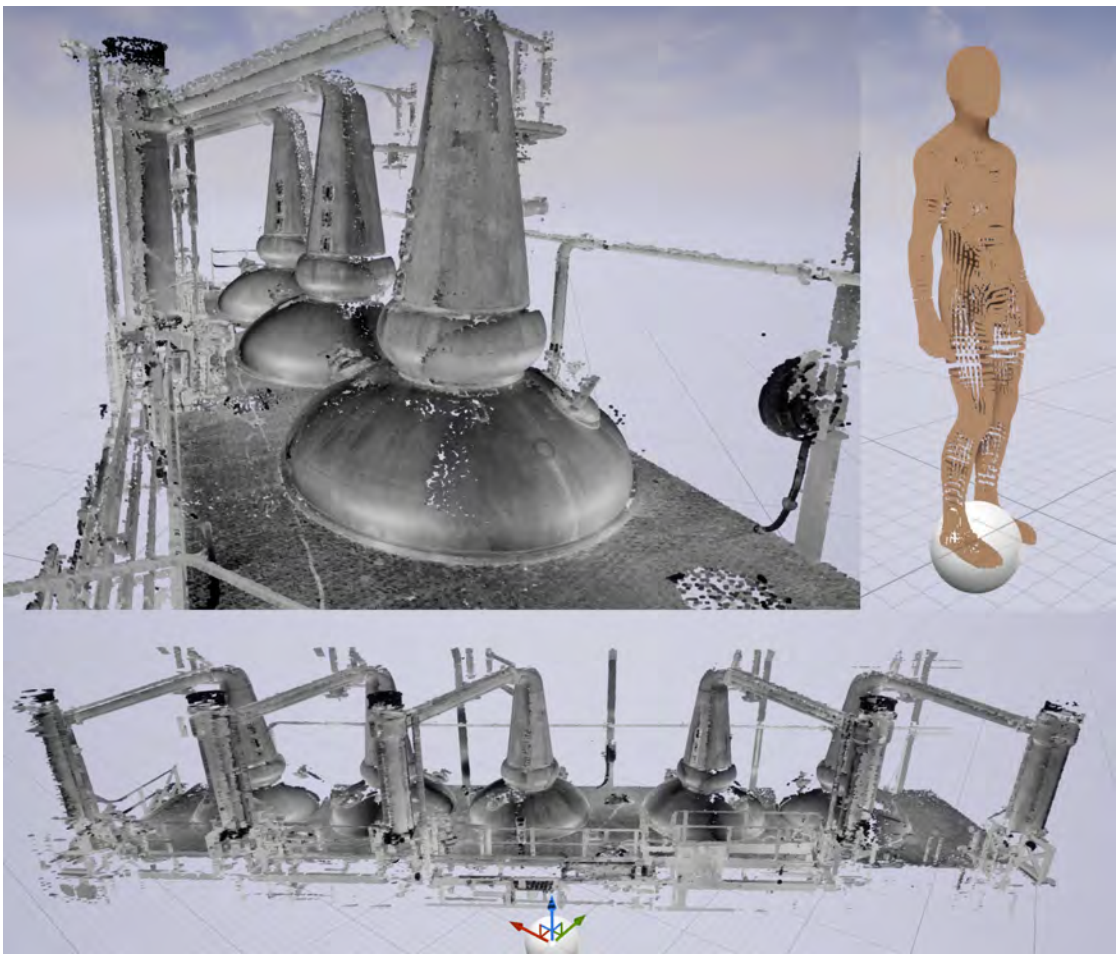

Figure 5.3 The renderings of the datasets that can be rendered by the static point cloud renderer without surface reconstruction and without the bunny dataset (for renderings of the bunny dataset see e.g. fig. 5.5). The other datasets could not be rendered on the testing machine due to not enough available memory.

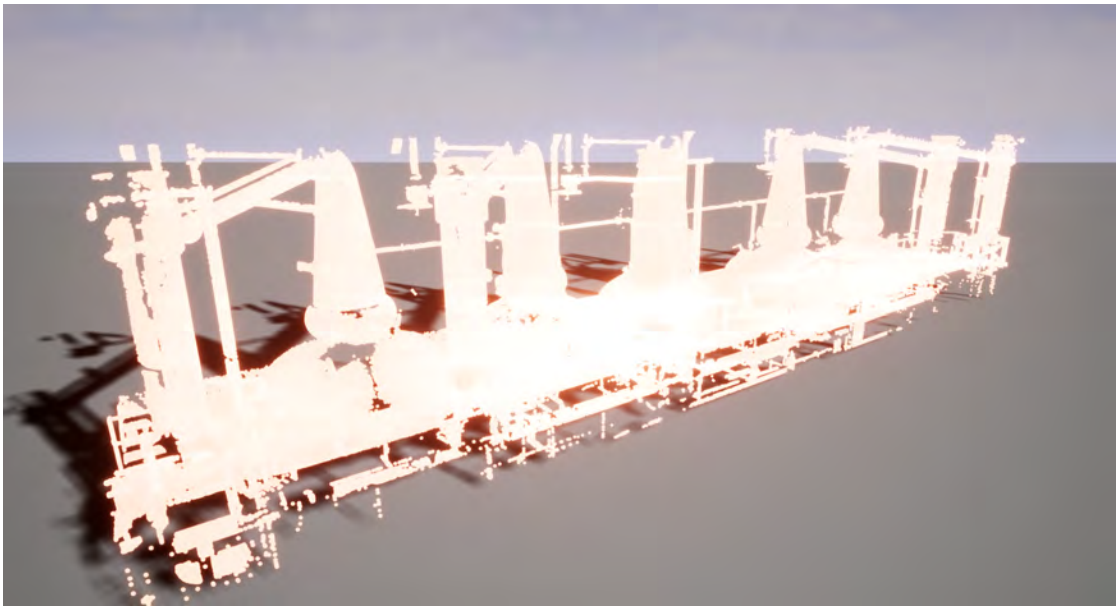

Figure 5.4 The static point cloud renderer also easily allows for advanced materials and shadows.

5.1.2 Surface estimation

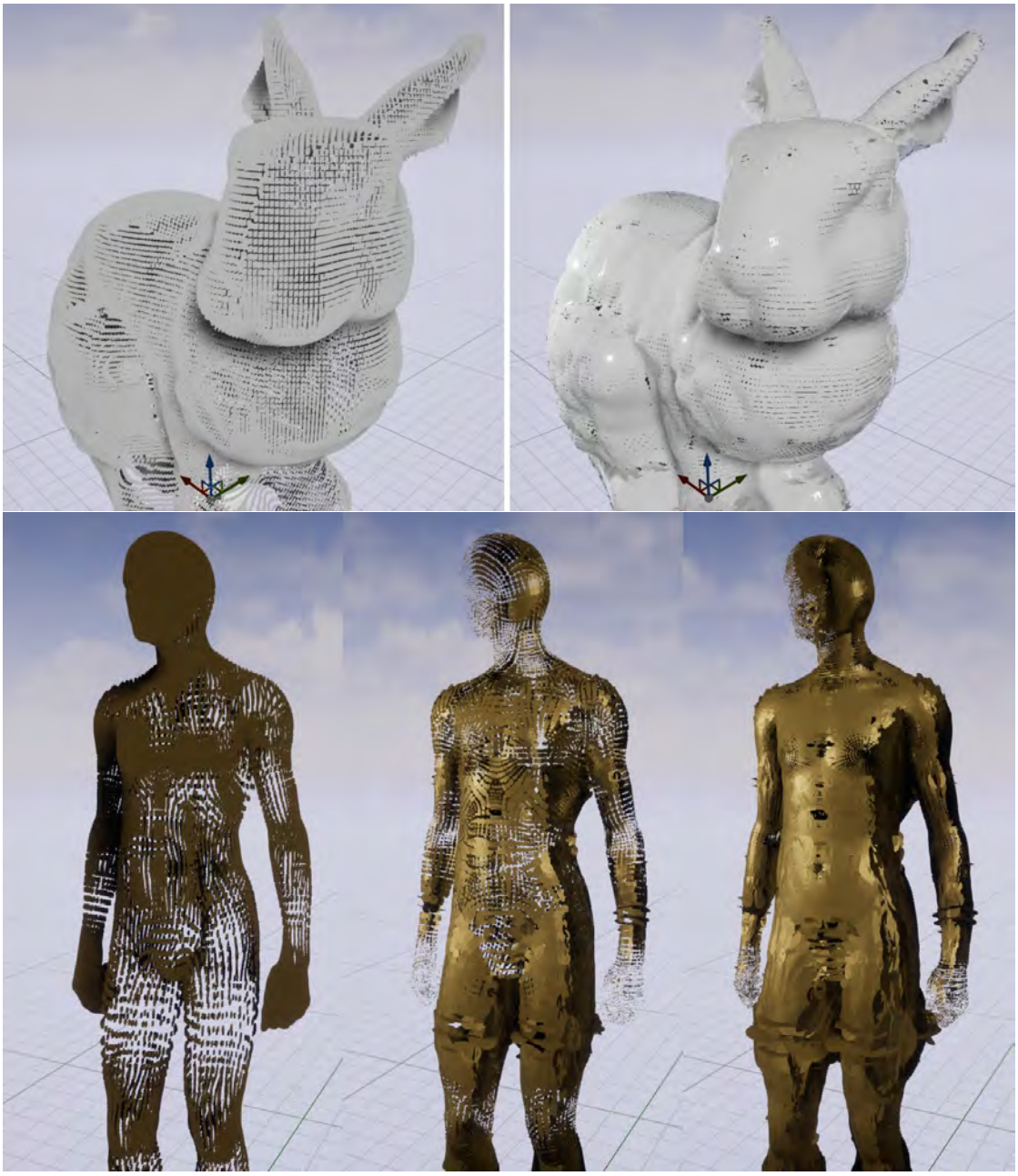

Figure 5.5 The bunny and david datasets rendered by the static point cloud renderer without (left images) and with surface estimation. It can be seen that the points can be properly shaded with the computed normals, resulting in visually more appealing renderings and a better impression of the underlying object. Furthermore, the surface estimation tries to compensate for differences in point densities, although this seems to be problematic for areas with high point density (see e.g. the hands and face of the david dataset).

Figure 5.6 The renderings of the bunny dataset by the static point cloud renderer with surface estimation based on k=2, 3, 4, 5, 7 and 11 neighbours (from left to right). It can be seen that for this dataset, a surface estimation based on k=4 yields a reasonable result and that there is almost no visual improvement between k=7 and k=11.

5.1.3 Dynamic Point Cloud Renderer

As stated already, the user can influence the rendering of the dynamic point clouds through several parameters that are exposed to the user. The most important ones are the splat size and the falloff variable, which controls the ”softness” of the splat (see eq. (3.1)). The interaction between falloff parameter and the splat size and their influence on the rendering can be seen in fig. 5.7. It is visible that a small splat size and a bigger falloff value tend to produce good visual results (first row of the image matrix).

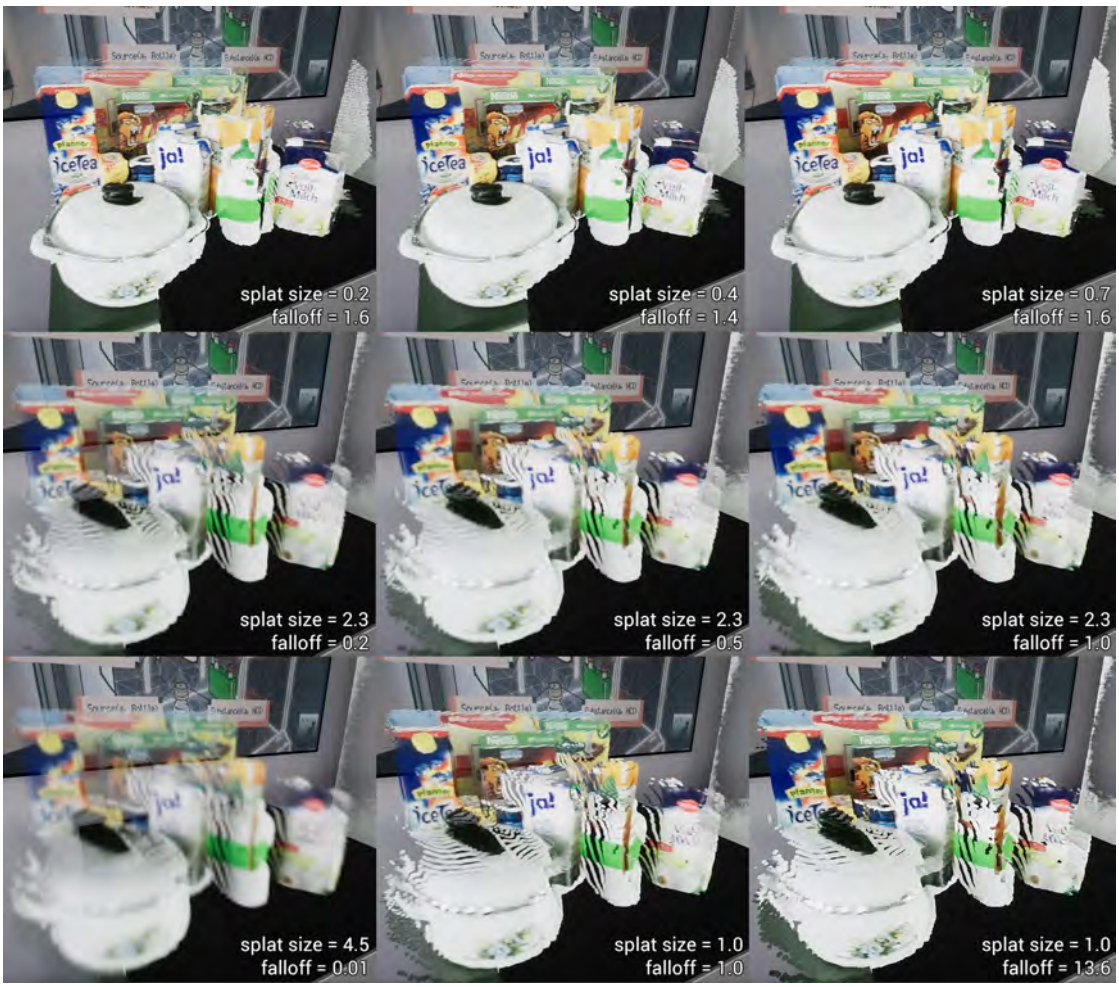

Figure 5.7 Renderings of a kinect point cloud with different user parameters by the dynamic point cloud renderer. A small splat size and a bigger falloff value seem to produce good visual results (see first row).

The renderings of the point clouds as produced by the GPUPointCloudRenderer for dynamic point clouds are depicted in the following pictures:

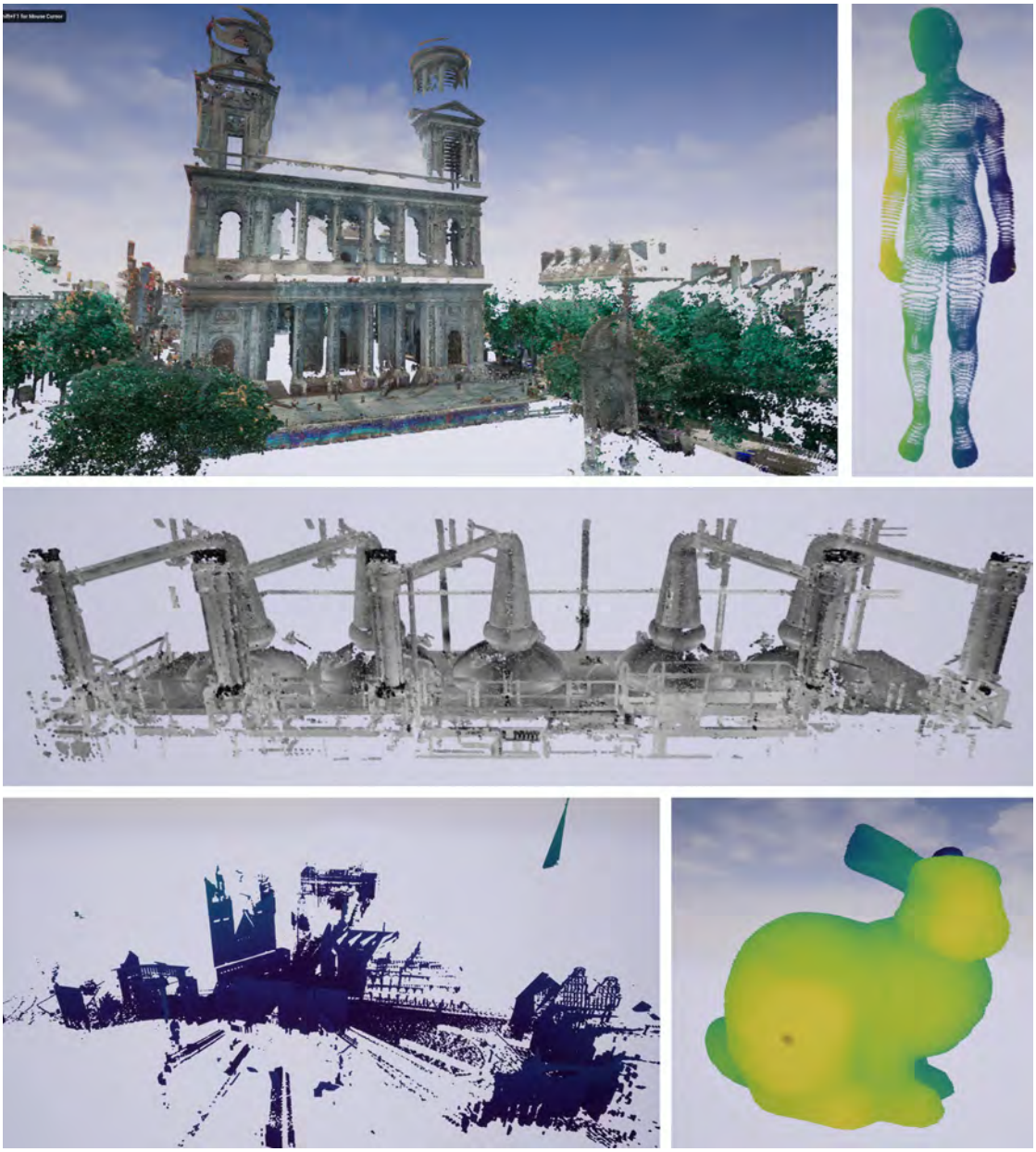

Figure 5.8 The renderings of the given datasets as produced by the dynamic point cloud renderer and without surface reconstruction. The datasets without colour information were rendered with a custom colormap for better visualisation.

Figure 5.9 The Trimble dataset as rendered by the dynamic point cloud renderer (top image) compared to the rendering by the CloudCompare software (bottom image).

5.1.3.1 Kinect rendering

While the GPU-based approach allows for dynamically changing point clouds and the presented datasets were indeed rendered successfully, so far they were not changing over time. To now test the ability to render fully dynamic point clouds, a point cloud based on a Kinect datastream is rendered. This could have several practical applications, like for example in collaborative virtual environments (CVEs) or on-the-fly environment scanning. The latter is particularly relevant for robotics, where a movable robot often has to gain a virtual representation of its environment to avoid collisions etc. To test the suitability of the dynamic point cloud renderer for such an use case, I combined the dynamic point cloud coming from the Kinect with a positional tracking which was realised by a HTC Vive (see fig. 5.10). The test area was the kitchen environment which is used by the Institute of Artificial Intelligence of the University of Bremen to test their robots. The advantage of this environment is on the one hand that a HTC Vive tracking system is already installed and available. On the other hand, the kitchen environment is also precisely reproduced inside the Unreal Engine as a virtual 3D model, thus providing a form of virtual ground truth the point cloud can be compared to (see fig 5.12).

Figure 5.10 The hardware setup for the test: A HTC Vive tracker mounted to a Microsoft Kinect, thus combining Real-Time tracking and 3D scanning.

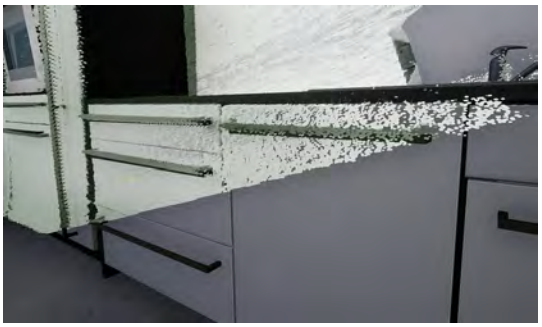

Figure 5.11 The precision of the system is sufficient to provide a good virtual representation of the environment.

Figure 5.12 The real kitchen testing environment of the Institute of Artificial Intelligence of the University of Bremen (upper left), the precise virtual model of the same environment inside the Unreal Engine (bottom left) and the same environment with additional point cloud snapshots, complementing the virtual representation (right images). One can easily see now that the table in the virtual environment is at the wrong location and not matching with the real-world object.

The system was implemented in such way that the point cloud stream was displayed inside Unreal the whole time with the world space position determined by the tracking system. It was then possible to take the currently displayed point cloud and add it to a ”global” point cloud. Thus, the user could easily ”scan” real objects or even the whole environment by adding ”snapshots” of the point cloud stream to the resulting point cloud. In that way, the user can decide which objects are scanned and also in which detail (by making less or more individual snapshots). Two resulting point clouds are depicted in figures 5.12 and 5.13. In doing so, the virtual representations can either be completed or also checked for precision or correctness (see for instance the table in fig. 5.12). Of course, the precision of the tracking system plays also a major role in the resulting accuracy

of the system, which yet seems to be visually sufficient as depicted in fig. 5.11.

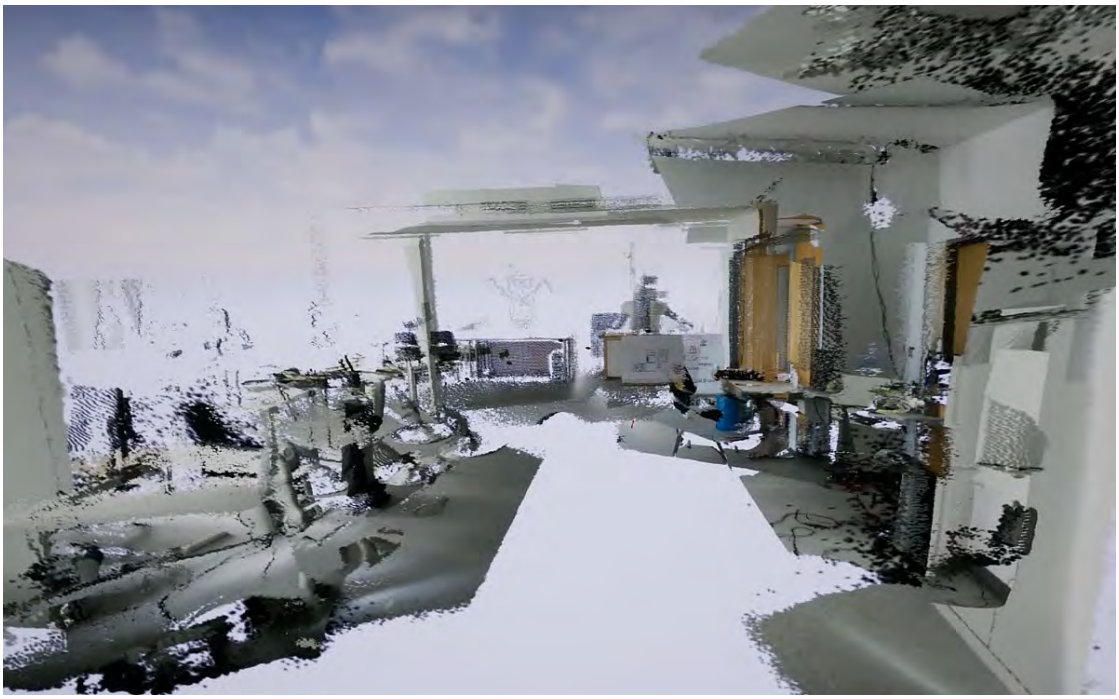

Figure 5.13 A rendering of the resulting point cloud, comprised of several "snapshots" of the Kinect stream.

5.1.3.2 Parallel Bitonic Sorting

The parallel bitonic sorting reorders the point positions in the point position texture according to their distance to the camera. Fig. 5.14 compares a sorted and an unsorted point position texture. This visual impact on the resulting rendering is evident and depicted in fig. 5.15. It can be seen that the sorting works in general, however, the ordering of the points remain incorrect for certain areas (see difference image).

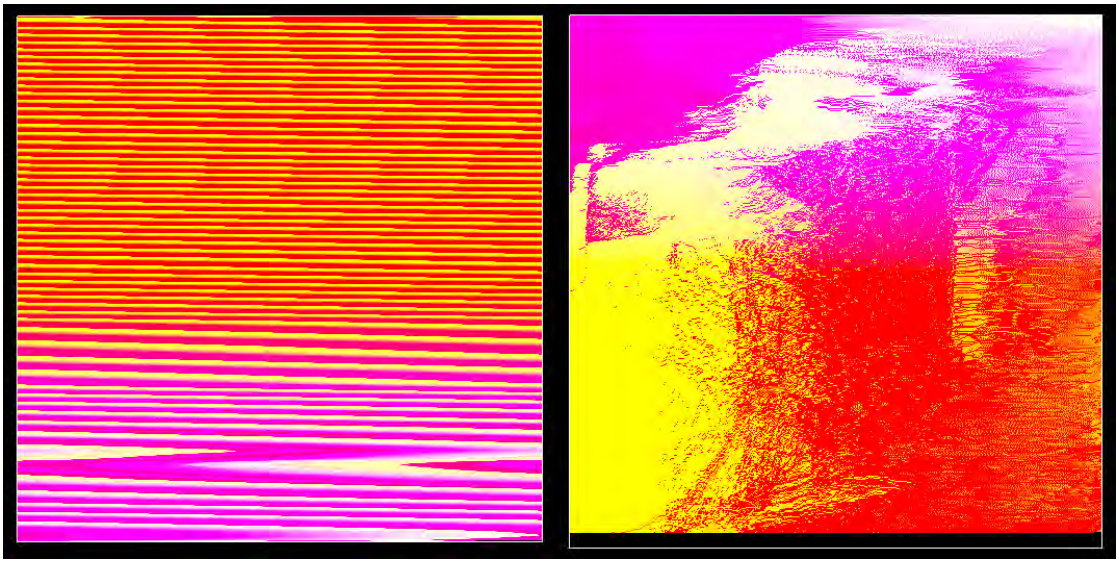

Figure 5.14 A unsorted point position texture (left) and a sorted one (right). Due to the restriction of the sorting algorithm to have a power-of-two problem size, the sorted point position gets created with the next larger power-of-two dimensions, leaving a black area on the bottom of the texture.

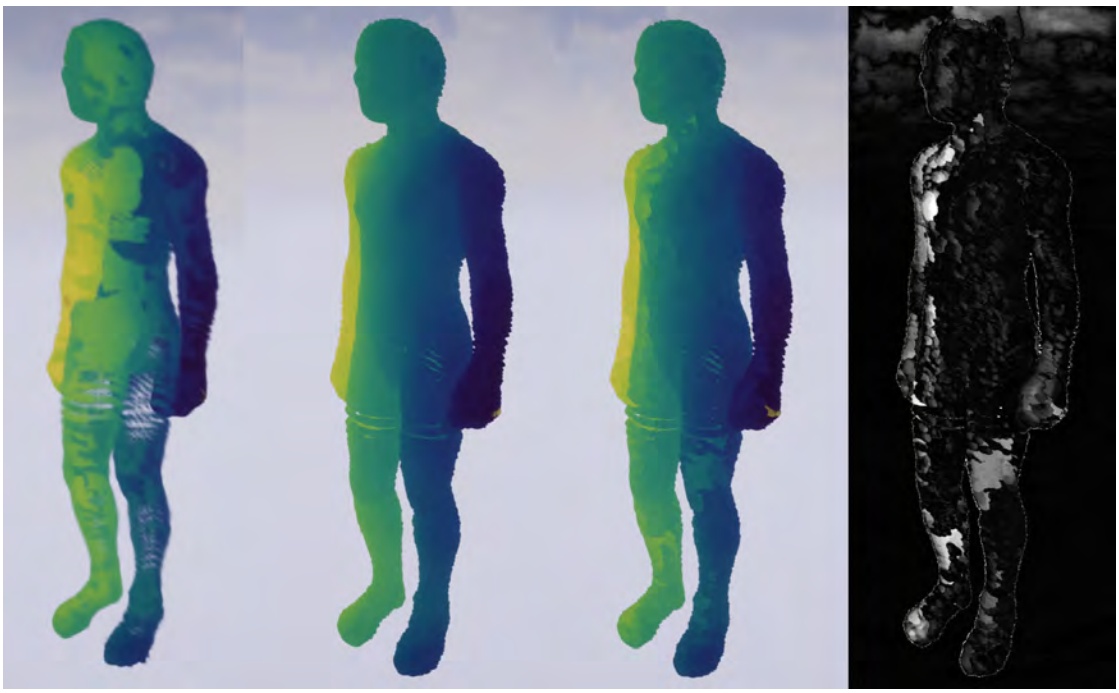

Figure 5.15 Comparison of renderings with an unsorted point position texture, a non-transparent material (ground truth), a sorted point position texture and the difference between the ground truth and the rendering with the sorted point position texture (from left to right). It can be seen that the sorting vastly improves the rendering, yet some areas remain incorrect.

5.2 Timings

The timings were recorded on a machine running Windows 10 64bit with an Intel i5-4670k CPU running at 3.40GHz, 16GB of RAM and a Nvidia GeForce GTX 970 with 4GB VRAM. The tests were conducted with PCL in the version 1.8.1, boost in the version 1.64 and the Unreal Editor in the version 4.18. The resolution of the renderings inside the Editor was 1205 by 695 on high settings.

5.2.1 Processing

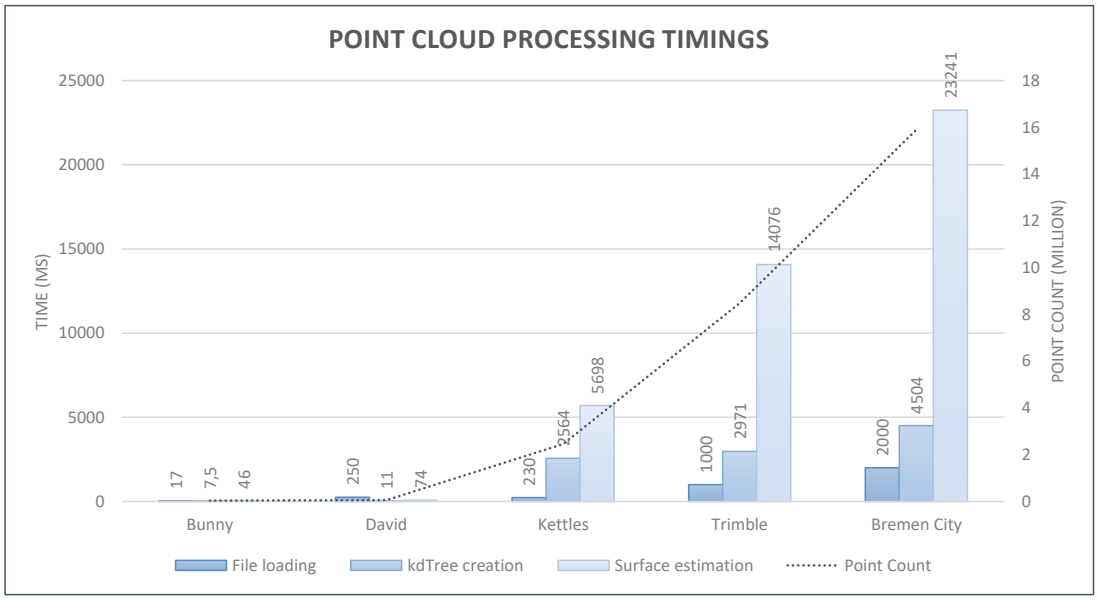

Figure 5.16 The timings for file loading, kdTree creation and surface estimation (the sum of all nearest neighbour searches of all points performed by the FLANN library, with $k = 5$) for each respective dataset.

Figure 5.16 presents the timings for file loading, kdTree creation and surface estimation (nearest neighbour search) for all datasets. The plot shows that the surface estimation time seems to increase linearly with the point count and therefore quickly becomes prohibitively slow. Consequently, a surface reconstruction

during Run-Time is not feasible for standard-sized point clouds with the current naive, CPU-based implementation. However, a surface estimation is feasible for static point clouds, even for large-sized ones.

5.2.2 Rendering

Table 5.2 contrasts the framerates for the indicated datasets for both the static point cloud renderer and the dynamic point cloud renderer. Both renderers were set up to not cast or receive shadows and without including lighting (”unlit”). The shader for the dynamic renderer was set to ”masked” mode with a splat size of 1, which is why the following framerates are not containing the parallel sorting of the point positions. The datasets ”Trimble” and ”Bremen City” could not be rendered by the static point cloud renderer, as it crashed due to not enough available memory. Furthermore, the framerates of the static point cloud renderer differed a lot, depending on the used shader, distance to the cloud and various other parameter why it is difficult to define a distinct framerate.

Modelname	**Pointcount**	**Static PC Renderer**	**Dynamic PC Renderer**
Bunny	35.947	92 FPS	120+ FPS
David	52.566	92 FPS	120+ FPS
Kettles	2.453.890	32 FPS	120+ FPS
Trimble	8.484.455	(not possible)	50 FPS (19,8 ms)
Bremen City	15.896.874	(not possible)	3 FPS (280 ms)

Table 5.2 The framerates of the static and the dynamic point cloud renderer for the given static point cloud datasets with splat size = 1 and without shadows, lighting and the parallel point position sorting.

Figure 5.17 illustrates the data as well in a plot:

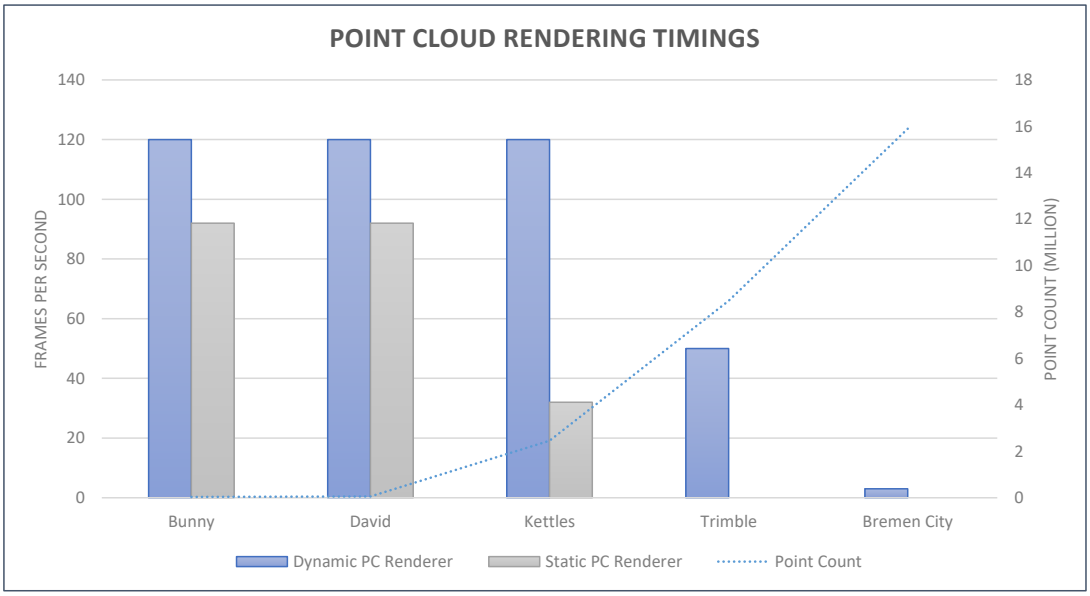

Figure 5.17 The framerates of the static and the dynamic point cloud renderer for the given static point cloud datasets with splat size = 1 and without shadows, lighting and the parallel point position sorting.

Since the framerate of the dynamic point cloud renderer is dependent on the splat size as well, figure 5.18 illustrates the framerates in relation to the splat size.

Since the fully dynamic point clouds coming from the Kinect were fairly small (512 by 424 = 217.088 points), the Kinect point clouds could always be rendered with the maximum framerate of 120 FPS.

Since there are just very little similar approaches to implement a point cloud renderer inside a high-end graphics engine, just little scientific and documented data is available which makes it quite difficult to properly compare the current results to other approaches. One of the few is [Fra17], who reports for his implementation of a point cloud renderer inside the Unity3D engine slightly better framerates (74-97 FPS for 10 Million points with a resolution of 1920 by 1080), but on a better machine. [Pre+12] is reporting for his approach, which was not implemented inside a graphics engine, timings of approximately 20 ms for the mere rendering of a point cloud containing 10 million points, but with a slightly less powerful testing machine. This timings are very similar to my point cloud renderer, which

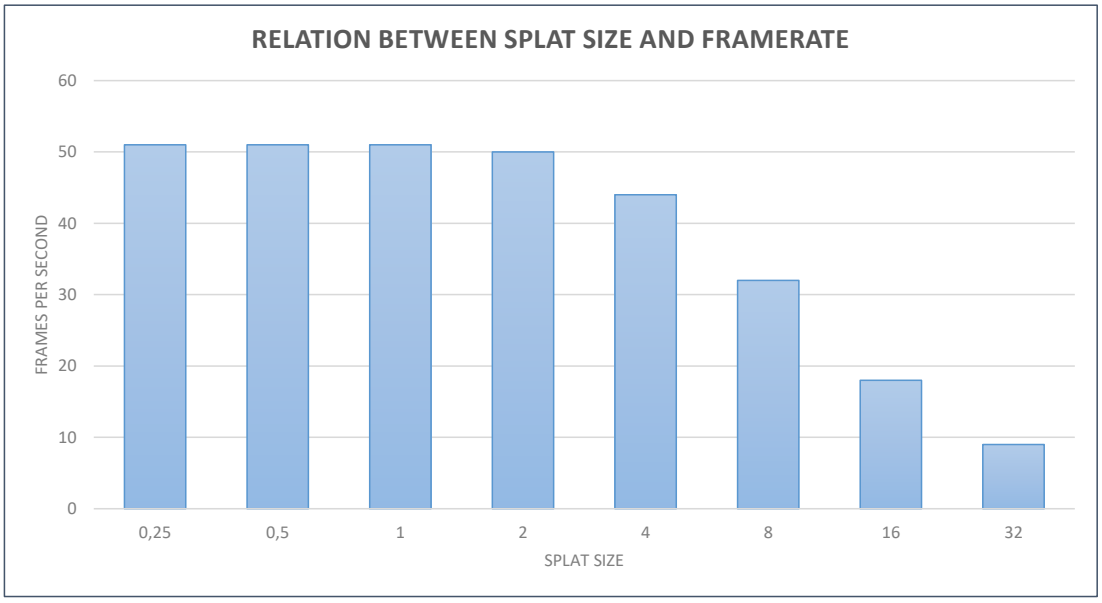

Figure 5.18 The framerates of the dynamic point cloud renderer with the Trimble dataset in relation to the splat size.

needs 19,8 ms for the similar sized Trimble dataset. The timings including the surface estimation are remarkably higher, at approximately 95 ms with k=10 for the nearest neighbour search. For static point clouds, my point cloud renderer thus produces renderings with global surface estimation at significant higher framerates with the drawback of the precomputation time for the nearest neighbour search.

5.2.2.1 Parallel Bitonic Sorting

The timings of the in this thesis presented parallel bitonic sort implementation are now set in contrast to comparable approaches. In particular, these are two publicly available C++ AMP implementations of both radix sort and bitonic sort and the official CUDA implementations from the CUDA8 samples. The timings are given in detail in the table below and are presented visually in figure 5.19 (with the ”Bitonic Sort Compute Shader” being the in this thesis presented approach). All timings were recorded on the same machine but with the difference that the in this thesis presented implementation operates on signed float values while all

other approaches operate on unsigned integer values. Moreover, all of the tested approaches are limited to a problem size of 1024^2 due to limited shared GPU memory or the maximum number of threads in a thread group (1024 on DirectX 11) except of the CUDA implementations. The timings of my bitonic sort compute shader were measured with the "Nvidia Nsight Graphics" software.

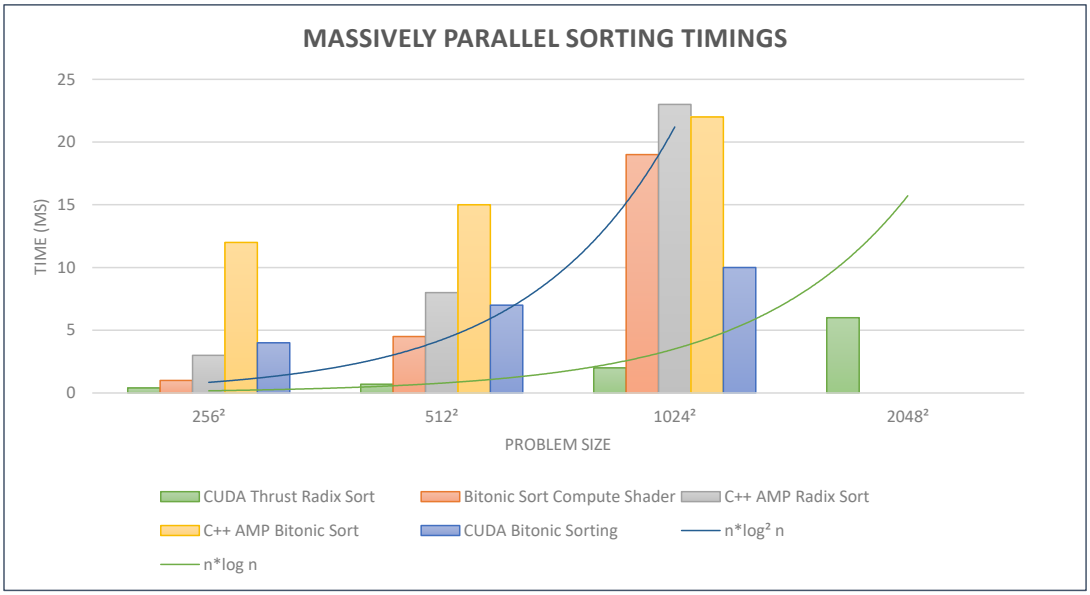

Figure 5.19 The timings for the in this thesis presented compute-shader-based parallel bitonic sorting implementation and the timings of comparable approaches. Please note that my algorithm operates on signed float values while all other approaches operate on unsigned integer values. The green and blue lines indicate the expected trend given by the computational complexity of the radix sort ($\mathcal{O}(n \log n)$) and the bitonic sort ($\mathcal{O}(n \log^2 n)$).

Problem size	**Bitonic Sort Compute Shader**	**C++ AMP Radix Sort [Bas13]**	**C++ AMP Bitonic Sort [Bha11]**	**CUDA Bitonic Sorting**	**CUDA Thrust Radix Sort**
256^2	1 ms	3 ms	12 ms	4 ms	0.4 ms
512^2	4.5 ms	8 ms	15 ms	7 ms	0.7 ms
1024^2	19 ms	23 ms	22 ms	10 ms	2 ms
2048^2	-	-	-	-	6 ms

Table 5.3 The timings of the parallel bitonic sorting compute shader and comparable approaches.

Figure 5.19 shows that the timings for the compute-shader-based bitonic sorting nicely follows the estimated complexity of $\mathcal{O}(n \log^2 n)$. In addition to that, it has better timings than the C++ AMP implementations for the given problem sizes. It still does not reach the very efficient CUDA implementations.

Chapter 6

Conclusion & Future Work

6.1 Summary

To goal of this thesis was to identify a possible way of implementing an efficient point cloud renderer inside a state-of-the-art graphics engine on the example of the Unreal Engine that is capable of rendering huge static and dynamic point clouds in Real-Time. In doing so, an initial set of distinct aims were identified in the introduction (chapter 1) that should ensure that the point cloud renderer will be able to fulfil its purpose. After that, fundamentals on point clouds and a broad overview on the current research was given (chapter 2) to illustrate the different research areas and problems and their possible solutions. In the following, the most promising approaches in the context of this thesis were identified, which laid the foundation for the general conception and algorithm design of the renderer (chapter 3). These concepts were then implemented inside Unreal, which was presented in detail in the implementation section (chapter 4). During the implementation, two different approaches emerged, while the one approach was suited for static point clouds and the other one was well suited for static and dynamic and/or very large datasets. Now that the results were presented (chapter 5), it can be differentiated if and to what extent the initial goals for the renderer and the thesis were fulfilled. Furthermore, as a last step I will identify and summarise the limitations of the current approach and give ideas for possible future work while embedding the current approach in the research context at last.

6.2 Conclusion

The aims, as defined in section 1.2, were grouped into three basic principles: efficiency, quality and usability. Based on the presented results in the former chapter, I will now try to evaluate the fulfilment in this three dimensions.

To ensure the usability of the point cloud renderer, the plugin was tested with both static point cloud datasets and fully dynamic point cloud data coming from a Microsoft Kinect in a robotics-related use case. To ensure the efficiency of the

renderer, the used datasets were partly of industry-standard sizes of more than one million points and framerate measurings were taken under various conditions. To ensure the quality of the renderer, screenshots of the renderings of all datasets were taken, which were presented in chapter 5.

Generally, it can be concluded from the results that the point cloud renderer, as presented in this thesis, is capable of rendering both huge static and fully dynamic point clouds inside the Unreal Editor in Real-Time. The performance of the renderer is slightly worse than the comparable approach, however. Visually, the results are very close to the renderings given by the CloudCompare software. The visual quality can be further improved by employing a surface estimation, which compensates different point densities and allows for proper shading of the point clouds. However, the surface estimation, as implemented in the presented way, does not allow for surface reconstruction of fully dynamic point clouds as it is too computationally expensive to be performed on a per-frame basis. To further improve the visual quality, the dynamic renderer relied on splats with soft edges to yield better and softer renderings. For this, a compute-shader-based sorting was employed, which ensures a proper depth ordering and can be performed on a per-frame-basis, but has a distinct limit of approximately one million points per point cloud. In the best case, the here presented approach can produce high-quality renderings of at least medium-sized and dynamic point clouds with lighting and shadows in Real-Time.

Thus, it can be said that the here presented renderer fulfilled the former set aims very well as it is capable of rendering huge point clouds in Real-Time in various scenarios and use cases. It might very well be used for CVEs or on-the-fly environment scanning purposes, for instance in an robotics use case where a movable robot gains a representation of its surroundings. Merely the quality aspect is questionable, since a good visual result is ensured by soft-edged splats and a surface estimation procedure, but only works for comparably small and static point clouds.

Regarding the larger context question – if and how well a point-based renderer can be integrated into and implemented in a state-of-the-art graphics engine – it can be said that this indeed is possible. However, this means that it is likely

that one has to again rely on polygons (as I did in my implementation), but it still is possible to realise an efficient renderer, even though a tailored and custom rendering environment or pipeline can likely be pushed way further in terms of efficiency and quality.

Regarding the scientific relevance of this thesis it can be said that it contributes and points out several aspects in the context of point cloud rendering: On the one hand, a way to implement an efficient, GPU-based point cloud renderer in a polygon-based pipeline was presented. The ability to render large and dynamically changing point clouds is particularly remarkable since very few approaches focus on the rendering of dynamic point clouds. In this context, an alternative method for Order-Independent Transparency was proposed by the parallel bitonic sorting of the point cloud's point positions texture according to the camera position. Moreover, it was shown that the combination of a scanning device (in this case the Microsoft Kinect) and a positional tracking system (in this case the HTC Vive) allow for an easy and efficient environment scanning and can make computationally expensive matching or registration of the individual point cloud parts (for instance by the "iterative closest points" (ICP) algorithm) superfluous or at least improve it. This is particularly interesting for the mapping of smaller indoor environments, for instance in robotics, where until now computationally expensive matching/registration algorithms are dominant ([Iza+11], [Hen+12], [Kid+12], [Rus+09]).

6.3 Limitations & Future Work

As described on the former pages, the presented approach has various limitations and drawbacks. One major limitation of the dynamic point cloud renderer is the size of the point cloud, given by the maximum texture size in Unreal, which is 8192 by 8192, limiting the point clouds to a maximum of 67 million points. However, it is likely that the renderings suffer from poor precision of the point positions due to the poor precision of large float values when rendering very large point clouds (both in size and scaling). For sorted point positions textures, the maximum texture size

is even further limited (to 1024*1024 = approximately one million points on the testing machine) due to the shared memory/thread count limit. Furthermore and as stated above, the surface estimation is only feasible for static point clouds and cannot be performed in Real-Time on dynamic point clouds.

This limitations lead to several points which would be reasonable to be further investigated in future work. Firstly, this is improving the nearest neighbour search to leverage the surface estimation to be performed every frame. This could be realised by moving the surface estimation from object space to Image Space by implementing one of the presented approaches, e.g. [Pre+12]. Additionally or as an alternative to that, it could also be investigated if the search tree could be created and stored directly on the GPU. Also, updating the tree structure instead of rebuilding the whole tree every frame could be considerably beneficial.

In addition to that, one major restriction for rendering currently is that the point clouds have to fit into the system memory. As a remedy, appropriate hierarchical data structures or an Out-Of-Core system could be employed.

Also, several things could further improve the overall performance of the renderer. This could be for instance to use the CUDA-powered versions of the PCL libraries and to improve and refactor the overall implementation.

As seen is chapter 5, the parallel sorting of the point positions could also be further improved. It would be for instance reasonable to implement another parallel sorting algorithm, preferably similar to the CUDA Radix Sort implementation. Alternatively, the bitonic sorting algorithm could be further improved by several enhancements, like for example adaptive bitonic sorting [Gre+06] or a more sophisticated procedure that allow for more elements to be sorted. Furthermore, the sorting procedure has to be adapted to also sort the point colour data according to the point position data, which is not happening at the moment.

Moreover, since the bitonic sorting compute shader was used as a means of Order-Independent Transparency, it would be very interesting to further investigate the usage of parallel sorting algorithms as a means of OIT and to properly compare them to usual OIT approaches. While the parallel bitonic sorting as implemented in this thesis is likely too slow, other approaches might be considerable or even

faster than common approaches, like for example the high-performance CUDA radix sort. Depending on the outcome of this, other approaches for correct depth ordering might be preferable for my point cloud renderer.

To sum up, it would be interesting to see an implementation which utilises the GPU as much as possible, e.g. with an Image-Space surface estimation similar to [Pre+12] or a hierarchical tree structure directly on the GPU for an efficient nearest neighbour search and a Level-Of-Detail based rendering. In conjunction with that, a highly optimised massively parallel sorting algorithm could ensure the proper depth ordering of the individual points.

Appendix A

Appendix

A.1 List of Figures

A.2 List of Tables

A.3 Bibliography

[Ana17] Kostas Anagnostou. *How Unreal Renders a Frame*. 2017. URL: *https://interplayoflight.wordpress.com/2017/10/25/how-unreal-renders-a-frame/* (visited on 06/14/2018).

[Ark+17] Dmitri I. Arkhipov, Di Wu, Keqin Li, and Amelia C. Regan. "Sorting with GPUs: A Survey". In: (2017). arXiv: *1709.02520*. URL: *http://arxiv.org/abs/1709.02520*.

[Bas13] Debdatta Basu. *Parallel Radix Sort on the GPU using C++ AMP*. 2013. URL: *https://www.codeproject.com/articles/543451/parallel-radix-sort-on-the-gpu-using-cplusplus-amp* (visited on 08/16/2018).

[Bat68] K. E. Batcher. "Sorting networks and their applications". In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*. New York, New York, USA: ACM Press, 1968, p. 307. DOI: *10.1145/1468075.1468121*. URL: *http://portal.acm.org/citation.cfm?doid=1468075.1468121*.

[Ben+01] Steve Benford, Chris Greenhalgh, Tom Rodden, and James Pycock. "Collaborative virtual environments". In: *Communications of the ACM* 44.7 (2001), pp. 79–85. ISSN: 00010782. DOI: *10.1145/379300.379322*. URL: *http://portal.acm.org/citation.cfm?doid=379300.379322*.

[Ben75] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517. ISSN: 00010782. DOI: *10.1145/361002.361007*. URL: *http://portal.acm.org/citation.cfm?doid=361002.361007*.

[Ber+14] Matthew Berger, Pierre Alliez, Andrea Tagliasacchi, Lee M Seversky, Claudio T. Silva, Joshua a. Levine, and Andrei Sharf. "State of the Art in Surface Reconstruction from Point Clouds". In: *Proceedings of the Eurographics 2014, Eurographics STARs* (2014), pp. 161–185. ISSN: 1017-4656. DOI: *10.2312/egst.20141040*. URL: *http://lgg.epfl.ch/reconstar*.

[Bha11] M. Bharath. *Bitonic Sort Sample using C++ AMP*. 2011. URL: *https://blogs.msdn.microsoft.com/nativeconcurrency/2011/11/11/bitonic-sort-sample-using-c-amp/* (visited on 08/16/2018).

[Bot+05] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. "High-quality surface splatting on today's GPUs". In: *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.* (2005), pp. 17–141. ISSN: 1511-7813. DOI: *10.1109/PBG.2005.194059*. URL: *http://ieeexplore.ieee.org/document/1500313/*.

[Bru+14] Gerd Bruder, Frank Steinicke, and Andreas Nüchter. "Poster: Immersive point cloud virtual environments". In: *2014 IEEE Symposium on 3D User Interfaces (3DUI)*. IEEE, 2014, pp. 161–162. ISBN: VO -. DOI: *10.1109/3DUI.2014.6798870*. URL: *http://ieeexplore.ieee.org/document/6798870/*.

[Cap+12] Gabriele Capannini, Fabrizio Silvestri, and Ranieri Baraglia. "Sorting on GPUs for large scale datasets: A thorough comparison". In: *Information Processing and Management* 48.5 (2012), pp. 903–917. ISSN: 03064573. DOI: *10.1016/j.ipm.2010.11.010*. URL: *http://dx.doi.org/10.1016/j.ipm.2010.11.010*.

[Dob+10] Petar Dobrev, P Rosenthal, and L Linsen. "An image-space approach to interactive point cloud rendering including shadows and transparency". In: *Computer Graphics …* (2010), pp. 1–29. URL: *http://cgg-journal.com/2010-3/02/*.

[Ede+94] Herbert Edelsbrunner and Ernst P. Mücke. "Three-dimensional alpha shapes". In: *ACM Transactions on Graphics* 13.1 (1994), pp. 43–72. ISSN: 07300301. DOI: *10.1145/174462.156635*. URL: *http://portal.acm.org/citation.cfm?doid=174462.156635*.

[EDF18] EDF. *CloudCompare*. 2018. URL: *https://github.com/cloudcompare/cloudcompare*.

[Fra17] Simon Maximilian Fraiss. "Rendering Large Point Clouds in Unity". PhD thesis. Technische Universität Wien, 2017, p. 38.

[Fu+16] Cong Fu and Deng Cai. "EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph". In: (2016), pp. 1–20. arXiv: *1609.07228*. URL: *http://arxiv.org/abs/1609.07228*.

[Fur+10] Yasutaka Furukawa and Jean Ponce. "Accurate, dense, and robust multiview stereopsis". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.8 (2010), pp. 1362–1376. ISSN: 01628828. DOI: *10.1109/TPAMI.2009.161*.

[Fur+15] Yasutaka Furukawa and Carlos Hernández. "Multi-View Stereo: A Tutorial". In: *Foundations and Trends® in Computer Graphics and Vision* 9.1-2 (2015), pp. 1–148. ISSN: 1572-2740. DOI: *10.1561/0600000052*. arXiv: *0703101v1* [cs]. URL: *http://www.nowpublishers.com/article/Details/CGV-052*.

[Gei+13] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. "Vision meets robotics: The KITTI dataset". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1231–1237. ISSN: 0278-3649. DOI: *10.1177/0278364913491297*. arXiv: *1102.0183*. URL: *http://journals.sagepub.com/doi/10.1177/0278364913491297*.

[Gre+06] Alexander Greß and Gabriel Zachmann. "GPU-ABiSort: Optimal parallel sorting on stream architectures". In: *20th International Parallel and Distributed Processing Symposium, IPDPS 2006* 2006 (2006). ISSN: 1530-2075. DOI: *10.1109/IPDPS.2006.1639284*.

[Gün+13] Christian Günther, Thomas Kanzok, Lars Linsen, and Paul Rosenthal. "A GPGPU-based pipeline for accelerated rendering of point clouds". In: *Journal of WSCG* 21.2 (2013), pp. 153–161. ISSN: 12136972.

[Hai+18] Andrei Haidu and Michael Beetz. *RobCog: Robot Commonsense Games*. 2018. URL: *http://robcog.org* (visited on 04/10/2018).

[Hen+12] Peter Henry, Michael Krainin, Evan Herbst, Xiaofeng Ren, and Dieter Fox. "RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments". In: *International Journal of Robotics Research* 31.5 (2012), pp. 647–663. ISSN: 02783649. DOI: *10.1177/0278364911434148*.

[Hof17] Matt Hoffman. *Unreal Engine 4 Rendering*. 2017. URL: *https://medium.com/lordned/unreal-engine-4-rendering-overview-part-1-c47f2da65346* (visited on 06/13/2018).

[Iza+11] S Izadi, Kim, O Hilliges, D Molyneaux, R Newcombe, P Kohli, J Shotton, S Hodges, D Freeman, A Davison, and A Fitzgibbon. "Ki-

nectFusion: real-time 3D reconstruction and interaction using a moving depth camera". In: *Proceedings of the 24th annual ACM User Interface Software and Technology Symposium - UIST '11* (2011), pp. 559–568. ISSN: 9781450307161. DOI: *10.1145/2047196.2047270*. URL: *http://dl.acm.org/citation.cfm?id=2047270%5Cnpapers://c80d98e4-9a96-4487-8d06-8e1acc780d86/Paper/p5008*.

[Kid+12] Ross Kidson, Dejan Pangercic, Darko Stanimirovic, and Michael Beetz. "Elaborative Evaluation of RGB-D based Point Cloud Registration for Personal Robots". In: *ICRA 2012 Workshop on Semantic Perception and Mapping for Knowledge-enabled Service Robotics*. St. Paul, MN, USA, 2012.

[Kle+04a] Jan Klein and Gabriel Zachmann. "Nice and fast implicit surfaces over noisy point clouds". In: *ACM SIGGRAPH 2004 Sketches on - SIGGRAPH '04* 80 (2004), p. 85. DOI: *10.1145/1186223.1186329*. URL: *http://portal.acm.org/citation.cfm?doid=1186223.1186329*.

[Kle+04b] Jan Klein and Gabriel Zachmann. "Proximity graphs for defining surfaces over point clouds". In: *Journal of the ACM* (2004), pp. 131–138. URL: *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.9.7048&rep=rep1&type=pdf*.

[Kob+04] Leif Kobbelt and Mario Botsch. "A survey of point-based techniques in computer graphics". In: *Computers and Graphics (Pergamon)* 28.6 (2004), pp. 801–814. ISSN: 00978493. DOI: *10.1016/j.cag.2004.08.009*. URL: *http://linkinghub.elsevier.com/retrieve/pii/S0097849304001487*.

[Kop+11] Hema Swetha Koppula, Abhishek Anand, Thorsten Joachims, and Ashutosh Saxena. "Semantic Labeling of 3D Point Clouds for Indoor Scenes". In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor Weinberger, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Vol. 24. Curran Associates, Inc., 2011, pp. 244–252. ISBN: 9781618395993. DOI: *10.1109/CVPR.2012.6247992*. URL: *http://pr.cs.cornell.edu/sceneunderstanding/nips_2011.pdf*.

[Law72] Charles L. Lawson. "Transforming triangulations". In: *Discrete Mathematics* 3.4 (1972), pp. 365–372. ISSN: 0012365X. DOI: *10.1016/0012-365X(72)90093-3*.

[Ler+16] Adam Lerer, Sam Gross, and Rob Fergus. "Learning Physical Intuition of Block Towers by Example". In: (2016). arXiv: *1603.01312*. URL: *http://arxiv.org/abs/1603.01312*.

[Lev+85] Marc Levoy and Turner Whitted. *The Use of Points as a Display Primitive*. Tech. rep. Chapel Hill, Noth Carolina, USA: University of North Carolina at Chapel Hill, 1985.

[Lev05] Marc Levoy. *The Stanford 3D Scanning Repository*. 2005. URL: *https://graphics.stanford.edu/data/3Dscanrep/* (visited on 08/09/2018).

[Lev09] Marc Levoy. *The Digital Michelangelo Project*. 2009. URL: *http://graphics.stanford.edu/data/mich/* (visited on 08/07/2018).

[Lev98] David Levin. "The approximation power of moving least-squares". In: *Mathematics of Computation* 67.224 (1998), pp. 1517–1532. ISSN: 0025-5718. DOI: *10.1090/S0025-5718-98-00974-0*. URL: *http://www.ams.org/journal-getitem?pii=S0025-5718-98-00974-0*.

[Liu+12] Ming Liu, Francois Pomerleau, Francis Colas, and Roland Siegwart. "Normal estimation for pointcloud using GPU based sparse tensor voting". In: *2012 IEEE International Conference on Robotics and Biomimetics, ROBIO 2012 - Conference Digest* May (2012), pp. 91–96. DOI: *10.1109/ROBIO.2012.6490949*.

[Liu13] Fang Liu. "Efficient rendering of order independent transparency on the GPUs". In: *Lecture Notes in Earth System Sciences*. 9783642164040. Springer Berlin Heidelberg, 2013, pp. 437–455. ISBN: 978-3-642-16404-0 978-3-642-16405-7. DOI: *10.1007/978-3-642-16405-7_28*. URL: *http://link.springer.com/10.1007/978-3-642-16405-7_28*.

[Lor+87] William E. Lorensen and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *ACM SIGGRAPH Computer Graphics* 21.4 (1987), pp. 163–169. ISSN: 00978930. DOI: *10.1145/37402.37422*. URL: *http://portal.acm.org/citation.cfm?doid=37402.37422*.

[Mau+11] Marilena Maule, João L.D. Comba, Rafael P. Torchelsen, and Rui Bastos. "A survey of raster-based transparency techniques". In: *Computers and Graphics (Pergamon)* 35.6 (2011), pp. 1023–1034. ISSN: 00978493. DOI: *10.1016/j.cag.2011.07.006*. URL: *http://linkinghub.elsevier.com/retrieve/pii/S009784931100135X*.

[Mcg+13] Morgan Mcguire and Louis Bavoil. "Weighted Blended Order-Independent Transparency". In: *Journal of Computer Graphics Techniques* 2.2 (2013), pp. 122–141. ISSN: 2331-7418.

[Mea82] Donald Meagher. "Geometric modeling using octree encoding". In: *Computer Graphics and Image Processing* 19.2 (1982), pp. 129–147. ISSN: 0146664X. DOI: *10.1016/0146-664X(82)90104-6*. URL: *http://linkinghub.elsevier.com/retrieve/pii/0146664X82901046*.

[Men+97] Robert Mencl and Heinrich Müller. "Interpolation and Approximation of Surfaces from Three-Dimensional Scattered Data Points". In: *Proceedings of the Conference on Scientific Visualization*. DAGSTUHL '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 223–232. ISBN: 0-7695-0503-1. URL: *http://dl.acm.org/citation.cfm?id=789080.789103*.

[Mit+03] Niloy J. Mitra and An Nguyen. "Estimating surface normals in noisy point cloud data". In: *Proceedings of the nineteenth conference on Computational geometry - SCG '03* 14.04n05 (2003), p. 322. ISSN: 0218-1959. DOI: *10.1145/777792.777840*. URL: *http://portal.acm.org/citation.cfm?doid=777792.777840*.

[Muj+14] Marius Muja and David G. Lowe. "Scalable Nearest Neighbour Algorithms for High Dimensional Data". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11 (2014), pp. 2227–2240. ISSN: 0162-8828. DOI: *10.1109/TPAMI.2014.2321376*. URL: *http://elk.library.ubc.ca/handle/2429/44402%5Cnhttp://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6809191*.

[Nüc+16] Andreas Nüchter and Kai Lingemann. *Robotic 3D Scan Repository*. 2016. URL: *http://kos.informatik.uni-osnabrueck.de/3Dscans* (visited on 08/07/2018).

[Paj+05] Renato Pajarola, Miguel Sainz, and Roberto Lario. "XSplat: External memory multiresolution point visualization". In: *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing (VIIP)* (2005), pp. 628–633. URL: *http://www.actapress.com/PaperInfo.aspx?paperId=21744*.

[Pet+10] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. "Fast in-place sorting with CUDA based on bitonic sort bitonic sort". In: *Parallel Processing and Applied Mathematics*. 2010, pp. 403–410. ISBN: 978-3-642-14390-8. DOI: *10.1007/978-3-642-14390-8_42*. URL: *http://link.springer.com/10.1007/978-3-642-14390-8_42*.

[Pfi+00] H. Pfister, M. Zwicker, J. Van Baar, and M. Gross. "Surfels: Surface Elements as Rendering Primitives". In: *Proceedings of the 27th* pp (2000), pp. 335–342. ISSN: 0278-3649. DOI: *10.1145/344779.344936*.

[Pin+11] Ruggero Pintus, Enrico Gobbetti, and Marco Agus. "Real-time rendering of massive unstructured raw point clouds using screen-space operators". In: *Proceedings of the 12th International conference on Virtual Reality, Archaeology and Cultural Heritage* (2011), pp. 105–112. DOI: *10.2312/vast/vast11/105-112*. URL: *http://www.crs4.it/vic/data/papers/vast2011-pbr.pdf*.

[Pre+12] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. "Auto Splats: Dynamic Point Cloud Visualization on the GPU". In: *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2012), pp. 139–148. DOI: *10.2312/EGPGV/EGPGV12/139-148*. URL: *http://diglib.eg.org/EG/DL/WS/EGPGV/EGPGV12/139-148.pdf.abstract.pdf;internal&action=action.digitallibrary.ShowPaperAbstract*.

[Qiu+16] Weichao Qiu and Alan Yuille. "UnrealCV: Connecting Computer Vision to Unreal Engine". In: *Computer Vision – ECCV 2016 Workshops*. Lecture Notes in Computer Science 9915 (2016). Ed. by Gang Hua and Hervé Jégou. ISSN: 0302-9743. DOI: *10.1007/978-3-319-49409-8*. arXiv: *1603.06937*. URL: *http://link.springer.com/10.1007/978-3-319-49409-8*.

[Ree83] William T. Reeves. "Particle systems - a technique for modeling a class of fuzzy objects". In: *ACM SIGGRAPH Computer Graphics* 17.3 (1983), pp. 359–375. ISSN: 00978930. DOI: *10.1145/964967.801167*. URL: *http://portal.acm.org/citation.cfm?doid=964967.801167*.

[Rem04] Fabio Remondino. "From point cloud to surface: the modeling and visualization problem". In: *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 34 (2004). DOI: *10.3929/ethz-a-004655782*. URL: *http://pdf.aminer.org/000/291/298/surface_reconstruction_from_large_point_clouds_using_virtual_shared_memory.pdf*.

[Rus+00] Szymon Rusinkiewicz and Marc Levoy. "QSplat". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00* (2000), pp. 343–352. DOI: *10.1145/344779.344940*. URL: *http://portal.acm.org/citation.cfm?doid=344779.344940*.

[Rus+09] Radu Bogdan Rusu, Zoltan Csaba Marton, Nico Blodow, Andreas Holzbach, and Michael Beetz. "Model-based and learned semantic object labeling in 3D point cloud maps of kitchen environments". In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009*. IEEE, 2009, pp. 3601–3608. ISBN: 9781424438044. DOI: *10.1109/IROS.2009.5354759*. URL: *http://ieeexplore.ieee.org/document/5354759/*.

[Rus+11] Radu Bogdan Rusu and S. Cousins. "3D is here: point cloud library". In: *IEEE International Conference on Robotics and Automation* (2011), pp. 1 –4. ISSN: 1050-4729. DOI: *10.1109/ICRA.2011.5980567*. URL: *http://pointclouds.org/*.

[Sch+15a] Andre Schollmeyer, Andrey Babanin, and Bernd Froehlich. "Order-Independent Transparency for Programmable Deferred Shading Pipelines". In: *Computer Graphics Forum* 34.7 (2015), pp. 67–76. ISSN: 14678659. DOI: *10.1111/cgf.12746*. URL: *http://doi.wiley.com/10.1111/cgf.12746*.

[Sch+15b] Markus Schütz and Michael Wimmer. "Rendering Large Point Clouds in Web Browsers". In: *Proceedings of CESCG 2015: The 19th Central*

European Seminar on Computer Graphics (non-peer-reviewed) (2015). URL: *http://old.cescg.org/CESCG-2015/papers/Schutz-Rendering_Large_Point_Clouds_in_Web_Browsers.pdf*.

[Sha+17] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: (2017). ISSN: 1938-7228. DOI: *10.1007/978-3-319-67361-5_40*. arXiv: *1705.05065*. URL: *http://arxiv.org/abs/1705.05065*.

[Smi+11] Jan Smisek, Michal Jancosek, and Tomas Pajdla. "3D with Kinect". In: *Proceedings of the IEEE International Conference on Computer Vision*. IEEE, 2011, pp. 1154–1160. ISBN: 9781467300629. DOI: *10.1109/ICCVW.2011.6130380*. arXiv: *arXiv:1011.1669v3*. URL: *http://ieeexplore.ieee.org/document/6130380/*.

[Sug14] Tsukasa Sugiura. *Kinect2Grabber Github repository*. 2014. URL: *https://github.com/UnaNancyOwen/KinectGrabber* (visited on 08/03/2018).

[Wal15] Chuck Walbourn. *DirectCompute Basic Win32 Samples*. 2015. URL: *https://code.msdn.microsoft.com/windowsdesktop/DirectCompute-Basic-Win32-7d5a7408* (visited on 06/20/2018).

[Wan+07] Michael Wand, Alexander Berner, Martin Bokeloh, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier, Dirk Staneker, and Andreas Schilling. "Interactive Editing of Large Point Clouds". In: *Eurographics Symposium on Point-Based Graphics* (2007), pp. 37–45. DOI: *10.2312/SPBG/SPBG07/037-045*.

[Wes89] Lee Westover. "Interactive volume rendering". In: *Proceedings of the 1989 Chapel Hill workshop on Volume visualization - VVS '89*. Vol. 41. 7. Chapel Hill, Noth Carolina, USA: ACM Press, 1989, pp. 9–16. DOI: *10.1145/329129.329138*. URL: *http://portal.acm.org/citation.cfm?id=329138http://portal.acm.org/citation.cfm?doid=329129.329138*.

[Whi+10] Mark Whitty, Stephen Cossell, and KS Dang. "Autonomous navigation using a real-time 3D point cloud". In: *Australasian Conference on Robotics and Automation* (2010), pp. 1–10. URL: *http://www.araa.asn.au/acra/acra2010/papers/pap151s1-file1.pdf*.

[Wim+06] Michael Wimmer and Claus Scheiblauer. "Instant Points : Fast Rendering of Unprocessed Point Clouds". In: *Eurographics / IEEE VGTC*

Conference on Point-Based Graphics (2006), pp. 129–137. DOI: *10.2312/SPBG/SPBG06/129-136*. URL: *http://dx.doi.org/10.2312/SPBG/SPBG06/129-136*.

[Zha+16] Yi Zhang, Weichao Qiu, Qi Chen, Xiaolin Hu, and Alan Yuille. "UnrealStereo: A Synthetic Dataset for Analyzing Stereo Vision". In: (2016). arXiv: *1612.04647*. URL: *http://arxiv.org/abs/1612.04647*.

[Zho+08a] Kun Zhou, M Gong, X Huang, and B Guo. "Highly parallel surface reconstruction". In: *Microsoft Research Asia* (2008). DOI: *10.1.1.142.5807*. URL: *http://kunzhou.net/2008/MSR-TR-2008-53.pdf*.

[Zho+08b] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. "Real-time KD-tree construction on graphics hardware". In: *ACM Transactions on Graphics* 27.5 (2008), p. 1. ISSN: 07300301. DOI: *10.1145/1409060.1409079*. URL: *http://portal.acm.org/citation.cfm?doid=1409060.1409079*.

[Zwi+01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. "Surface splatting". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01* (2001), pp. 371–378. DOI: *10.1145/383259.383300*. URL: *http://portal.acm.org/citation.cfm?doid=383259.383300*.