FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Real-Time Visualization Pipeline for Dynamic Point Cloud Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

BSc Hansjörg Hofer
Matrikelnummer 01026632

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Mag. Dr. Margrit Gelautz
Mitwirkung: Dr. Florian Seitner

Wien, 26. August 2018

Hansjörg Hofer	Margrit Gelautz

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Real-Time Visualization Pipeline for Dynamic Point Clouds

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

BSc Hansjörg Hofer
Registration Number 01026632

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Mag. Dr. Margrit Gelautz
Assistance: Dr. Florian Seitner

Vienna, 26th August, 2018

Hansjörg Hofer | Margrit Gelautz

Erklärung zur Verfassung der Arbeit

BSc Hansjörg Hofer
Sechsschimmelgasse 16/13

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. August 2018

Hansjörg Hofer

Danksagung

Ich möchte mich bei allen bedanken, die einen Beitrag zu dieser Abschlussarbeit und dem resultierenden Projekt geleistet haben. Bei Florian Seitner, für seine Leitung und die Hilfe den Fokus der Schlüsselaspekte dieser Arbeit nicht aus den Augen zu verlieren und dafür, dass er mir die Zeit und den Raum zur Verfügung gestellt hat, dieses Projekt fertigzustellen. Bei Michael Hödlmoser für seine Unterstützung und Hilfe. Bei meiner Betreuerin, Margrit Gelautz, für die reibungslose Zusammenarbeit mit emotion3D.

Vielen Dank an Reinhold Preiner, für seine vorhergehende Arbeit an AutoSplats und die Bereitstellung des Shader-Quellcodes für die Singulärwertzerlegung.

Vielen Dank an Christian Schönauer, für das Bereitstellen des mobilen Testgeräts.

Vielen Dank an alle Mitarbeiter bei emotion3D, für die Unterstützung und das geteilte Interesse am Projekt.

Vielen Dank an meine Freundin, für die Hilfe und Motivation, besonders in den langen und oft späten Stunden in der Bibliothek.

Besonderen Dank an meine Familie, für die Unterstützung und dafür, dass sie mir ermöglichen meiner Leidenschaft nachzugehen.

Diese Masterarbeit wurde von dem Projekt Precise3D (num. 855442) unterstützt, welches vom Bundesministerium für Verkehr, Innovation und Technologie (BMVIT) zusammen mit der Forschungsförderungsgesellschaft (FFG) finanziert und gefördert wird. Precise3D ist Teil des „ICT of the Future“ Programms.

Acknowledgements

I would like to thank everyone who contributed to this thesis and the resulting project. To Florian Seitner, for his guidance and help focusing on the key aspects of this thesis and for giving me the time and space to accomplish this project. To Michael Hödlmoser for the support and help. To my advisor, Margrit Gelautz, for the seamless cooperation with emotion3D.

Thanks to Reinhold Preiner, for his previous work on AutoSplats and for providing the shader source code for the singular value decomposition.

Thanks to Christian Schönauer, for providing the mobile testing device.

Thanks to all my coworkers at emotion3D and all their support and shared interest for this project.

Thanks to my girlfriend, for her help and the motivation, especially in the long and often late hours in the library.

Special thanks to my family, for supporting me and enabling me to pursue my passion.

This work has been supported by the project Precise3D (no. 855442), which is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) in conjunction with the Austrian Research Promotion Agency (FFG) under the program „ICT of the Future“.

Kurzfassung

Aktuelle Entwicklungen in Sensor und Computer Vision Technologien erweitern die Grenzen des Machbaren im Bereich von Extended Reality (XR) Anwendungen. Individualisierbare digitale Avatare, die menschliche Emotionen und Gesten nachahmen, werden bald durch echte 3D Aufnahmen unserer selbst in der virtuellen Welt ersetzt. Digitale 3D Rekonstruktionen von echten Umgebungen und lebendigen Objekten erlauben realitätsnahe und immersive Erlebnisse. Solche Aufnahmen können die virtuelle Welt um lebensechte Inhalte erweitern und so die Möglichkeiten für Virtual Reality (VR), Augmented Reality (AR) und Mixed Reality (MR) Anwendungen vergrößern. Photogrammetrie und Tiefensensoren sind bereits in der Lage starre Körper detailgetreu, mit erstaunlich lebensechten Resultaten, zu digitalisieren. Moderne Tiefensensoren haben bereits Einzug in den Smartphone-Markt erhalten und legen somit den ersten Stein für mobile Applikationen mit Echtzeit-3D-Rekonstruktions-Funktionalität. Allerdings ist das Rekonstruieren von sich schnell ändernden Szenen mit flexibler Topologie immer noch eine große Herausforderung, besonders in Echtzeitsystemen. Diese Diplomarbeit präsentiert eine neue Visualisierungs-Pipeline zur Oberflächenrekonstruktion von dynamischen Punktwolken, integriert in ein weitverbreitetes und flexibles Framework. Das modulare System erlaubt das flexible Verarbeiten und Darstellen von Punktwolken in der *Unity3D* Spiel-Engine, eine gängige plattformunabhängige Engine für XR Applikationen und mobile Spiele. Die Implementierung ermöglicht das Darstellen von fotorealistischen dynamischen Objekten direkt aus den Live-Daten von Tiefenkameras. Zusätzlich fördert die modulare Architektur die Skalier- und Erweiterbarkeit für verschiedenste Anwendungsfälle mit Punktwolkendaten. Die Qualität dieser neuen Visualisierungs-Pipeline wird durch den optischen Vergleich mit klassischen Visualisierungstechniken bestimmt. Dadurch kann gezeigt werden, dass sich diese kaum von den Ergebnissen der vorliegenden Arbeit unterscheiden. Zugleich wird die exakte Durchlaufzeit der verwendeten Algorithmen für verschiedene Punktwolken gemessen, um die Echtzeit-Lauffähigkeit zu belegen.

Abstract

Current developments in sensor and computer vision technologies are pushing the boundaries of extended reality (XR) applications. Digital customizable avatars, mimicking human emotions and gestures, are soon to be replaced by true 3D capturings of humans in mixed reality environments. Recording action filled scenes to recreate and place them in virtual or remote environments promises more lifelike and immerse experiences, extending the possibilities of virtual reality (VR), augmented reality (AR) and mixed reality (MR) applications. Photogrammetry and depth ranging sensors are already capable of bringing rigid real-world objects to the virtual world, with astoundingly realistic results. In addition, with modern depth sensors entering the smartphone industry, the next steps for mobile 3D capturing and online reconstruction have already been taken. However, dynamic content with rapid changes in structure and topology is challenging to reconstruct, particularly for usage in real-time. This thesis presents a novel visualization pipeline, which facilitates the real-time reconstruction of dynamic point clouds in a widely used, flexible and powerful framework. We contribute a novel modular processing and rendering pipeline for the *Unity3D* game engine, a popular multi-platform engine for XR applications and mobile games. The implementation is capable of reconstructing photo-realistic 3D objects from dynamic depth sensor streams. Furthermore, the modular architecture allows for scalability and easy expandability. The quality of this novel visualization pipeline is determined through optical comparisons with classical visualization techniques, which show that those techniques are not easily distinguishable from the results of our work. Moreover, the exact execution time of the algorithms is measured for different point clouds to proof their real-time ability.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Goal . . . 2
1.2 Thesis Structure . . . 3

2 Related Work **5**
2.1 End-to-End Systems . . . 5
2.2 Point Cloud Processing . . . 7

3 Engine Framework and Libraries **11**
3.1 Unity3D Game Engine . . . 11
3.2 LibRealSense . . . 15
3.3 OpenCV . . . 16

4 System Concept **17**
4.1 Input Handling . . . 18
4.2 Data Culling . . . 20
4.3 Surface Reconstruction . . . 22
4.4 Rendering . . . 23

5 Implementation **25**
5.1 Point Cloud . . . 25
5.2 Base Pipeline Component . . . 26
5.3 Data Handler Components . . . 27
5.4 Culling Helper Components . . . 30
5.5 Surface Processor Components . . . 33
5.6 Renderer Components . . . 36
5.7 Data Provider Interface . . . 39
5.8 Restrictions . . . 41

6 Evaluation and Results **45**
6.1 Data Generation 45
6.2 Visual Quality Comparison 46
6.3 Performance Comparison 53
6.4 Applications 56

7 Conclusion and Future Work **63**

List of Figures **65**

List of Tables **69**

Acronyms **71**

Bibliography **73**

CHAPTER 1

Introduction

The increasing popularity of virtual reality (VR), augmented reality (AR) and mixed reality (MR) applications in commercial and industrial environments, due to the emergence of powerful mobile devices and novel hardware, encourages further advancements in computer vision and computer graphic technologies. VR systems recreate our world in a virtual environment and allow us to fully immerse in artificial microcosms. AR and MR applications aim to enhance our everyday experiences in the real world, through digital augmentations [CBHH17]. All this is achieved through state-of-the-art technologies, allowing to overcome the boundaries between reality and virtuality. Equally important is the use of authentic digital content in order to create seamless, immersive experiences. Most applications rely heavily on manually fabricated content. Such 3D models, crafted by artists and designers, can be expensive, as lifelike high fidelity models require a lot of time, even for skilled professionals. Modern approaches try to minimize the manual labor, while increasing the visual quality of the 3D objects. This can be accomplished with the aid of 3D reconstruction technologies. A common procedure is photogrammetry, where a static object is digitally reconstructed from multiple sources [REH06]. Feature detection techniques allow to identify points across different views and calculate 3D point clouds by triangulating the corresponding features. The result is a colored point cloud which is then refined and converted to a polygonal mesh. This process can be further improved with additional depth sensors, simplifying the point cloud extraction. Content created with these technologies is often more authentic than hand-made replications and allows creating photo-realistic environments rendered in real-time [Bou18].

Reconstructing static objects or objects with defined movement constraints, from point cloud data, is an ongoing field of research. Sophisticated algorithms are constantly enhanced to efficiently and accurately reconstruct the original shape and appearance of a variety of captured environments. However, dynamic scenes with no movement constraints, nor previous knowledge of the change in topology, are troublesome to reconstruct in an efficient and automated manner. Virtualizing scenes with moving people, animals

or other non-rigid bodies is still an open challenge. Nevertheless, such scenes are quite desirable in many applications. The real-time reconstruction of dynamic data allows using sensor live streams to scan and record arbitrary scenes, with no prior restrictions or offline preprocessing steps. Especially extended reality (XR) applications can benefit from the seamless integration of concurrent reconstruction and visualization of live captured 3D data. This becomes even more relevant with the recent advent of depth sensors for mobile devices [iPh, Str], which open a completely new area of possibilities. For example, mobile telepresence systems with a live 3D projection of the dialogue partner, making it possible to talk to a person as if he or she was physically present. By extending this idea, one could enjoy recordings of a dance performance captured in 3D, or experience a live concert in the own living room.

While existing approaches focus on the rigorous real-time reconstruction of manifold 3D meshes, trading performance and robustness for accuracy, the need for a flexible but still visually appealing visualization is overlooked. Many real-world applications have no need for precisely reconstructed surface meshes, instead perceivably satisfying representations of the captured scene with interactive frame-rates are more desired. Commercial systems have an increasing interest in virtual representations of people, with interactive visual feedback. In contrast to the true reconstruction of people in 3D, some commercial applications focus on the usage of customizable avatars as a digital representative of humans. Sensors and other input devices capture emotions, poses and gestures, which are then imitated by the virtual personification. With the possibility of truly recreating dynamic scenes and movements in 3D, it also becomes viable to have realistic virtual representations of people without the need for sophisticated pre-trained models. This could make cartoonish avatars become obsolete, as they may be inappropriate for many serious applications.

This thesis tries to overcome some of these issues by presenting a scalable real-time visualization pipeline for dynamic point cloud data that runs on current consumer hardware.

1.1 Goal

The aim of this work is to provide a real-time visualization pipeline for dynamic point clouds. The reconstruction and visualization algorithms are chosen carefully, as they need to function without the need for pre-trained models or any other previously defined constraints of the shape, topology or behavior of the incoming data. The proposed pipeline is completely unaware of the actual processed scene and its variation over time. The dynamic data is represented by recordings of human faces and upper bodies, to show varying facial expressions and movements. Additionally to the dynamic data, also static point clouds need to be processed with the proposed pipeline, to show its scalability and flexibility regarding the input data.

The implementation is realized as a set of components and plugins for a common game engine. The engine simplifies the utilization of the point cloud visualization and enables

its usage in a variety of applications and platforms, such as different desktop and mobile operating systems. The pipeline consists of four different stages, which will be discussed separately. Each stage is implemented with modularity in mind and offers scalability in quality and performance. The focus lies on the robustness, easy expandability and adequate hardware requirement of the plugin. This work will not only be a proof of concept with theoretically unlimited resources but rather a fully functional point cloud extension to a widespread multi-platform game engine.

1.2 Thesis Structure

This thesis is structured in seven chapters, which are shortly summarized in this section. First, we provide an outline of related state-of-the-art techniques, covering similar visualization workflows and publications related to the different point processing stages of our proposed pipeline. In Chapter 3 a closer look at the adopted engine, the programming framework and relevant concepts is taken and additional libraries are introduced. Chapter 4 covers the conceptual outline of the proposed visualization pipeline. This guides the reader through the subsequent Chapter 5, where implementation details and design specific decisions are explained in depth. After the concept and implementation details of this work are clarified, the results and evaluations are shown and discussed in Chapter 6. Lastly, we summarize our outcome and give an outlook on future advancements and possible further developments.

CHAPTER 2

Related Work

Real-time 3D reconstruction is a steadily growing field of research. Faster and cheaper hardware allows not only to capture 3D data in real-time, but also to reconstruct and visualize it with interactive frame rates. Early approaches used, for example, multi-view 2D RGB images to extract 3D information of dynamic objects [LZZ13]. Modern systems are capable of reconstructing triangle meshes of dynamic point clouds from depth sensors in real-time [OERF+16].

2.1 End-to-End Systems

In this section, real-time reconstruction and visualization pipelines for objects and scenes are presented. We focus on publications that describe end-to-end systems from the point cloud generation, data preprocessing and surface estimation to finally rendering of the reconstructed object. Many of these systems deal with similar problems, limitations and restrictions, trading visual quality for a higher performance.

An approach using only multi-view 2D images of the object was proposed in [LZZ13]. The work focuses on the real-time digitizing of objects based on the visual hull computed from segmented images. The proposed algorithm constructs a volumetric grid and culls voxels based on the object's silhouette from the different captured views. The object is carved out of the raw voxel block in a highly parallel manner. Finally, each surface voxel is rendered using a point rendering technique called splatting [ZPVBG01]. A splat is an oriented ellipse rendered in place of each surface point. By choosing an appropriate radius, the splats overlap and can be blended to create a smooth surface [BSK04]. Each splat is colored from the source images weighted by their contribution in the newly synthesized view. The size of the splats, and thus the visual quality, depends on the resolution of the volume.

Current graphics hardware is not optimized for handling massive amounts of pixel-sized splats. The article [Ric] gives a rough idea of the possible impact of small geometry, by using very simple benchmarks on a limited set of GPUs. Therefore, an often-favorable alternative to point based rendering is to generate a simplified triangle mesh from the captured dense point cloud data. However, point clouds have no connectivity or neighborhood information stored, which makes meshing algorithms very expensive. Reconstructing a triangle mesh from 3D points is a challenging field of research. The popular Point Cloud Library [RC11] provides offline algorithms to estimate surface normals and compute triangle meshes. Building such meshes in real-time is still an ongoing challenge. Furthermore, when dealing with dynamic point clouds, which may change rapidly from frame to frame, meshing algorithms need to keep track of the changes and have to adapt the reconstructed mesh accordingly [NFS15, DKD+16].

The more recent publication [OERF+16] describes the usage of a high-resolution multi-stereo-camera setup. The authors present a 360° immersive 3D telepresence system. A scene with static furniture, moving persons and even animals can be captured by multiple stereo and RGB cameras. Each sensor setup is preprocessed by a multi-GPU workstation, where the depth is computed and the foreground segmented. An additional workstation is then collecting all views to combine them to compute the final textured triangle mesh. The used reconstruction algorithm proposed previously by [DKD+16] is the computationally most expensive part in the pipeline. The algorithm was implemented on a dual-GPU scheme to allow room scaled reconstruction while still having real-time performance. The big advantage of computing triangle meshes, as opposed to point based visualization techniques, is the straightforward high-performance rendering. This allows the remote rendering of the reconstructed mesh on a multitude of devices, including Microsoft's HoloLens [Hol], only limited by the communication bandwidth. The focus of [OERF+16] lies in the high-quality reconstruction while still having real-time performance. An immense amount of processing power was needed to achieve this goal.

In contrast to the extensive use of high-end hardware needed for the Holoportation system [OERF+16], Tytgat et al. [TADB+16] present a low-cost end-to-end application running on consumer hardware. They use similar reconstruction techniques but with lower resolution and on a smaller scale. The proposed system is capable of capturing and reconstructing a single person's upper body and subsequently rendering it remotely in real-time. The used setup consist of multiple depth and color cameras. The gathered data is then meshed by first computing an implicit surface, which is then triangulated with the marching cubes technique [LC87]. Furthermore, different architectures with distinct compression and bandwidth requirements are discussed. The authors investigated different bandwidth usages and processing strategies, and state their advantages and use cases. A use case for low bandwidth usage would be the streaming of depth maps and color images from the server side to perform a full reconstruction on the client device. This requires high computational capabilities on the client side. The second strategy suggests to perform the implicit surface calculation on the server side and stream the surface data to the client. This allows the client to implement different meshing

algorithms based on its capabilities. In the last approach, the fully constructed mesh and texture data are compressed and transmitted to the client. Then, the client has the minimal computational effort, but an efficient mesh compression algorithm is required to save bandwidth.

Bonatto et al. [BRS+16] focus on large, static point clouds and exploit state-of-the-art techniques to achieve high frame-rates, which are required for the use in modern head mounted displays (HMDs). To avoid computationally heavy meshing algorithms, they make use of offline preprocessing steps to estimate the orientation of each surface point. With known surface normals, the point cloud can then be visualized in real-time using the surface splatting technique [ZPVBG01]. This allows to efficiently render a closed and shaded surface from oriented points, independent from the resolution or the point cloud density. Dealing with massive point clouds requires advanced memory management, so called out-of-core algorithms [VM02]. In favor of faster access, the point cloud is stored using optimized spatial data structure layouts. This improves loading times, minimizes working memory usage and prevents wasting processing power on points outside the current view. Furthermore, new points are gradually loaded to increase the point density and thus, improve the level of detail. The rendering in HMDs is especially challenging because the scene is rendered in stereo, ultimately doubling the required rendering time while frame rates of 60-90 frames per second (FPS) are desirable to reduce motion sickness.

2.2 Point Cloud Processing

The processing of point clouds encompasses a multitude of research areas, from the point cloud generation through photogrammetry or sophisticated scanning devices to continuous data compression, object reconstruction and optimized visualization techniques. The following publications cover different aspects of point processing, which represent the state-of-the-art in each respective field.

2.2.1 Input Data Compression

With improving depth-sensing technologies and increasing sensor resolutions, the amount of captured data is constantly rising. Novel compression algorithms specifically designed for dynamic point clouds can help to reduce the required bandwidth for streaming applications. In [KBR+12], the authors propose a lossy compression algorithm for point cloud streams. An octree is populated with the dataset captured at each point in time, in order to detect spatial and temporal redundancies between consecutive point clouds. This information is subsequently used to compress the data stream. Thanou et al. [TCF15] focus on the compression of color information by introducing a graph based approach with feature matching and motion estimation between frames.

2.2.2 Culling and Level of Detail

When dealing with large point clouds, the processing power often represents a major bottleneck, considering that millions of points are fairly common in current datasets. Therefore, it is necessary to reduce the number of processed points to the visible portion of the data. Furthermore, a dynamic level of detail is desirable, to reduce the complexity of dense point clusters and allow interactivity even on large scenes. A fast hidden point removal operator was introduced in [KTB07]. The visibility of a point can be determined without the requirement of surface normals and independently from the screen resolution. The point cloud is first transformed and then the convex hull of the result is computed. If the point appears on the convex hull, it is also visible on the screen. Since convex hull computations are very time consuming and not feasible in real-time, an approximation of the visibility operator, which is taking advantage of current GPU architectures, has been proposed by Machado e Silva et al. [eSEO12]. Dynamic levels of detail for point clouds for efficient rendering are proposed by Renato Pajarola [Paj03]. The author uses spatial data structures to determine visible points and creates levels of detail by calculating the extent of each hierarchy node with the novel concept of transformation-invariant homogeneous covariance matrices.

2.2.3 Reconstruction

Contrary to classic polygonal meshes, point clouds lack vertex neighborhood information. This makes normal estimation calculations very computational expensive and often infeasible for online processing. With improved GPU capabilities, new algorithms are developed to estimate point orientations or entire polygon meshes in real-time. Newcombe et al. [NFS15] propose DynamicFusion, an online mesh reconstruction for non-rigid motions captured with a single depth camera. The algorithm describes each frame as a flow field, which warps the point cloud into an initial estimation of the 3D mesh. This initial mesh, called canonical frame, is constructed from an implicit surface function, which is continuously enhanced with the increasing amount of captured frames. With this technique, it is possible to reconstruct a dense polygonal mesh of dynamic point clouds captured by a single camera. However, fast changes in topology and rapid motions are still challenging. Fusion4D by Dou et al. [DKD+16] tackles this challenge by introducing multiple key frames and numerous camera views. The key frames behave similar to the canonical frame in DynamicFusion [NFS15] but allow to dynamically blend between various key frames to handle larger topological changes. Opposed to the online mesh reconstruction, AutoSplats from Preiner et al. [PJW12] estimates point normals of unorganized large point cloud data. The authors exploit modern GPU capabilities to estimate the surface normal of each projected point in screen space. Multiple fragment shader passes are used to find the k nearest neighbors (kNN) and finally estimate the normal by fitting a plane in the neighboring points.

2.2.4 Point Rendering

Correct illumination of object surfaces is crucial for 3D shape perception. Illumination models require surface information to approximate the reflected and refracted light in order to create realistic shadings. Point clouds typically lack surface normals and require additional processing steps to acquire this information. Boucheny et al. present Eye-Dome Lighting [Bou09], a visualization technique to enhance shape perception for point clouds without surface information. Their technique resembles the screen space ambient occlusion (SSAO) technique [Kaj09] with the most notable difference of neglecting the surface normals. [Bou09] use a view aligned half dome centered at each pixel. The number of points inside this half dome determine the shading intensity. This approach can also be interpreted as an edge detection filter on the z-buffer. Datasets with point normals can take advantage of physically based per pixel lighting models. Dobrev et al. [DRL10] propose an image-space technique to render closed surfaces from sparse point clouds. The idea is to project the illuminated point cloud into screen space and write colors and normals into separate render textures. The point surface is then reconstructed using conditional dilation filters, which attempt to fill holes between the projected points. Botsch et al. [BSK04, BHZK05] pursue a different approach to reconstruct local surfaces for illuminated point cloud rendering. The basic idea is to adopt the surface splatting technique [ZPVBG01] for GPUs. Their surface splatting algorithm renders the albedo color and surface normal as oriented ellipses for each point in different render textures. The overlapping ellipses are then blended and create a closed surface with averaged properties across the rendered ellipses. The final rendering pass uses deferred lighting algorithms for physically correct per-pixel shading with support for multiple light sources.

2.2.5 Correlations with this work

This work combines relevant State-of-the-Art methods and algorithms to achieve the goal of creating a real-time reconstruction and visualization pipeline for dynamic point cloud data. The pipeline is separated in multiple stages, similar to the client-server architecture proposed in [TADB$^+$16]. This allows spreading the pipeline over multiple machines and thus, distributing the required processing power among all devices. Contrary to [TADB$^+$16] and [OERF$^+$16], no meshes are reconstructed in our point cloud processing pipeline. Meshing of point clouds has the big advantage that standard rendering techniques can be used. However, the reconstruction requires high amounts of processing power and mostly relies on temporal consistent inputs to sustain real-time performance. This makes robust real-time mesh reconstruction challenging for fast changing data [NFS15, DKD$^+$16]. The surface reconstruction step used in our work takes another path and calculates per-point surface normals, comparable to [LZZ13]. The performance of the reconstruction in [LZZ13] is bound to the spatial resolution of the volumetric grid. A more flexible alternative is presented in [PJW12] by approximating the reconstruction in screen space. [PJW12] require no temporal coherent inputs and are therefore robust to rapid changes in movement and topology. The point cloud with enriched surface normal information is rendered with a modern surface splatting technique [BHZK05] that is taking advantage

of modern GPU features. This technique is often used in point based visualization algorithms [LZZ13, BRS+16].

CHAPTER 3

Engine Framework and Libraries

In this chapter, we present the engine, framework and libraries used to implement this work. The graphics engine was chosen carefully, as it has a great leverage on how the actual pipeline concept is implemented. The decision is explained in detail and was made due to four significant reasons. Subsequently, the most relevant engine concepts, with respect to this work, are explained. How does it work under the hood? What does the engine provide out of the box? How can it be extended for custom applications? The answers to these questions will be essential for a better understanding of the Chapter 5, where the implementation is explained in detail. Lastly, other third party libraries and their usage are described.

3.1 Unity3D Game Engine

Unity3D, more often just referred to as *Unity*, is a cross-platform game engine developed by Unity Technologies [Uni]. The engine supports the development of 2D/3D games as well as XR applications. The included features encompass a modern real-time rendering engine, a physics engine, spatial sound, animations and a cinematic composition editor, just to name a few. The rendering engine has support for various graphic application programming interfaces (APIs) like OpenGL [Ope], DirectX [Dirb], Vulkan [Vul] and Metal [Met].

Since its first release in 2004, *Unity* is steadily gaining popularity. The statistics published on their company website claim that 34% of the top 1000 mobile games in 2016 were made with their engine. Furthermore, they claim that 90% of Samsung-Gear-VR games and 53% of Oculus-Rift games are made with *Unity* [UPR].

Unity's increasing popularity has several reasons: First, *Unity* uses a very open license agreement, allowing free usage up to a maximal annual revenue and fair annual fees if this threshold is exceeded. This attracts many independent developers and small

companies, leading to a steadily growing community. Secondly, the engine was built with the develop-once, deploy-everywhere mentality. It supports more than 25 platforms [Uni] including Windows, Mac OSX, Android, iOS, Play Station 4, XBox One and many others. This enabled smaller teams to publish their games on multiple platforms with almost no additional overhead. The fourth reason for its popularity is the huge community. There are many forums, books, tutorials and other resources available to support the development of multi-platform games and XR applications, which can lower the entry barrier for beginners and non-programmers. The last reason was probably the main reason we have chosen *Unity* over other competitive engines currently available. *Unity* is labeled as a game engine but is not restricted for game development only. The generic concepts used to build games can be adopted for interactive applications with the need for real-time graphics. At its core, *Unity* can be seen as an high level abstraction framework on top of the underlying graphics API and operating system.

3.1.1 Programming

The *Unity* run-time environment is built on a platform dependent C/C++ core, providing the graphics layer abstraction and operating system specific code. On top of that sits the open source cross-platform Mono framework [Mon]. This framework offers an alternative to Microsoft's .NET framework [NET] and allows the developer to use the exposed run-time functionality in a high-level language like C# or JavaScript.

Most internal processes are hidden from the developer. There are only a few access points to extend and customize the functionality of the engine. Beyond that, the developer can freely write custom classes or include external libraries, which are compatible with the used Mono/.NET version.

3.1.2 GameObjects and Components

Unity scenes are composed from *GameObjects*. *GameObjects* represent the base for everything that can be interacted with, executes code or that is visible in the final rendered screen. Complex scenes contain a hierarchy of numerous *GameObjects* with different purposes. They can also be created and destroyed at run-time.

A *GameObject* holds various components, which define its behavior and appearance. Per default each *GameObject* has a transform component which basically represents the model matrix and, thus, describes how the object is placed in the virtual world. Other components provided by the engine are, for example, the camera, light or mesh component. These components are directly integrated in the engine and provide basic functionalities for games and other interactive applications.

To extend the *GameObject* with custom behavior it is possible to write component scripts. Custom component scripts can be created by extending the internal *MonoBehaviour* base class. The *MonoBehaviour* has access to all other components on the owning *GameObject* and can then be used to implement various routines to change or extend the object's properties. The scripting framework makes use of different method hooks to control and

execute the component in the underlying game loop. Commonly used hooks are the *Start* and the *Update* method, which are called for initialization and before each frame is rendered.

3.1.3 Mesh Filter and Mesh Renderer

Imported 3D models are stored as mesh assets and can be rendered by applying it to a *GameObject*. The components responsible for drawing a mesh are the *Mesh Filter* component, and the *Mesh Renderer* component. The *Mesh Filter* holds the reference to the imported 3D model and is a simple geometry provider for the rendering component. A *Mesh Renderer* component receives the geometry from the *Mesh Filter* and issues the actual draw commands to the graphics API. Additional options allow changing the behavior of the renderer and improving the visual quality or performance. The most prominent settings are shadow casting/receiving and environmental reflections. In order to control the visual appearance, every renderer holds one, or multiple references to materials, used to render the different parts of the mesh.

3.1.4 Materials and Shaders

In modern computer graphics an object's appearance is defined by the shader used to draw its geometry on the GPU. Shader program combinations, shading properties and textures make up the look of a rendered object and are, for this reason, bundled together into so-called materials. This is a convenient way to reuse and control the visual characteristics of multiple objects in a scene. Many objects with the same material are also batched in the graphics pipeline and therefore handled in a more optimized way to gain performance. *Unity* includes physically based materials to create realistic surfaces and lifelike scenes. The standard material allows rendering metallic as well as diffuse surfaces, with additional color textures, normal maps, displacement maps and much more.

Shaders make up a large portion of the modern programmable graphics pipeline. It has become an absolute requirement for every modern rendering engine to support custom shader programs. In *Unity* shaders are declared in the proprietary *ShaderLab* syntax, which wraps multiple shader programs and shader passes up in one file. The shader programs are then written in high level shading language (HLSL) and subsequently cross-compiled for every supported platform. For applications where OpenGL support is sufficient, it is also allowed to use OpenGL shading language (GLSL) in *Unity's ShaderLab* files.

Additionally to the common shader programs, *Unity* also supports compute shaders. This shader type exposes general-purpose computation on GPU (GPGPU) technology to the graphics pipeline and can be used for arbitrary computations on the GPU exploiting the highly parallelized architecture. This special shader type is discussed in the Section 3.1.5.

Unity has a strong focus on platform independence, this is especially challenging if a consistent rendering output is expected on all supported devices. Different platforms support different graphic APIs, and each device has different hardware capabilities. To support all of them and still guarantee to deliver identical output on every device would mean to restrict the complete engine to the smallest common denominator, in terms of GPU features. However, in order not to restrict modern applications to the limits of older GPUs or mobile GPUs, *Unity* tries to separate the shader features by compilation target levels. Each level supports different GPU feature sets on different graphic APIs. Therefore, it is the developer's responsibility to provide fall-back solutions for newer features.

3.1.5 Compute Shader and Compute Buffer

If the target platform supports GPGPU features, compute shaders can be used to run massively parallelized algorithms on the GPU. This functionality in *Unity* closely resembles DirectX 11 DirectCompute technology [Dira]. Data arrays can be sent to the GPU memory and are stored in compute buffers. Prior to the compute shader execution multiple compute buffers can be bound to the program as an input, output or both. Furthermore, it is possible to bind compute buffers to vertex, geometry and fragment shaders[1] in order to directly use the generated data for rendering.

3.1.6 Command Buffer

As mentioned before, *Unity* can be seen as a high level abstraction layer on top of the native graphics pipeline. Besides the out-of-the-box rendering functionality, it is also possible to issue low-level graphics commands to the GPU directly in the custom component code. This can be achieved through the *GL* class, which exposes similar basic functionality as most graphic APIs. However, this is often not sufficient, as we want to extend and modify *Unity's* rendering behavior directly. Issuing *GL* commands results in either rendering before or after *Unity's* default rendering workflow. This approach makes it impossible to use or alter intermediate rendering results from *Unity's* pipeline. At the time this thesis was written the only way to extend the default graphics pipeline within *Unity's* framework, were so called command buffers. Command buffers are a list of low level graphics instructions injected into the default rendering pipeline. This allows great control over various pipeline stages, and enables to write custom render components for structures and visuals, which were not considered by *Unity's* default pipeline, while still using the overall advantages of the engine. A command buffer may, for example, contain custom shader passes to add contents to the internal G-Buffer [AMHH08], extending the deferred shading pipeline. Even if this feature allows many customizations, it is restricted by the position of predefined injection hooks in the default pipeline. A more flexible approach introduced in a recent release is shortly discussed in Section 5.8.2.

[1] Pixel shader in DirectX [Dirb]

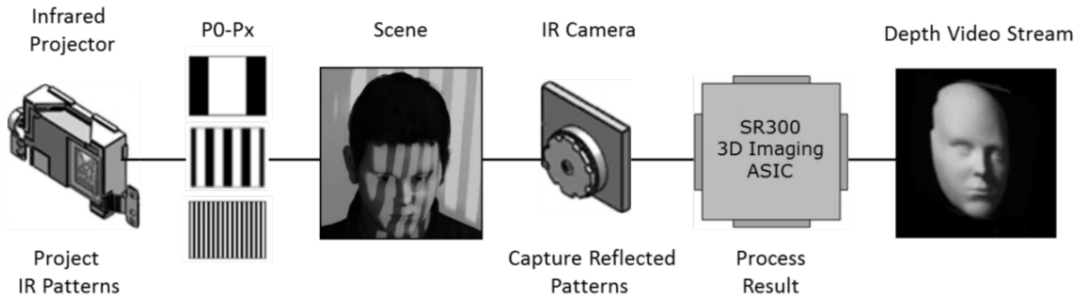


Figure 3.1: Depth capturing process in the RealSense SR300 [Int16]

3.1.7 Native Plugins

Within a custom component script, it is also possible to call code from external C/C++ libraries. These libraries are called native plugins and can be used to include extended functionality, which may only be accessible through native code. Additionally, popular libraries like OpenCV (discussed in Section 3.3) can be included by using native Plugins. The platform independence must be guaranteed by the developer of the plugin interface, as different platforms require different plugin binaries. Native plugins have also access to the graphics API through a low-level native plugin interface. Allowing to directly issue GPU commands from the plugin code makes massive data exchanges between the plugin and the application in many cases obsolete.

3.2 LibRealSense

This thesis covers real-time point cloud processing of live sensor streams. For this purpose, we used the Intel RealSense SR300 RGB and depth (RGB-D) camera. The term RGB-D camera is frequently used for devices incorporating color and depth sensing technologies. The depth capturing in the RealSense SR300 functions with the structured light technology. The general process is outlined in Figure 3.1. An infrared laser projector illuminates the scene with multiple consecutive patterns, these patterns are reflected and captured by the infrared camera, which then translates the distorted patterns into per-pixel depth values [Int16]. The camera is suited for a close range usage and provides a full HD (1920×1080) RGB color stream and VGA (640×480) depth stream at 30 FPS. The RGB data stream has a color depth of 8 bits per channel, resulting in 24 bits per pixel. The depth values are streamed as 16 bit integer values, representing the distance in millimeters.

The RealSense software development kit (SDK) [Lib] is open source and developed in C++. In addition to the color, infrared and depth stream, the SDK provides access to the camera's intrinsic and extrinsic (relative offset of the color camera to the depth sensor) calibration information. The mapping between color and depth information is

handled internally by the SDK. Since the color and infrared camera have mismatching aspect ratios (color 16:9, infrared 4:3), the color values do not cover the complete depth map. This results in uncolored points in upper and lower regions in the point cloud.

The LibRealSense SDK can be integrated as a native plugin (Section 3.1.7) for the usage in *Unity*. In this case, the plugin serves as an interface to control the SDK directly from code executed by *Unity*. For live-streaming applications, performance is crucial, therefore the additional overhead of copying each received frame from the unmanaged memory (in the plugin) to the managed memory (in *Unity*) should be avoided. Implementation details and how this issue was tackled can be found in Section 5.3.2.

3.3 OpenCV

OpenCV [Bra00] is an open source library and provides a broad collection of high performance computer vision tools and algorithms. The library is written in C/C++ and has additional interfaces for Python and Java. OpenCV has become the de-facto standard for image processing applications, it implements a multitude of processing tools as well as high level object recognition algorithms and can handle almost every common image file format.

There is no official support for the usage of OpenCV within *Unity* and its C# scripting framework. However, there are some third party plugins available for purchase in the *Unity* asset store. A free alternative to charged plugins is the EmguCV open source project [Emg]. The EmguCV project is a cross platform .NET wrapper for OpenCV. This library can therefore simply be included in *Unity* C# projects, as it supports most desktop and mobile platforms.

If only a limited amount of OpenCV's functionality is needed, building a custom plugin may also be a suitable choice. *Unity's* native plugin interface is shortly described in Section 3.1.7. All functionalities required from OpenCV can be outsourced to a C/C++ plugin. This means the native plugin serves as a simple wrapper for the needed features. For certain *Unity* applications, which rely on a lot of OpenCV's functionality, this approach is less recommended as maintaining a continuously growing interface would get too labor intensive.

CHAPTER 4

System Concept

From the raw point cloud data, to the final rendered reconstruction placed in a virtual environment, several steps are necessary. The proposed visualization pipeline is structured in four pipeline stages with modular character (Figure 4.1): The *Input Handler* stage, followed by the *Data Culling* and the *Surface Renconstruction* stage and finally concluding with the *Rendering* stage. Each stage can have a variety of implementations depending on the needed functionality, from photo-realistic reconstruction of live sensor data, to coarse high performance previews of large environmental scans. Furthermore, the proposed modular architecture extends the idea of a separate data-providing server-side and a visualizing client-side [TADB+16]. Modules may be separately executed on different physical machines by implementing remote connections between their interfaces. This flexibility allows high-quality reconstructions on low performance devices, while heavy processing algorithms are outsourced to remote machines. The following sections will explain each pipeline stage in detail and introduce possible implementation variants.

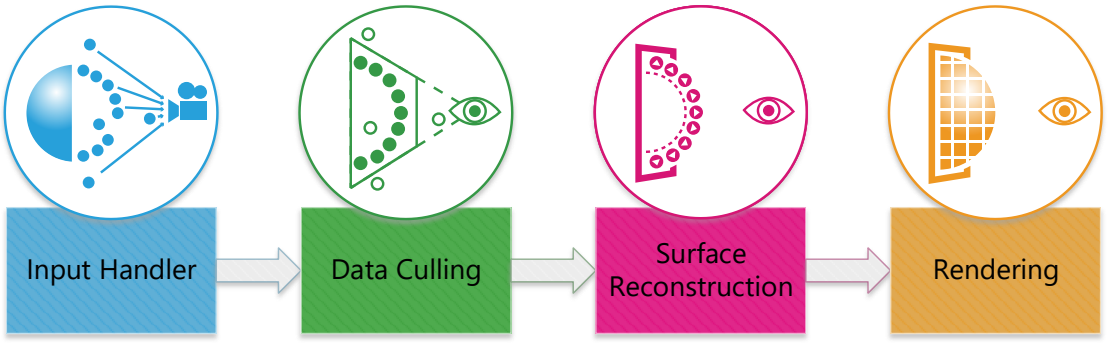


Figure 4.1: Flowchart of the visualization pipeline stages and their execution order.

4.1 Input Handling

The first stage of the visualization pipeline covers the initial data acquiring process. Point clouds can be represented and stored in many different forms. The simplest form are a set of points, i.e. positions in 3D space. This type of data could be the result of a sampled 3D model or preprocessed sensor data. Simple point sets can be extended with further information about the original sampled signal, like surface colors, orientation and curvature. Points with surface information are referred to as *surfels*, derived from the term *surface element* [PZVBG00]. In this thesis, we reference both simple point position and *surfel* datasets as points or point clouds, regardless of their extra information. Where it is necessary to point out that additional surface knowledge is required, it is explicitly mentioned that this part refers to *surfel* datasets.

The input handling stage is responsible for the loading, parsing and storing of the raw point cloud data. This can range from loading binary data files, reading raw sensor outputs, to handling live streams from connected devices or remote data sources. The following sections discuss some distinct formats, which need to be considered when working with heterogeneous point cloud data sources.

4.1.1 Point Cloud Files

Various file formats can be used to store point cloud data. Commonly accepted are file formats known from 3D meshes. These formats have already been established in the computer graphics community and provide enough flexibility to store points or *surfels*. Exemplary file formats which are commonly used are the Wavefront file format (.obj), polygon file format (.ply) and object file format (.off), just to name a few. Every file format capable of storing 3D geometry is generally also able to hold point cloud data, as most file formats store additional information on a vertex basis. This means that *surfel* data, like per point color and normals, are also usually needed for each vertex in a typical polygon mesh. However, these files were not intended for point cloud usage and therefore lack flexibility as more precision or arbitrary information per element are desired. The point cloud library (PCL) [RC11] introduced a new file format (.pcd) specifically tailored for point cloud data. The biggest advantages of this format are higher read/write performances, storing of organized point clouds and the support for various data types.

The input handler stage is responsible for parsing these files and storing the gathered point or *surfel* information directly onto the GPU memory.

4.1.2 Sensor Data

In addition to the usage of preprocessed 3D point cloud data, it has advantages to store and process raw 2D sensor outputs. Modern depth sensing cameras provide depth maps, containing the measured distance for each pixel in the projected sensor image plane. Moreover, RGB-D cameras often grant access to color images as well as confidence maps

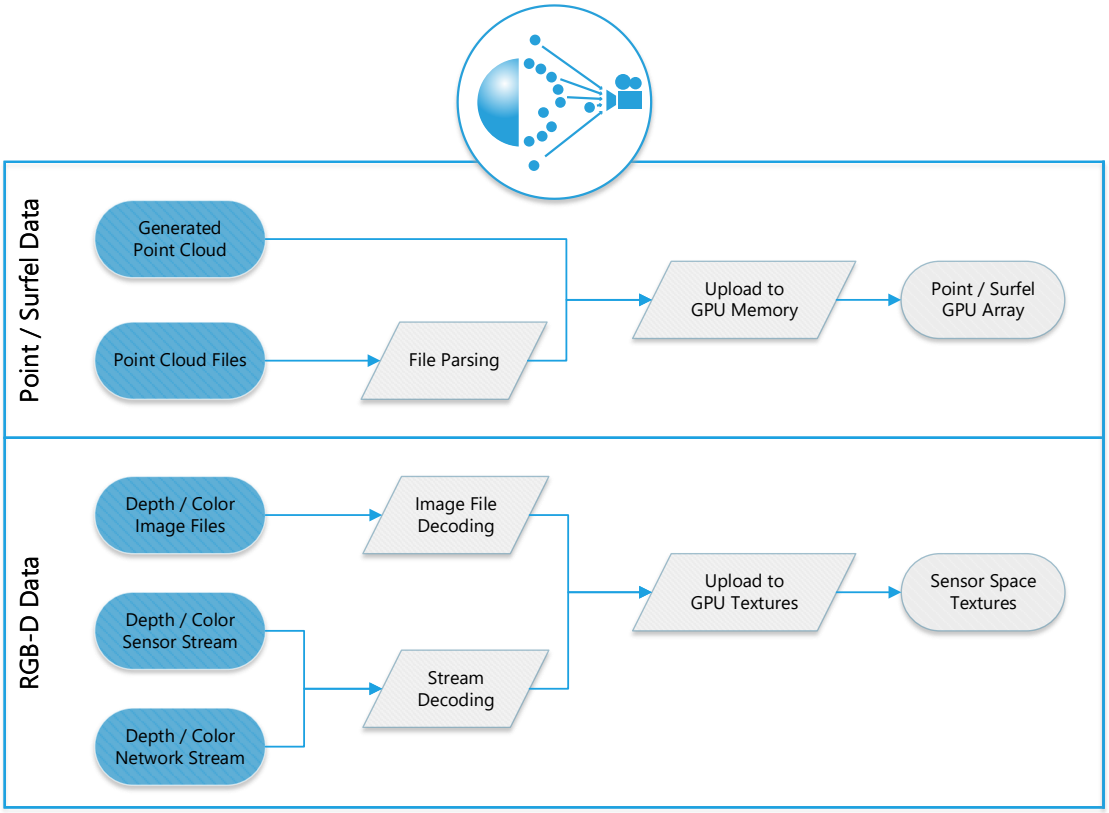


Figure 4.2: Flowchart of the *Input Handler Stage* and its input/output interface. The flowchart outlines the internal workflow of the stage, from the provided input, the main processing steps and the final output. The vertical lanes differentiate between the two fundamentally different data types, the point based data and the raw sensor image data. The input for the *Input Handler Stage* is heterogeneous and requires customized implementations for each source, while the output is reduced to the two main types, point arrays and sensor textures.

and near-infrared intensity values. This additional information is very useful and often desired in specific visualizations. Furthermore, the input handler stage should not be simply reduced to a 3D point cloud handler, since the 2D domain of raw sensor inputs has convenient advantages in subsequent stages (e.g. image filtering for noise reduction, see Section 4.2.3). This leads to the fundamental distinction of the two main input handler types: *Point Input Handler* and *Image Input Handler*.

4.1.3 I/O Interface

Both input handler types need to manage data from specific sources, depending on the used module. Each source can provide either static or dynamic data, the input handler is then responsible for the memory management of the currently relevant information.

The distinction between point and image data leads to different outputs of this stage. *Point Input Handler* provides a reference to the GPU memory where the array of points or *surfels* is stored. The *Image Input Handler* holds an array of GPU texture references, for each sensor image.

4.2 Data Culling

After retrieving the raw data from the various input sources, it is necessary to reduce the further processed data. This is essential to minimize the additional overhead of processing unwanted information or data, which is not visible in the final rendered image. This process is generally called *culling* and can be achieved in many different approaches.

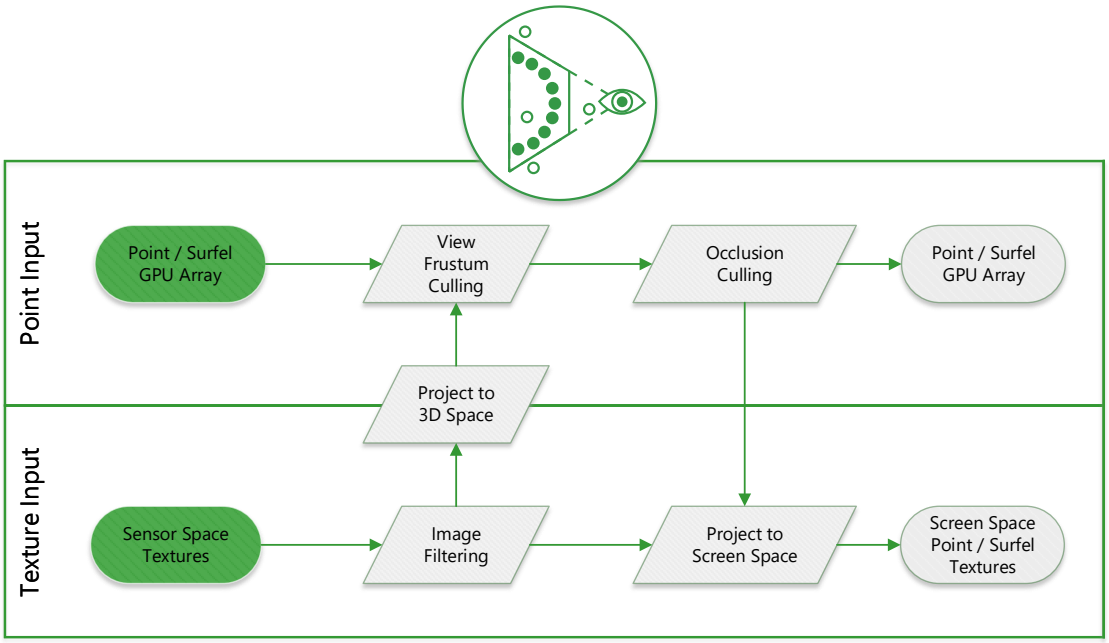


Figure 4.3: Flowchart of the *Data Culling Stage* and its input/output interface. The flowchart outlines the internal workflow using the provided input resulting in two distinct outputs. The two lanes separates our two main data types, point and sensor image data. Note that the texture input and output are defined in different spaces. The image input lane expects raw depth values in sensor space while the image output consists of screen space attribute textures (Similar to a G-Buffer [AMHH08]). Furthermore, this stage allows to transform the data type to adapt the output for subsequent stages.

4.2.1 View Frustum Culling

A simple and efficient way to render huge amounts of polygons in modern graphic applications is to reduce the number of draw calls issued by the CPU, to lower the GPU workload. This is achieved by discarding objects, which are not contained in the camera's

view frustum and thus not present in the final rendered image [AMHH08]. Testing if an object, or its bounding box, is inside of the view frustum is a simple geometric operation. To further reduce the processing time, the objects in the scene are organized into spatial search data structures, e.g. an Octree or k-d tree [AMHH08]. Static point datasets can be easily organized in such a way, because the overhead of creating the spatial data structure has only to be considered once at loading time. Dynamic datasets suffer from the continuous rebuilding of the data structure and hence losing its initial benefit.

4.2.2 Occlusion Culling

Even though view frustum culling assures that only points which can be seen by the camera are sent to the GPU, occlusions by other polygons or points are still a problem. Various occlusion-culling techniques evolved in the past to reduce the cost of unnecessary overdraw [AMHH08]. Furthermore, point clouds suffer from different point densities on the projected image. Surfaces which are farther away are usually represented by more points than surfaces which appear closer. Especially the back-facing points cannot be as easily discarded as with typical triangle geometries, if no surface information is available. This results in the redundant processing of potentially occluded but densely represented surfaces, while the occluding surface seems very sparse. Detecting occluders and discarding occluded point subsets before they are rendered could save a lot of performance. This point cloud specific occlusion culling technique is referred to as *hidden point removal* [KTB07, eSEO12].

4.2.3 Filtering and Clipping

The previously made distinction of point datasets and image datasets also applies to the data culling stage. Image datasets are usually unprocessed sensor outputs and can therefore be refined before projecting them back into 3D space. Raw 2D sensor data can be enhanced by applying image processing filter operations. Such filters are able to manipulate the depth values in order to remove outliers or reduce sensor noise [GTKK13]. Additionally, sensor depth images can be clipped using manually adjusted near and far planes in the sensor camera space. This allows discarding points when projecting them into the 3D sensor camera space. Points too close or far away from the sensor's optimal working range are often erroneous and should therefore be clipped, as they are not reliable. After the projection, image inputs are treated like point datasets and go through the same culling steps.

4.2.4 I/O Interface

In the most trivial case, all input points are simply passed through and nothing needs to be culled. In a more optimal situation, parts of the passed input points are outside the view frustum or occluded and, thus, culled. Then the output is a subset of the initial point cloud. Filtered depth images are first projected into the 3D space and then passed through the same culling steps. The standard output of the data culling stage is therefore

a reduced GPU point array, for point and image data inputs. An alternative thereto is the output of multiple textures with the required information stored in the color channels, similar to a G-Buffer [AMHH08]. For this, each point is projected into screen space, where only the closest points to the viewer are kept, discarding occluded points. Screen space outputs are required for screen space surface reconstruction algorithms, which are discussed in Chapter 4.3.

4.3 Surface Reconstruction

Point positions and colors are not enough to render a point cloud with a visually appealing closed surface. The surface orientation is crucial for lighting and reflection calculations. If the desired surface normal is not available in the first place, an offline preprocessed step is necessary to recalculate it for the complete dataset. However, this offline preprocessing step is not always possible, especially when the point cloud needs to be captured and rendered in real-time. This section describes the pipeline stage, which provides the functionality to reconstruct the surface of the remaining point cloud subset, after the data culling stage.

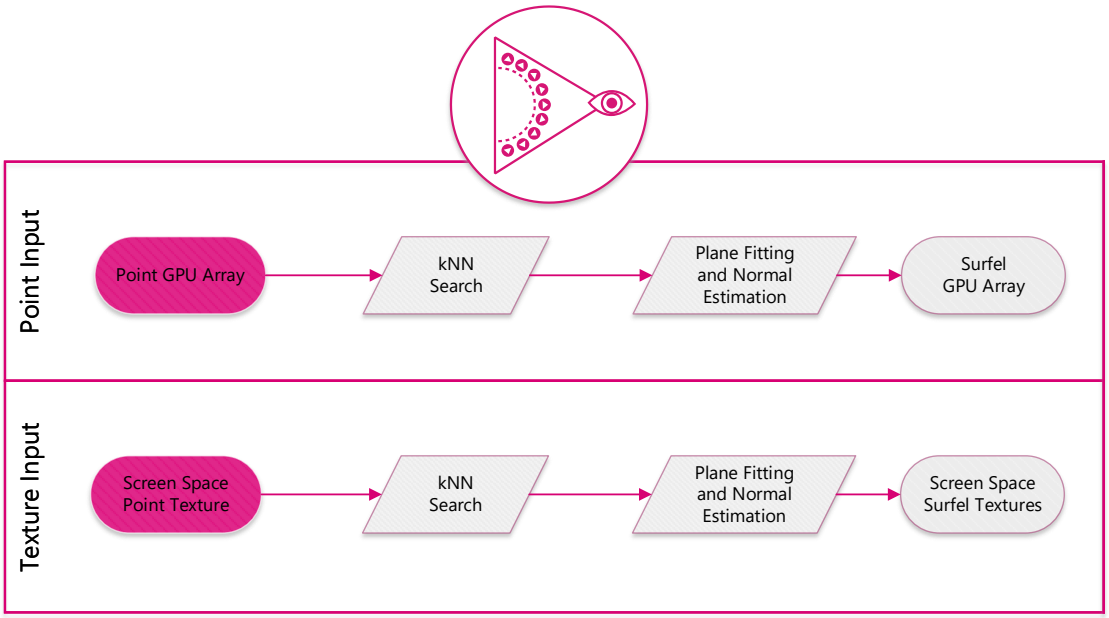


Figure 4.4: Flowchart of the *Surface Reconstruction Stage* and its input/output interface. The flowchart outlines the internal workflow, the required input and the expected output. Vertical lanes separate the point data from the screen space attribute textures. The internal processing for this stage was depicted using the kNN plane fitting approach [PJW12, Sch16], but could vary depending on the implemented algorithm.

4.3.1 Screen Space Reconstruction

A common simplification of the surface reconstruction problem is the reconstruction of the surface points visible in screen space. This is obviously an approximation of the real surface, because only the visible part of the surface is considered [PJW12]. Nevertheless, this approach has still big advantages like the real-time performance, robustness and sufficiently convincing visual results. The reconstruction algorithms attempt to find the kNN of each point and subsequently fit a regression plane into this point subset. The normal of the estimated supporting plane directly corresponds to the point's surface normal. A more detailed explanation of such algorithms can be found in Section 5.5.

4.3.2 I/O Interface

The input requested from the surface reconstruction stage is either a point array or the pre-projected screen space textures created in the data culling stage. The exact procedure of estimating the surface of the passed point cloud is implementation dependent. The expected outputs are the resulting *surfel* properties: mainly the surface normal and in some advanced cases also the tangent, co-tangent and curvature of the originally sampled surface.

4.4 Rendering

The final stage draws the point cloud with varying realism, depending on the properties of the initial dataset or the additionally gained properties from the various stages. The point cloud can be represented as simple colored dots all the way to a closed, shaded surface with reflections and shadows.

4.4.1 Billboard Points

Without surface information points can only be displayed as camera facing quads, so called billboards [AMHH08]. To reduce the aliasing artifacts due to the quad edges, pixels outside the desired disk radius are discarded. Billboard points are a rudimentary visualization which is often good enough to give a fast first impression of the captured data. This rough depiction is often used to visualize various properties of the point cloud, encoded in the color. For the visualization of meta data, like the actual sensor depth values, surface normals or the near infrared intensity, this representation usually suffices.

4.4.2 Blended Splats

For a visual reconstruction of the discretely sampled object it is necessary to render a continuous surface without holes. These surfaces are approximated with so-called splats [ZPVBG01]. Similar to billboards, splats are generated quads centered in the *surfels* position. The surface normal is then used to orient the quad such that the quad's normal is aligned with the *surfel* normal. Additional surface properties can be used to

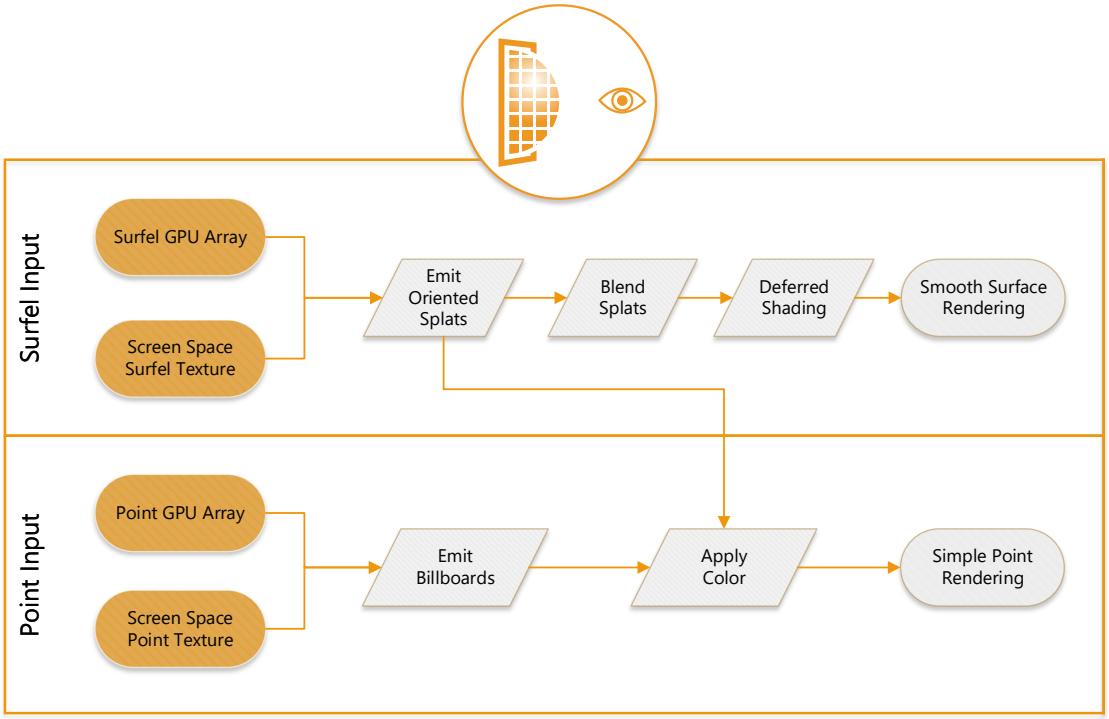


Figure 4.5: Flowchart of the *Rendering Stage* and its input/output interface. The flowchart outlines the internal workflow of the stage, from the provided input, the main processing steps and the final output. Contrary to the previous flowcharts (Figures 4.2, 4.3, 4.4), the lanes in this figure do not separate the input by type (points or images) but instead by the information they contain. Specifically *surfel* input, containing positions and surface normals, and input containing plain point positions. Note that the additional surface information can always be omitted or used for simplified visualization paths.

further distort the disk. Finally, the overlapping splat colors and normals are blended together to achieve a smooth appearance. The resulting surface is then shaded according to the material properties and the environmental effects.

4.4.3 I/O Interface

The rendering stage accepts point and *surfel* data, either as data arrays or attribute textures defined in screen space. The output is the final rendered point cloud, ready for the scene composition in the standard rendering pipeline.

CHAPTER 5

Implementation

This chapter is dedicated to the implementation of the proposed visualization pipeline. It gives an in detail description on how the presented stages from Chapter 4 are integrated in the chosen framework (Chapter 3.1). The focus of this work is the visual reconstruction of dynamic, constantly changing point clouds, in real-time. Algorithms and techniques used in the various pipeline module implementations were chosen to process and render live sensor data in an efficient manner while maintaining a robust reconstruction and high-quality visuals. Some alternative module implementation are provided to demonstrate the flexibility of the proposed pipeline. Figure 5.1 shows the implemented modules from the four different pipeline stages, introduced in Figure 4.1. Starting from an input handler module and ending at a renderer module, each connection shown in Figure 5.1 represents a valid pipeline setup. In addition to the general implementation explanations, Section 5.8.5 gives an overview of the connection, inheritance and cooperation between all implemented classes of the novel visualization pipeline. It provides an outline of the base modules, as well as the sample implementations of each stage, and can be seen as the pipeline's backbone.

5.1 Point Cloud

In order to seamlessly integrate point cloud rendering into *Unity*, we based our implementation on *Unity*'s proven concepts and structures. *Unity's* representation of interactive scene elements are *GameObjects*. Customizable components define the object's behavior, appearance and functionality. 3D meshes are represented by *GameObjects* with built-in components responsible for managing the mesh data and rendering it to the display. A more in detail discussion can be found in Chapter 3.1.2. Based on the same concept we build our novel point cloud rendering pipeline.

The novel collection of custom components in our implementation allows displaying a point cloud with adaptable processing performance and customizable visual quality.

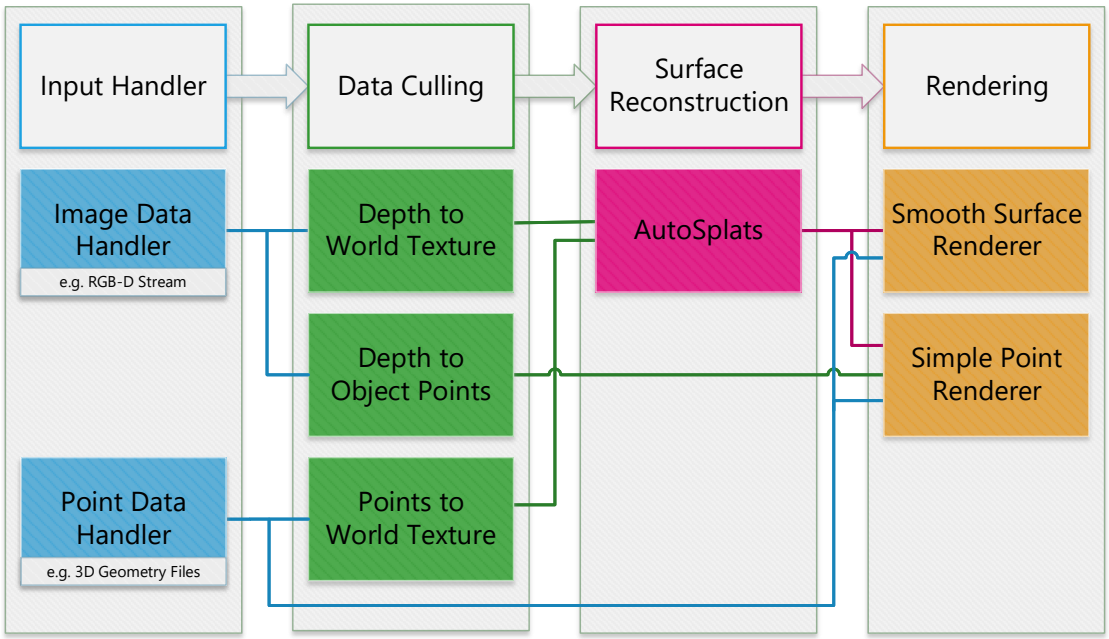


Figure 5.1: Flowchart of the implemented visualization pipeline stages, their execution order and possible combinations. Each connected set of modules (from left to right) represents a valid pipeline setup.

5.2 Base Pipeline Component

The basic implementation of a pipeline stage is crucial for the reliable functioning of the custom work-flow. The most convenient way to execute custom and reusable code in *Unity* is a *MonoBehaviour* component script. Analogous to the mesh and rendering components attached to *GameObjects*, we implement each pipeline stage as a distinct component. This creates a flexible and modular way to configure the different pipeline variations.

Component scripts are usually initialized independently on start-up or when they are first instantiated on run-time. This initialization process is executed sequentially on the main update thread, meaning that any long running initialization would restrain the start-up time of the application or freeze the application mid-frame. As our pipeline components potentially need to process millions of points by parsing a data source or to populate a spatial data structure, an asynchronous approach is preferable.

We achieved this non-blocking initialization with *Unity's* co-routines. A co-routine is a function, which allows execution over multiple frames, thus, preventing the application from getting unresponsive.

Still, concurrent initialization is only possible for every pipeline stage at a time, as subsequent stages may be dependent on each other. To prevent premature initialization

on an undefined predecessor state, each pipeline component defines its dependencies. The initialization process is then delayed until all dependencies are completely ready. For example, a renderer component is dependent on the data provider with the highest priority.

5.3 Data Handler Components

Data handler components retrieve, store and continuously update raw point cloud information. The data handler stage proposed in Chapter 4.1 may be implemented differently for each input source and data type. This work encompasses data handler components handling geometry files, images, sensor and network streams. Every data handler component inherits from the abstract *BaseDataHandler* class, which provides basic query methods like the maximum number of points or which supplementary information is stored within the input data. More specialized functionality can be found in the two extending abstraction layers *BasePointDataHandler* and *BaseImageDataHandler*. Both classes serve as generalizations of the two main input formats: 3D points/*surfels* sets and unprocessed 2D sensor outputs.

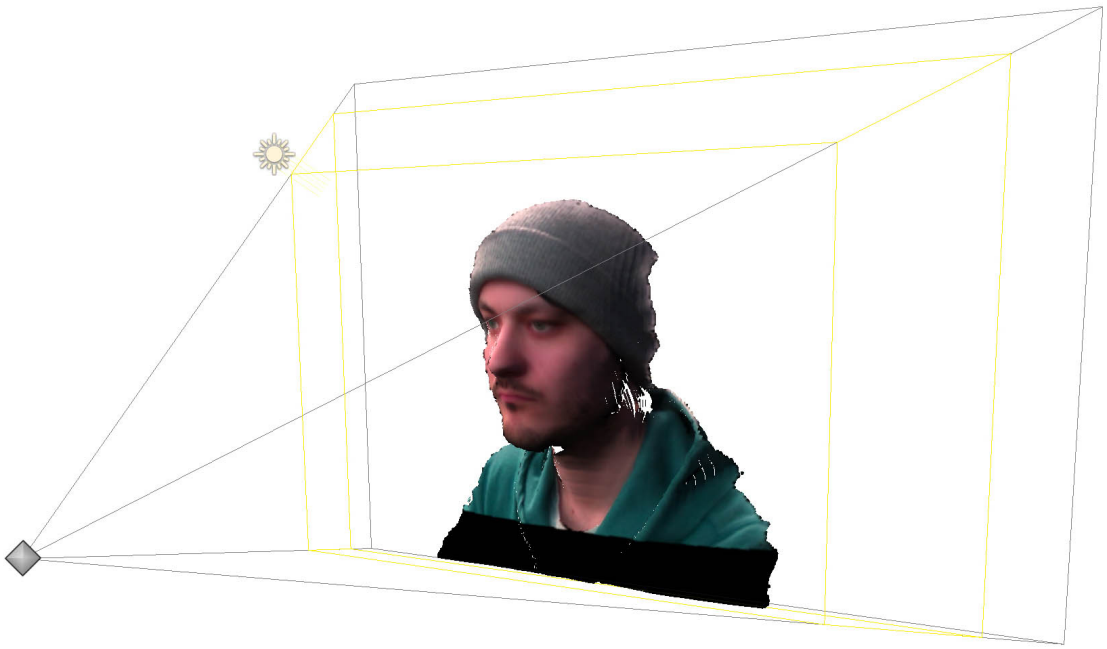


Figure 5.2: Live streamed data from a depth sensor and color camera. The streams are handled in a native plugin and directly uploaded to the GPU. The visualized frustum represents the view of the sensor in a virtual 3D coordinate system. The yellow area highlights the region of interest, points outside this frustum are discarded. The surface of the persons head is reconstructed and rendered in real-time. The sun icon shows the position of the virtual light source, used to recalculate the lighting of the captured scene.

5.3.1 Point Data Handler

Point data handlers load and store 3D positions and additional *surfel* information, if they are available beforehand. In the scope of this work, we mostly deal with easily manageable point cloud sizes and are therefore able to directly upload the data to GPU memory at once. Larger point sets require out-of-core loading schemes in order to efficiently handle the limited memory.

All graphics APIs provide some sort of vertex buffers to store custom array data on the GPU memory. *Unity's* graphics abstraction layer, unfortunately, does not allow direct access to vertex buffers. A naive method to circumvent this restriction is to exploit the built-in mesh class. Points can be passed as vertices to a mesh instance, along with their color and normal vector. An advantage of this workaround is that the engine handles the GPU upload and all bindings before each draw call. This seems rather convenient at first, but using these concealed built-in classes leads to a great shortcoming in flexibility. *Unity* does not provide any functionality to persistently modify[1] vertex buffers on the GPU. Furthermore, the mesh class only supports a predefined number of attributes per vertex, additional information has to fit into this predefined schema or cannot be used.

A more flexible alternative to this approach is to abandon common vertex buffers and use compute buffers[2] instead. Compute buffers are intended for the generation and manipulation of arbitrary data in compute shader programs. The buffers can also be bound to any other shader stage. This makes it ideal for our point cloud storage. Point and *surfel* data is stored in compute buffers where we can manipulate the data as we desire and finally read the required data for rendering. However, only newer devices with state-of-the-art graphics APIs support compute shaders and compute buffers. Thus, we still use mesh objects as a fallback solution for simple visualizations.

An exemplary implementation of a point data handler is the *PointCloudGenerator*, which generates rudimentary point clouds of basic geometric shapes. This data handler is mostly used for testing purposes. A component for real life use cases is the *PointFileLoader*. This component is used if the point cloud is present as a set of points stored in a geometry mesh file (e.g. processed 3D scan outputs). The file is loaded and parsed using a suitable parser for the given file extension. In the scope of this work, we implemented parsers for the Object File Format (.off) and the ASCII variant of the Polygon File Format (.ply).

5.3.2 Image Data Handler

Image data handlers store raw sensor output and update the memory if the input changes. This component abstraction is designed for sensors with 2D depth image outputs, like time of flight cameras, sensors using structured light or preprocessed stereo camera rigs. Additionally to the depth map, most sensors provide an infrared intensity image and a rgb-color image for each captured frame. The latter needs to be mapped into the sensor image space in order to match the points captured in the depth map.

[1] e.g. stream-output stage on DirectX and transform feedback on OpenGL.
[2] The matching technology in OpenGL is called *shader storage buffer*.

Image data handler implementations provide the intrinsic camera parameters of the depth camera for the usage in all subsequent stages. These parameters are required to re-project the depth values into 3D sensor space. Furthermore, the sensor dimensions define the size of the allocated GPU textures. Image data handlers hold two textures, which are used to store the raw sensor output, one for the depth image and another for either the infrared intensity image or the mapped color image.

The *ImageFileLoader* component is an example of an image data handler. This component loads images for the depth and the color or infrared image. The image decoding and uploading is handled by a custom plugin using OpenCV [Bra00]. The integration of external code in *Unity* was discussed in Chapter 3.1.7. OpenCV provides great flexibility when loading images of any type and color depth. High color depths are especially important for depth maps, to preserve higher precision and avoid quantization artifacts.

We implemented the *CameraStreamInputHandler* and the *TCPInputStreamHandler* component to handle continuous data of depth, color and infrared information. Live sensor data is either received from a remote streaming source or by polling the sensor API directly. Most devices only provide APIs programmed in C/C++, so it is necessary to write custom wrappers and integrate them as engine plugins. In this work, we integrated the Intel RealSense SR300 near-range depth-sensing camera. We poll the images in a separate thread and write them into the designated texture on each frame. Decoupling the sensor polling from the main thread is crucial, as our goal was to achieve interactive frame-rates, unbound by the low capturing speed of the sensor. The sensor API wrapper was built with future extensions in mind. A generic camera interface hides the sensor specific implementation and facilitates additional sensor adaptations.

Unmanaged Memory Our image data handlers involve native plugins to handle the diverse requirements of the raw depth sensors inputs. Memory in native C/C++ plugins is unmanaged and *Unity's* C# scripting framework cannot directly access it. If we still want to use *Unity's* cross-platform graphics abstraction, we would need to copy each sensor frame from unmanaged to managed memory, before we could start the GPU upload. This process is very time consuming, especially with high frame-rates and large image resolutions. Keeping the data in unmanaged memory would be favorable. Fortunately, *Unity* provides rendering event hooks, which were introduced to trigger rendering calls in native plugins. This enables us to process each sensor frame directly in the plugin and let *Unity* call the GPU upload at an appropriate point in its rendering pipeline, through the implemented event hook. Allowing the complete native graphics API to be accessed through native plugins may sound convenient, however, it comes at the additional overhead of platform dependency, forcing custom plugin implementations for each supported operating system.

5.4 Culling Helper Components

The rendering performance of large geometry sets can be drastically increased with efficient culling techniques [AMHH08]. Culling describes the removal of redundant data, which is not visible in the final output and may therefore be neglected early on, to save otherwise wasted processing time. Our proposed culling modules implement highly flexible and efficient culling techniques for dynamic datasets. Furthermore, the culling helper components are also responsible for potential transformations of the dataset to different coordinate systems. This is, for example, necessary for the further processing of depth maps from depth sensors. Depth maps are defined in the camera space of the sensor device, for further processing a transformation into world space or screen space can be advantageous (Section 5.5.1).

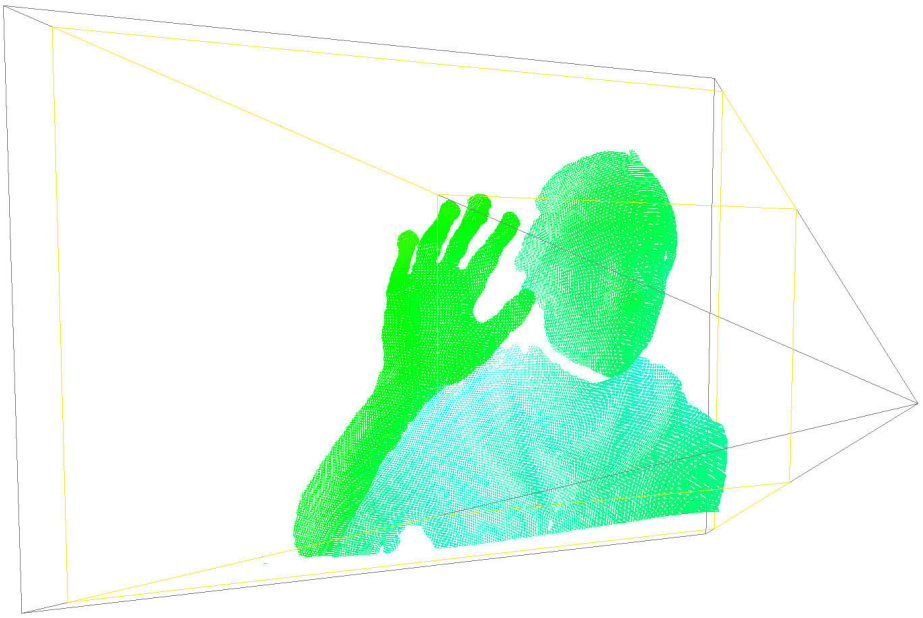


Figure 5.3: View frustum (grey) of a depth sensor. The yellow area marks the area of interest while everything else is culled. The color used to render this simplified visualization encodes the depth value, from green on the near culling plane to blue on the far culling plane.

5.4.1 Depth To Object Points

Our implementation focus in this stage was the efficient processing of continuously changing live sensor data. Spatial data structures for fast point removal are great for large static datasets, but often lack flexibility when it comes to dynamic, fast changing data. Efficient and more flexible approaches needed to be considered for live sensor stream processing. The implementation of our depth processing module exploits the

2D representation of the raw input data (Figure 5.4). Image filters provide an efficient way to reduce sensor noise and enhance the overall sensor output [GTKK13]. The filters used in our *DepthToObjectPoint* component are proven image processing tools, like the median filter and the bilateral filter [TM98]. McGuire [McG08] introduces a highly efficient median filter for modern programmable GPUs. That implementation performs in a single shading pass, exploiting the high parallelization capabilities of the fragment shader stage. Additionally to the median filter, we incorporate a subsequent modified bilateral filter [TM98], to smooth the noisy depth values without destroying characteristic edges. Our custom modification prevents the smoothing over invalid depth pixels, which would otherwise result in unwanted bleeding of invalid values into the dataset. The *BaseDepthCullingHelper* class implements both image filters with a 3×3 and a 5×5 kernel. This reduces unwanted depth outliers originating from sensor noise and smooths noise stricken surfaces.

Figure 5.4: Unprocessed raw depth map (left) and the same depth map after distance culling and filtering (right).

Subsequently we project the filtered 2D depth image into the sensor camera coordinate system. With the use of the intrinsic sensor calibration parameters, we are able to re-project each depth value from the 2D image space to the 3D sensor space:

$$d = \Im(x, y) \cdot s \tag{5.1}$$

$$\vec{p} = \begin{pmatrix} d\frac{x-c_x}{f_x} \\ d\frac{y-c_y}{f_y} \\ d \\ 1 \end{pmatrix} \tag{5.2}$$

Equation 5.2 depicts how the point $\vec{p}$ in sensor space is acquired. The depth value d is calculated from the depth stored in the image $\Im$ at the image coordinates (x, y), scaled with the factor s to adjust the sensor depth units to meters (which is *Unity's* default unit). Usually sensor metrics are defined in millimeters, in that case $s = 10^{-3}$. The parameters c_x, c_y and f_x, f_y state the principal point c and the focal length f of the depth

camera, provided by the intrinsic calibration. The re-projected point $\vec{p}.$ is then defined in the 3D sensor space, which can also be interpreted as the point clouds object space.

We execute a compute shader program to perform the projection on each valid depth pixel in the image and store the 3D points in a compute buffer. Commonly image processing on the GPU is executed in fragment shader programs. The texture fetching cache allows efficient access to surrounding pixels for various filter operations. However, projecting the depth values in the image can be done independently for each pixel and does not require any texture-tile caches. Furthermore, compute shaders provide great control over the patch size and the number of spawned threads, while avoiding the overhead of the rasterization and output stages in the standard rendering pipeline.

Depth values outside the needed working range are clipped using a configurable near and far plane in sensor space. The result of this process is shown in Figure 5.4 and illustrated in 3D in Figure 5.3. The number of valid depth projections is unknown at initialization time, only the upper bound is defined by the image resolution. To avoid wasting GPU memory we use the append compute buffer type. As the name suggests, an append buffer allows to append new points to the existing buffer without manual re-allocations or initial assumptions on the maximum size. This is a large advantage, as we only need to further process this subset of points, greatly reducing the number of draw calls.

The implementation of the described method can be found in the *DepthToObjectPoints* component. It is required to transform image data input handler outputs to 3D space. Afterwards, sensor inputs can be treated in the same way as datasets provided by point data handlers. Nonetheless, the points created from projecting depth values into 3D space still lack surface information and can therefore only be displayed as plain point clouds. For further processing an intermediate representation, mandatory for the AutoSplats [PJW12] algorithm, is required.

5.4.2 Intermediate World Texture Format

AutoSplats [PJW12] is a surface reconstruction algorithm, which estimates the surface normals of the currently visible point cloud in screen space. Projecting the points into screen space can be loosely interpreted as a simple form of view frustum and occlusion culling as hidden points are discarded and not further processed in the pipeline. Therefore, we decided to integrate this part of the algorithm into the culling helper stage. Making this component also requisite for point data handler outputs if no *surfel* information is known.

Projecting points and storing the output in textures is the designated work-flow of the standard rendering pipeline. Rendering points, which are stored in compute buffers is achieved with the *indirect* draw call of the graphics abstraction layer. This draw call does not require any vertex buffer bound to it, the number of desired vertices is simply passed as a parameter to the call. Instead of the vertex information, only a single index is provided as an input to the vertex shader stage. This index may be used to address a bound compute buffer, which serves as a vertex buffer replacement. All further steps are

performed as usual. Projecting points from a depth map in a single pass is achieved in a similar fashion. As we do not know how many valid points were captured by the depth sensor, we have to assume that all points are potentially projected. Therefore, we issue an indirect draw call with vertex number matching the number of pixels in the depth map. In the vertex shader the index is used to fetch the corresponding texel[3] in the sensor image. Invalid pixels are discarded by projecting them outside the clipping box.

DepthToWorldTexture and *PointToWorldTexture* are the exemplary implementations leading to this intermediate format. Figure 5.1 illustrates that either one of these culling helper modules is required for our subsequent implementation of the surface reconstruction stage. Both classes implement the *IWorldPositionTexture* interface to abstract the output access for the next stage.

5.5 Surface Processor Components

Modern computer graphics and rendering algorithms mostly operate on the visible outer hull of volumetric objects, often referred to as the boundary representation [Hof89]. A crude discretization of complex real-world surfaces are triangle meshes, which are easy to store and geometrical calculations can be greatly simplified. Triangles are especially convenient as the three vertices in 3D space always describe a plane and each point on the spanned plane inside the triangle can be uniquely parametrized. The neighborhood of a vertex can be found by iterating though the vertices of all the triangles, which contain the initial vertex. The orientation of a triangle face is not unambiguously defined, it can either be front facing or back facing. Therefore, it is necessary to agree on the orientation by considering the winding order of the vertices, being either clockwise or counter-clockwise. If the winding order is known, the face normal can be simply calculated by taking the cross-product of two edges in the correct order.

Point clouds are a discretization retrieved by capturing point samples of real-world surfaces. Point samples commonly hold information on the position and color of the observed surface. The lack of connectivity between neighboring points increases the required effort to reconstruct the surface orientation.

In general, two distinct approaches can be pursued to gain surface information from point cloud datasets. [NFS15, DKD$^+$16, TADB$^+$16, YGX$^+$17] apply algorithms similar to the triangulation of volumetric data. An implicit function is defined to parametrize the complete surface, this implicit function is then triangulated with the use of techniques like the marching cube algorithm [LC87]. Continuous mesh morphing and deformation, often referred to as geometry fusion, allows reusing the generated mesh over multiple frames for temporal consistent outputs and real-time efficiency [NFS15, DKD$^+$16, YGX$^+$17]. These algorithms work very well in a defined context and result in stunning and detailed dynamic meshes. However, real-time meshing is a fairly new field of research because the computational effort for real-time results is very high, bringing current generation GPUs

[3] short for texture element, a sampled value from a texture [AMHH08]

to their limits. State-of-the-art algorithms for real-time surface reconstruction often require initial assumptions and constraints regarding the target object, the topological changes or movements speeds [NFS15].

An alternative approach, used by [PJW12, LZZ13, Sch16], does not intend to reconstruct and maintain a whole triangle mesh, but rather estimates the surface normal on each sampled point and use point rendering techniques to visualize the point cloud's surface [PZVBG00, BSK04, BHZK05]. AutoSplats [PJW12] estimates the surface normal in screen space by finding the kNN for every point and fitting a plane into the local neighborhood. The normal of the plane is then oriented by estimating the interior of the closed object, or by assuming that all points are facing the viewer. The rendered splat size is also given by the determined kNN radius. An optimization of the GPU shader approach was proposed by [Sch16], using GPGPU programs and 2D search structures leading to a large performance boost. [LZZ13] roughly estimate the surface normals of their volumetric dataset from the gradient image calculated by the weighted camera views. Both techniques [PJW12, LZZ13] are approximations of the real surface, as only points in projected screen-space are considered. However, this approximation delivers sufficiently realistic results with real-time performance, while running on consumer hardware.

5.5.1 AutoSplats

The reconstruction module implemented in our real-time point cloud processing pipeline is based on the AutoSplats algorithm, proposed in the previous work of Preiner et al. [PJW12]. AutoSplats estimates the surface normal in screen space by finding the kNN for every point and fitting a plane into its local neighborhood. The algorithm operates on a frame-by-frame basis and does not require temporal coherence or previous knowledge of the input data, which improves the robustness when dealing with rapidly changing point clouds with large changes in the topology.

For each frame AutoSplats carries out the following steps in multiple shader passes:

1. Initial projection of the point cloud.
2. Estimate initial radius for each point.
3. Iteratively refine the radius depending on the enclosed number of neighbors, until the desired number of neighbors (k) is reached.
4. Accumulate neighboring point positions for the covariance matrix construction.
5. Calculate the surface normal by solving the singular value decomposition (SVD) of the covariance matrix.

The result are per-frame reconstructed normals for dynamic point clouds. The algorithm achieves real-time performance, even for large point clouds, through resourceful usages of

fragment shader programs. Elementary per-point operations are executed in single render passes, exploiting the parallelization on the GPU. This *parallel splat communication* [PJW12], introduced in AutoSplats, is used for simple position accumulation or range searching tasks. It functions by drawing screen space patches for each point and using the hardware accelerated *add* and *max* blending functions to obtain the specified goal on the projected position or radius buffer textures.

We use the same technique in our reconstruction module to estimate per-point normals. The input for the algorithm is a screen space buffer texture, holding each projected point's world position. The initial projection, is already performed by the previous culling helper stage. This step was separated from the main algorithm as it enables other surface reconstruction algorithms (e.g. [Sch16]) to reuse this intermediate point cloud representation. An additional texture created from the initial point projection contains each point's compute buffer index. This index is used to query additional properties directly from the buffer without allocating additional texture memory. The world position, however, is written directly into a separate four-channel texture (RGB channels for the x, y and z coordinate. Alpha channel for the point radius). Steps 2 and 3 from the original algorithm where skipped, favoring higher performance over reconstruction accuracy. Step 4 is carried out in two shader passes. First, the average point position of each fixed sized point neighborhood, is calculated in a single fragment shader pass. The results are again stored in a screen space buffer texture for a simplified fragment shader access in the subsequent passes. In the second shader pass, the scaled covariance matrix for each neighborhood is constructed using the previously calculated average positions. Finally, in step 5, the SVD of these covariance matrices is solved, to calculate the supporting plane's orientation. This orientation is a good estimate of the point surface normal and is stored in a per-point normal buffer texture. Contrary to the remarks in the AutoSplats publication, we use the geometry shader stage to dynamically draw quads for the *parallel splat communication*, instead of using OpenGL's point size (point sprite) feature, which is not supported on DirectX.

Our simplified reconstruction approach neglects the dynamic, per point, radii search and directly jumps from step 1 to step 4 using a manually adjusted radius, which is constant for each point. This is a crude approximation of the real kNN search as not every point has the same number of neighbors in the end. This leads to some minor artifacts when changing the point of view, because the number of effective neighbors may change due to the screen space projection, resulting in slightly different normals. However, this process, as also stated in the work of Preiner et al. [PJW12], accounts for almost 80-90% of the complete run-time of the algorithm. Depending on the number of neighbors and the point cloud size, this step requires up to 10 full-screen fragment shader passes. This can drastically affect the performance with higher resolutions and larger point densities.

In our evaluation we show that skipping this dynamic radii search allows much higher performance, while still resulting in satisfying per-point normals. Especially when dealing with point clouds with a low variation in densities, this adaptation delivers satisfying results while being less computational expensive and therefore, also more suitable for low

performance devices.

5.6 Renderer Components

The rendering of point-based models on modern GPUs is only supported up to a certain degree. Our goal is to render points with as little memory and processing overhead as possible. Point primitives, contrary to standard triangle primitives, are an obvious choice and supported by all major graphic APIs. Positions, colors and other attributes are passed to the GPU in a vertex list. Rendering point primitives with default settings results in pixel sized rectangles, which in some cases may lead to an acceptable representation, if the dataset density allows for a pixel accurate, gapless visualization. Otherwise, the point cloud may be too sparse and therefore, having visible holes between the displayed points or even be barely visible on higher display resolutions. Most graphic APIs support the point size attribute, enabling to draw a larger rectangle around the point, instead of just filling a single pixel. This feature is often referred to as point sprites [AMHH08]. Varying screen resolutions can then simply be compensated by adjusting the point size. However, this feature is not supported on all platforms and should therefore be used with care. Furthermore, point sprites are always screen-aligned, perspective distortions and correct depth values have to be reproduced inside the drawn rectangle. This technique is known as impostor rendering [AMHH08].

A modern approach to substitute the outdated hardware implementation of point sprites, are geometry shaders. With geometry shaders it is possible to dynamically generate new primitives for each input vertex. Allowing to create a quad for all of the originally drawn points. Correct projection and depth values are then taken care of in the default rendering pipeline. This approach was also pursued in our rendering pipeline as *Unity* does not offer consistent support for the point size feature across platforms. A downside to this method is that the number of drawn vertices is quadrupled since two new triangles are created per point. This has a clear impact on the overall performance, compared to the more simplistic point sprites.

Our implementation contributes two distinct point cloud rendering components which are explained in detail in the following sections.

5.6.1 Simple Point Renderer

The *SimplePointRenderer* component allows displaying plain point clouds by drawing a quad on the position of each point. If a point contains surface information, the quad is created in object space and first oriented along the surface normal before the perspective projection is applied. Hereby a simple and efficient estimation of the surface is rendered. The output looks similar to the non-overlapping surface splats in Figure 5.6. Point clouds without *surfel* information are represented by screen-aligned quads. Round edges enhance the simple visualization and mitigate aliasing effects. Additional attributes can be visualized by encoding them into the color.

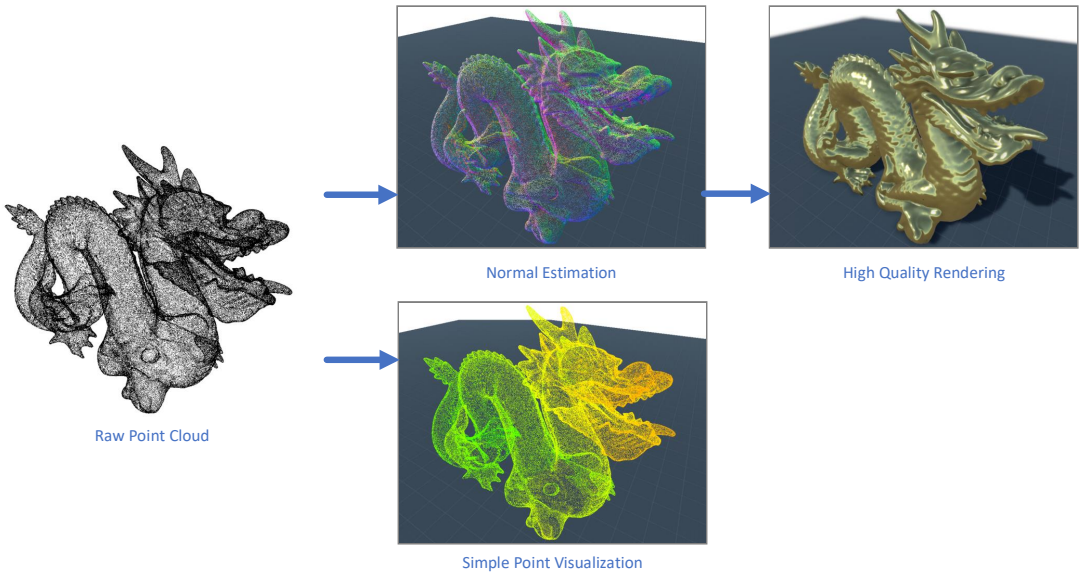


Figure 5.5: Different visualization paths depending on the provided or processed surface information. Starting from the raw point cloud, either present as point positions or depth values defined in sensor space, we can visualize the point cloud as it is, with additional color-coded information, or as an alternative we use the existing or estimated normals to reconstruct a closed surface with photo-realistic shading.

The *SimplePointRenderer* module is especially useful for straightforward point cloud visualizations on hardware with limited memory and processing power. Point based input handlers require no additional preprocessing, this input handler and renderer combination represents the minimum pipeline setup which can be used to display a point cloud. Also, image based input handlers benefit from this simple and efficient visualization, e.g. for fast sensor previews with color-coded depth values.

5.6.2 Smooth Surface Renderer

Assuming that the point cloud contains surface orientation information, this component is able to reconstruct a perceptibly closed and smooth surface. Furthermore, it integrates the point cloud into *Unity's* lighting system and makes use of all light sources, shadows and reflections. The implemented rendering technique is called *surface splatting* and first introduced by Matthias Zwicker et al. [ZPVBG01] in 2001. Later adaptations for modern lighting models and GPUs were proposed by Mario Botsch et al. [BSK04, BHZK05] a few years later. But even today this technique is still an efficient solution which can be integrated into modern state-of-the-art deferred shading pipelines and provide stunning high-quality results.

The concept of *surface splatting* is similar to painting differently colored dots onto an object in the real world. The larger the painted dots the more they start to blend into

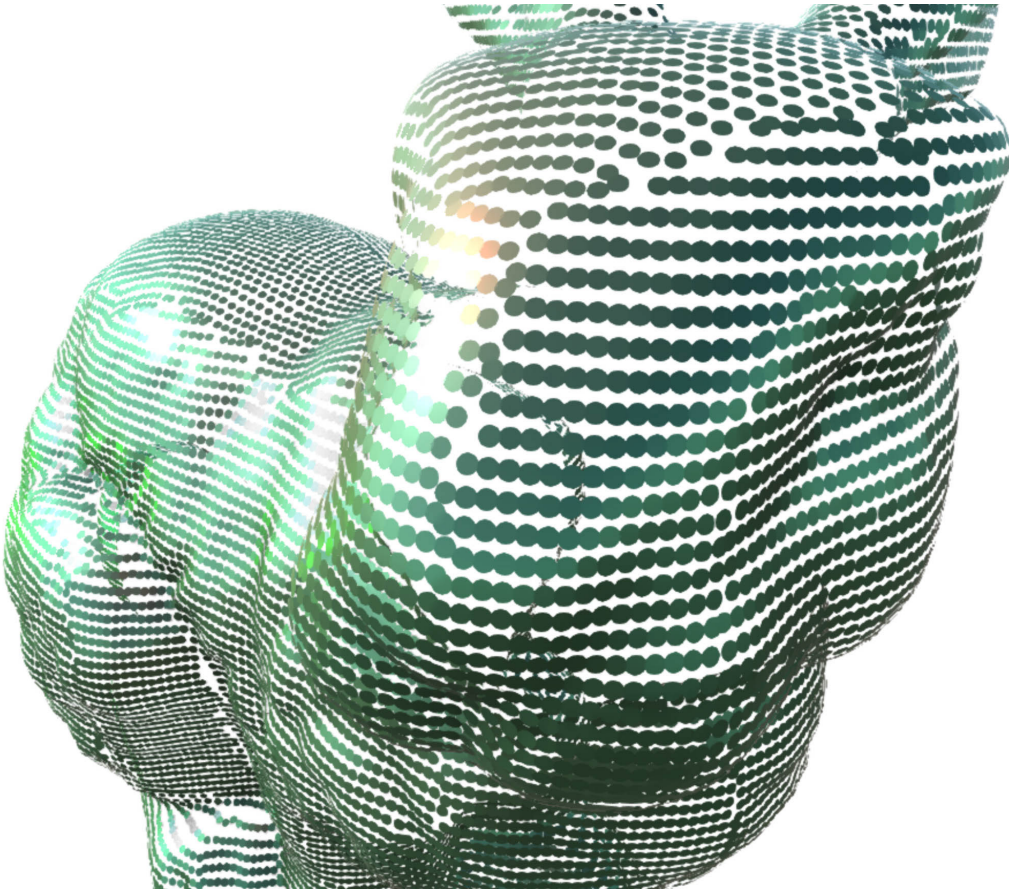


Figure 5.6: Surface splats made visible by decreasing the splat radius. Point attributes like color and *surfle* orientation are propagated across the splat and blended with overlapping neighboring points.

each other and the gaps between the dots vanish. These painted dots or *splats* are rendered as oriented disks at each point position. The size of these disks has to be chosen carefully, in order to create a sufficiently large overlap. Surface splats with reduced radius size can be seen in Figure 5.6.

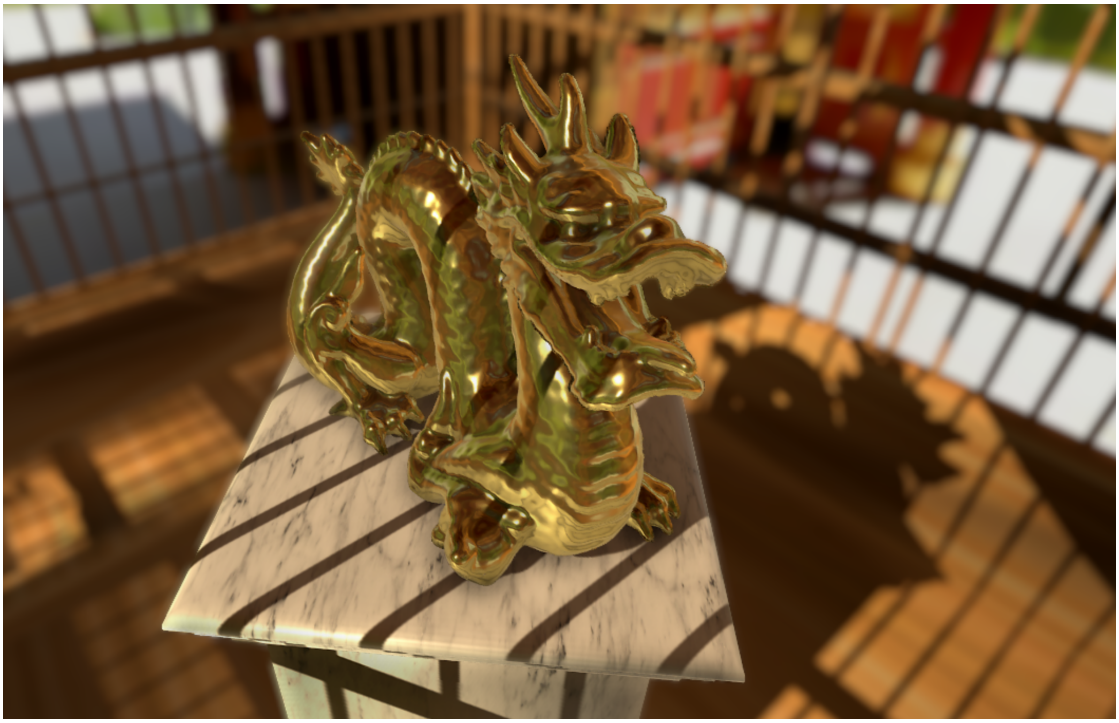
The surface splatting technique is then performed in three passes [BHZK05]: First, the visibility of the *splats* is determined by performing a depth only pass to speed up subsequent passes with the early-Z test [AMHH08]. In the second pass colors, normals and other attributes are rendered into multiple render buffers and blended together. By taking the weighted average of the attributes, smooth transitions between colors and normals of adjacent *splats* are generated. The weights are sampled from a precalculated Gaussian distribution using the Euclidean distance of the current pixel to its splat center. Finally, the blended attributes are divided by their accumulated weights and written into *Unity's* deferred G-Buffer [AMHH08]. This way *Unity's* illumination pipeline takes care

of the shading and environmental reflections.

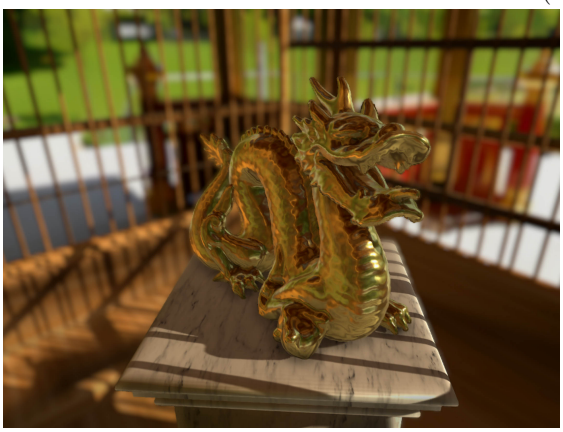
Shadow Casting *Unity* provides built-in real-time cascaded shadow mapping for directional lights. While this works out-of-the-box for *Unity's* renderer components, extending custom renderers with shadow functionality is not easily possible. It is possible to substitute default shadow shaders with custom shaders for built-in materials and renderers, but it is not allowed to inject arbitrary shader passes into the shadow collection stage. *Unity's* shadow collection stage is a highly optimized closed system and does not offer command buffer hooks or other possibilities to access the generated shadow map. Nevertheless, we provided real-time shadow casting support for static point clouds. Implementing a custom separate shadow collection stage with the same feature set as provided from *Unity* is not trivial and would be out of the scope of this work. Thus, we implemented a custom workaround still allowing to use *Unity's* sophisticated shadow mapping. Our approach was to generate an additional supporting mesh for the point cloud, which serves as a shadow caster for *Unity's* shadow system. The shadow caster mesh consists only of vertices from the original dataset and uses *Unity's* built-in renderer component with special materials and custom shaders to ensure that the shadow caster is only visible in the rendering passes responsible for the shadow map generation. The mesh renderer is then automatically called from the engine, drawing overlapping view-aligned quads, to create an approximated depth footprint of the point cloud from the light's perspective. An example of the dynamic point cloud shadow is shown in Figure 5.7. As already discussed in Chapter 5.3, working with *Unity's* built-in mesh type is not flexible enough for continuous updates and dynamic re-allocations of large vertex sets. Therefore, we limited shadow support to static point data handlers.

5.7 Data Provider Interface

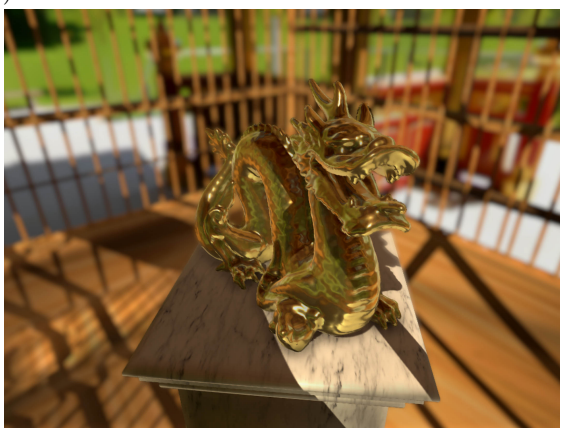
The main input source for our custom rendering components are data providers. Every component, which holds data that can be used to render the point cloud, implements the *IDataProvider* interface. This interface grants access to functionalities needed to display the point cloud. A big advantage of this abstraction layer is that the data provider itself is responsible for the data binding and issuing of the drawing commands. The renderer is unaware of the data type itself and only has to call the appropriate shader programs. On initialization, the renderer components dynamically search for an appropriate data provider, in the pipeline. If multiple suitable candidates are found, the data provider which comes last in the rendering pipeline is chosen. This order is called data provider priority (Input Data Handler < Culling Helper < Surface Processor). Our *SimplePointRenderer* implementation accepts data provider with compute buffer point data and mesh data (which can be used as a fallback, see Chapter 5.3.1). The *SmoothSurfaceRenderer* has higher requirements, as it only accepts compute buffer data with additional surface normal attributes. If no compatible data provider is found, the renderer is disabled triggering an error message. A list of implemented data providers and their properties is shown in Table 5.1.



(a)



(b)



(c)

Figure 5.7: Dynamic real-time shadow casting and receiving on the dragon dataset with reconstructed normals. The scene shows the combination and interaction between point cloud objects and classic triangles mesh based geometry. The dragon point cloud is placed on a marble pedestal inside a wooden cage. The gold-like material has a polished appearance with characteristic environmental reflections. The cage casts shadows onto the pedestal as well as the point cloud, at the same time is the dragon casting shadows onto the environment and itself. Figure (a) depicts the scene illuminated with a virtual light source projecting the dragon silhouette onto the cage, Figure (b) and (c) show the same scene with light coming from different directions. Additional post-processing effects, like SSAO and depth of field, improve the final result considerably.

Data Provider	Data Type	Normals	Priority
BasePointDataHandler (5.3.1)	Comp. Buffer | Mesh	From input	1
DepthToObjectsPoints (5.4.1)	Compute Buffer	No	2
BaseSurfaceProcessor (5.5.1)	Compute Buffer	Yes	3

Table 5.1: Different data provider implementations. The data type defines how the data needs to be rendered, and can be used to check if the renderer component is compatible with the specified type. Not all components are inherently able to provide surface normals, e.g. *DepthToObjectPoints* solely process depth maps to compute 3D points. Surface normals can then only be gained with further processing steps, e.g. with subsequent stages extending the *BaseSurfaceProcessor* component. The priority of a data provider tells the rendering component which component is the most relevant, if multiple data providers are present.

The *IDataProvider* as an additional abstraction layer allows introducing new pipeline components without additional integration overhead.

5.8 Restrictions

Although *Unity* generalizes a variety of modern GPU features for numerous platforms, there are still many functionalities not accessible by *Unity's* rendering framework. This section discusses some of the restriction we had to face in the development process of this work. Some of these issues are already mentioned in the previous sections.

5.8.1 Source Code and Extensibility

Unity has different licensing models, all of them provide the same engine features and just differ in support and additional cloud services. The engine's source code is not included in any of these licensing models. *Unity* states to grant source code access only with a special license, which is only available through direct contact. However, *Unity's* documentation is extensive and detailed for the most parts. The lack of source code access is not really an issue as long as the built-in features are sufficient for the project. Nevertheless, extending core functionality can be painful and often not even possible. For example, the renderer component is crucial for many of *Unity's* rendering features, like light-map baking, shadow map generation or real-time global illumination. Extending this component would be desirable to create custom objects, which exploit *Unity's* rich rendering pipeline. However, the component's class structure is hidden and required interfaces are not accessible.

5.8.2 Default Rendering Pipeline

At the time of writing this thesis, the only possibility to add functionality to *Unity's* default rendering pipeline is through command buffers. Command buffers are a list of

graphic API instructions, which can be inserted at predefined hooks in the rendering pipeline. With this feature, it is possible to add custom post-processing effects to the final rendered output, or to alter the G-Buffer [AMHH08] after its generation. For a variety of rendering algorithms, this type of access and execution is sufficient. Nonetheless, the default pipeline stays untouched and can only be extended in some predefined instants. Many internal processes, like the shadow map generation, stay a black box and cannot be altered or accessed (discussed in Chapter 5.6.2). This issue is hopefully solved by a new feature, the scriptable rendering pipeline, which was announced for a future release (announced for *Unity* 2018.1, current version *Unity* 2017.3). The scriptable rendering pipeline allows creating custom rendering pipelines in a C# script, giving more control over the internal rendering.

5.8.3 GPU memory

The graphics abstraction layer provides useful platform independent rendering functionality. Unfortunately, basic GPU memory access can be difficult to achieve. Vertex and index buffers are not accessible outside the built-in mesh component and cannot be arbitrarily manipulated. An independent vertex array implementation would be favorable to allow custom implementation of geometry classes or other arbitrary data structures. However, the engine does provide native memory pointers for the manipulation in native plugins. If platform independence can be neglected, this enables to use specific graphic API features, which are not abstracted by *Unity*, e.g. stream-output stage on DirectX and transform feedback on OpenGL. As an easy to use alternative, we switched to compute buffers, which are able to store arrays of arbitrary structs on the GPU and allow easy access and manipulation in the different shader programs (discussed in Chapter 5.3.1).

5.8.4 Lights and Shading

Unity's rendering engine supports two different types of shaders. Generally, they are referred to as unlit shaders and surface shaders. Unlit shaders represent classic vertex/fragment shader combinations, known from the programmable graphics pipeline. Whereas a surface shader only represents the part of the fragment shader where the inherent fragment properties are calculated and loaded, like albedo color, surface normal and smoothness factor. The surface shader function is then internally called by *Unity's* default fragment shader implementations and the final per-pixel lighting is calculated. Standard vertex/fragment shaders are labeled as *unlit* because they are not easily integrated in the engine's lighting system. If interaction with dynamic lights in the scene is desired, the documentation recommends using surface shaders instead, as they take care of the complex light handling code and the developer can focus on the essential surface properties of the object. The disadvantage of surface shaders is the limitation that comes with this simplification. As already stated in the previous paragraphs, *Unity's* rendering strongly relies on its built-in renderer and mesh components. Surface shaders assume to be applied to the default renderer component drawing the standard mesh type. For our point cloud implementation, we abandoned mesh components early on,

so we had to dismiss surface shaders as well. However, we still wanted to integrate our custom renderer into the engine's convenient lighting system. Fortunately, *Unity* supports deferred rendering as an alternative rendering path. With full access to the G-Buffer [AMHH08] and the appropriate invocations of our command buffer, we were able to inject the reconstructed point cloud into the lighting system (See Chapter 5.6.2).

5.8.5 Class Inheritance Structure

This section outlines the overall inheritance structure, shown in Figure 5.8. The modular implementation of our pipeline requires a strict inheritance structure. The base class for all components is *Unity's* built-in *MonoBehaviour* class. Overriding special methods ensures that our custom code is executed by the engine on specific events. The abstract *BasePipelineComponent* class (Introduced in Section 5.2) encapsulates functionalities all stage components share, like the asynchronous initialization. Each pipeline stage then inherits from a base class representing the basic interfaces for each module. The *InputHandler* requires an additional abstraction layer for the two main data types, point and image input data. The explicit classes shown in this figure are all sample implementations included in this thesis. New input sources, culling techniques, reconstruction algorithms and rendering methods can easily be added by extending the respective base classes.

Implementation relations of the *IWorldPositionTexture* (Section 5.4.2) and the *IDataProvider* (Section 5.7) interfaces are also shown in the diagram.

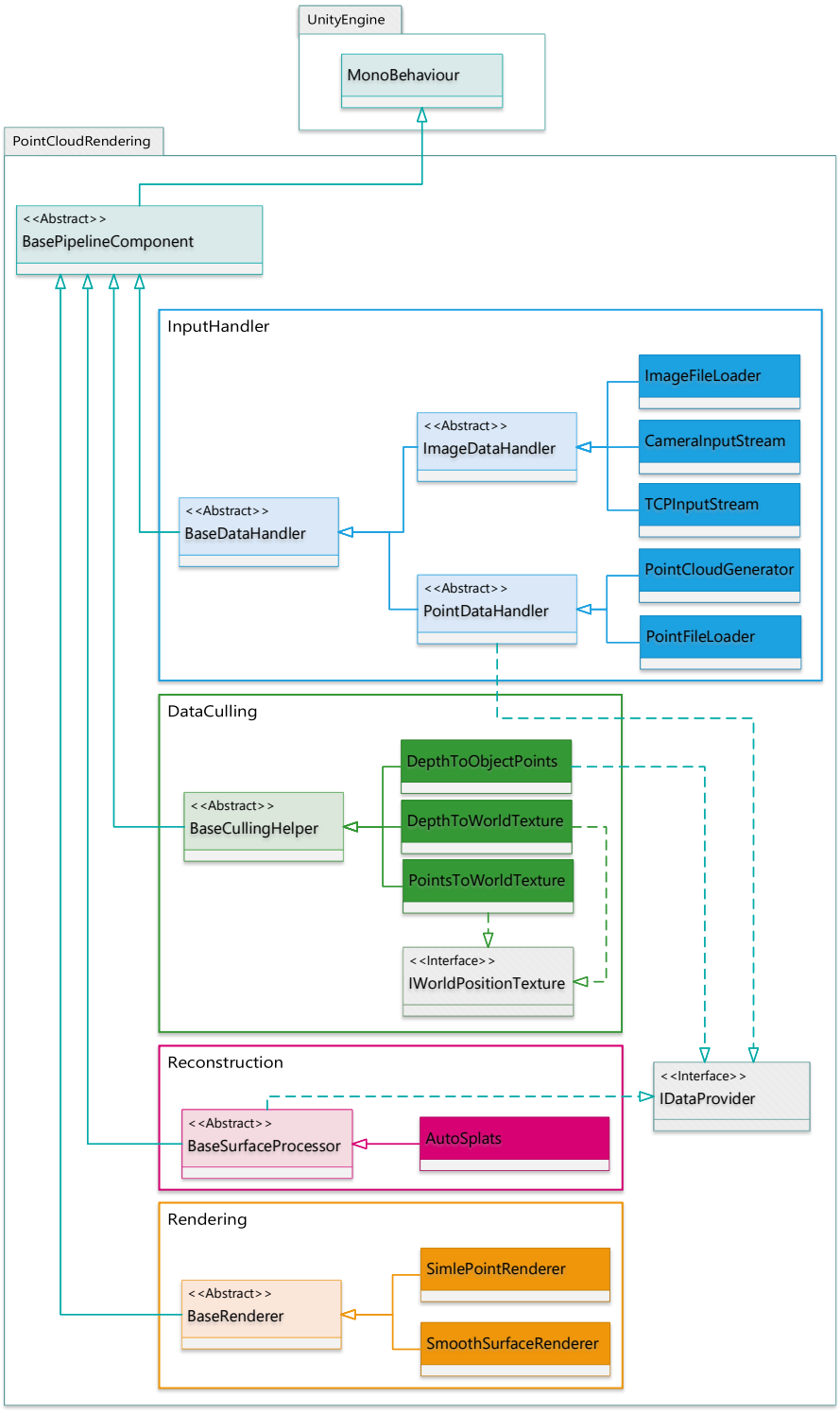


Figure 5.8: UML class diagram of the pipeline inheritance structure.

CHAPTER 6

Evaluation and Results

6.1 Data Generation

The datasets used to evaluate the proposed visualization pipeline are listed in Table 6.1. The datasets are separated by data type: point cloud files and sensor depth maps. The static point cloud scans are provided by the well-known Standford Scanning Repository [Sta]. Each point dataset contains normals, which are used to evaluate the rendering stage (Section 6.2). However, the preprocessed normals are neglected for the evaluation of the surface reconstruction stage and the performance comparisons (Section 6.3). The Teddy dataset was captured by the Technical University of Munich [SEE+12] and is used as a static image dataset. Intrinsic parameters were included in the dataset. The RealSense sensor represents a dynamic data-source, streaming 24-bit color and 16-bit depth data with a resolution of 640×480px at 30 FPS. The depth sensor was calibrated using the standard OpenCV [Bra00] calibration functions, using a checkerboard pattern captured by the infrared camera.

Even though the focus of this work lies on the processing of dynamic input data, we decided also to use static datasets for our evaluation, for the following reasons: Static datasets are often also provided with an reconstructed polygonal mesh with multiple levels of detail [Sta] and are therefore perfectly suited for a direct comparison of the rendering results (Section 6.2). Furthermore, as all of our processing stages (data culling, reconstructing and rendering) work on a per-frame basis, all performance evaluations are directly applicable to dynamic inputs (Section 6.3). This means that the performance measurements carried out on a static point cloud would lead to the exact same result as if the same point cloud was constantly altered, provided that the number of points does not drastically change.

Dataset Name	Data Type	Points	Colors	Normals	Dynamic
Bunny [Sta]	Point	36K	no	yes	no
Dragon [Sta]	Point	438K	no	yes	no
Asian Dragon [Sta]	Point	3.6M	no	yes	no
Statuette [Sta]	Point	5M	no	yes	no
Teddy [SEE+12]	Image	307K	yes	no	no
RealSense Stream	Image	307K	yes	no	yes

Table 6.1: Used datasets for the evaluation process.

6.2 Visual Quality Comparison

This section evaluates the overall rendering quality of the proposed visualization pipeline. The focus lies in the seamless integration of point cloud data in *Unity's* lighting pipeline. This encompasses physically correct shading, dynamic lighting, environmental mapping, real-time shadows and additional post-processing effects.

6.2.1 Mesh to Point Cloud Comparison

Figure 6.1 compares polygon meshes rendered with the default renderer in *Unity*, to the point cloud representation rendered with our custom pipeline. All scenes use the same lighting, environment mapping and material properties to allow for a meaningful comparison. Moreover, all images are rendered with additional post-processing effects, that is anti-aliasing, ambient occlusion and depth of field. Figure 6.1a and 6.1b show a high and low poly mesh, with two point light sources of different color and a directional light source casting the shadow. On the left side, the wireframe representation with additional scene information is shown, whereas on the right side the final rendered result is displayed. Figure 6.1c shows the rendered point cloud using our smooth surface renderer component. The point cloud consists of 36 thousand points, similar to the high poly mesh in figure 6.1a. The quality of our surface rendering component for point clouds is almost indistinguishable from the high poly mesh rendered with *Unity's* default renderer. Only upon closer inspection one can see that some details are getting lost due to the fixed splat radius, leading to slightly smoother normals in the bunny's body (Figure 6.2c) compared to the high-detail rendering using traditional mesh rendering (Figure 6.2a). This uniform radius was manually chosen to fill holes in sparse spots, while highly detailed point clusters are over-smoothed. The difference is visible in the specular highlights, created from the red and white illumination. A more prominent artifact is created on the point clouds silhouette. Hard edges lead to visible splat disks peering over the edge. This can be best observed on the inside of the bunny's ear (Figure 6.2c). The shadow on the ground plane as well as self-shadowing is rendered correctly. A negligible difference remarks the slightly larger point cloud shadow. This is caused by the simplified shadow rendering pass with light-aligned point billboards, meaning that silhouettes are expanded by the shadow disk radius.

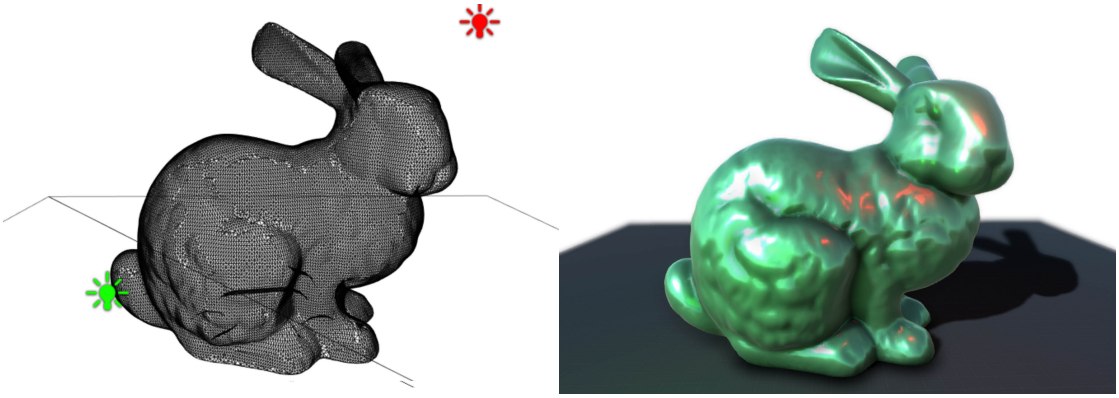
The visual quality of the same point cloud, this time processed with our real-time surface reconstruction component, is displayed in figure 6.1d. Similar to the splat rendering, the normal reconstruction stage uses uniform point radii for the whole point cloud. This accelerates the normal estimation process on the cost of preserving fewer details. As the reconstruction stage was introduced for high performance normal estimation for dynamic point clouds, a comparison to the low poly mesh in figure 6.1b is more suitable. The normals are computed correctly resulting in correct shading, especially notable from the specular highlights and environmental reflections visible in Figure 6.2d. Additionally to the artifacts caused by the rendering stage, we can see that some silhouette points produce erroneous normals (Visible as a black contour on the outside of the bunny's ears in Figure 6.2d). This was already stated as an unsolved challenge in the original publication of the reconstruction algorithm [PJW12].

6.2.2 Dynamic Point Cloud Reconstruction

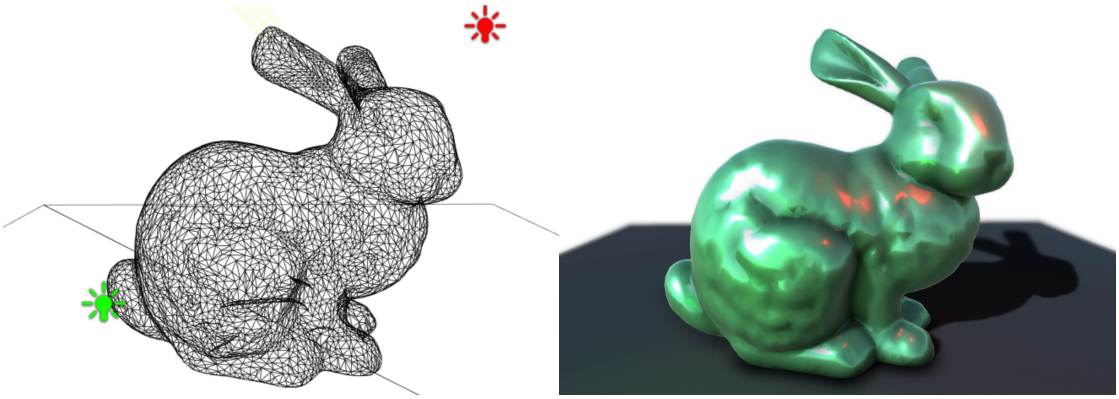
Reconstructing live sensor output in real-time is challenging. The depth maps needs to be processed with 30 FPS and the projected point clouds continuously change, due to natural movements or unpredictable sensor noise. This section presents the results of our visualization pipeline using a single RGB-D camera to capture live footage of a person's head.

Depth sensing devices provide detailed scans of the captured scenes. The acquired depth images are, depending on various conditions, often very noisy. This noise causes incorrectly projected points, leading to temporally inconsistent artifacts in the surface estimation process. These artifacts present themselves as randomly flickering bumps all over the reconstructed surface. Figure 6.3a shows an example of a noisy sensor image reconstruction. Such artifacts are particularly troubling on actually flat surfaces. Applying 2D image processing filters can mitigate this issue and provide smooth and temporal-consistent surfaces. Figure 6.3b shows a reconstructed scan filtered using a median filter for outlier reduction and a bilateral filter to smooth the depth values without over-smoothing hard edges. However, filtering of heavily noise stricken depth maps inevitably results in smoothing of important details.

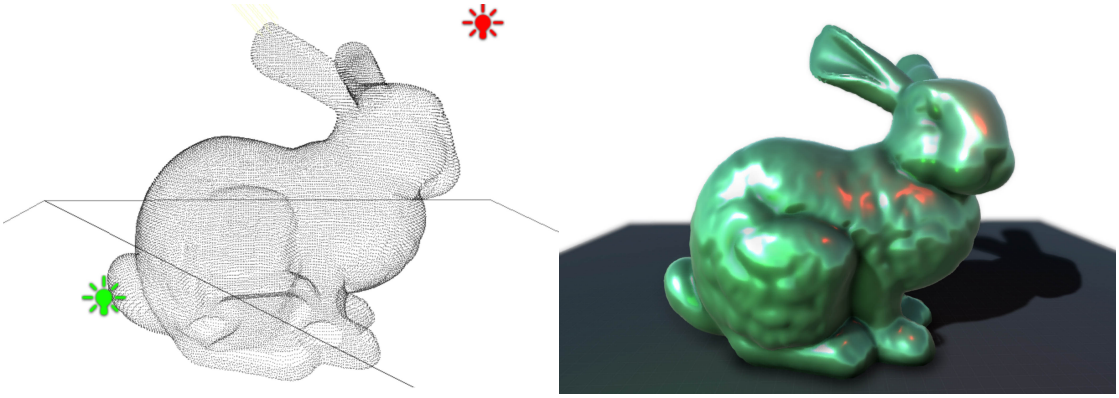
Figure 6.4 shows similar head poses with different virtual lighting conditions, displaying realistic shading and specular reflections. Figure 6.4a shows the reconstructed head rendered with a diffuse material. In the left rendering, the virtual light matches the lighting condition of the real world light of the captured scene. The right side in Figure 6.4a shows a relighting of the scene, with the light coming from the opposite side than in the real capturing. The surface is shaded properly and creates the impression of a real light source shining on the face from the top-right. Figure 6.4b shows the same scene with a glossy material, enhancing the original lighting on the left side while correctly relighting the head with a virtual light on the right side. The results can be further improved by using uniform diffuse lighting to illuminate the scanned person, this reduces specular highlights and shadows which are otherwise captured in the color image. An additional processing step could be introduced, to reduce the illumination on the



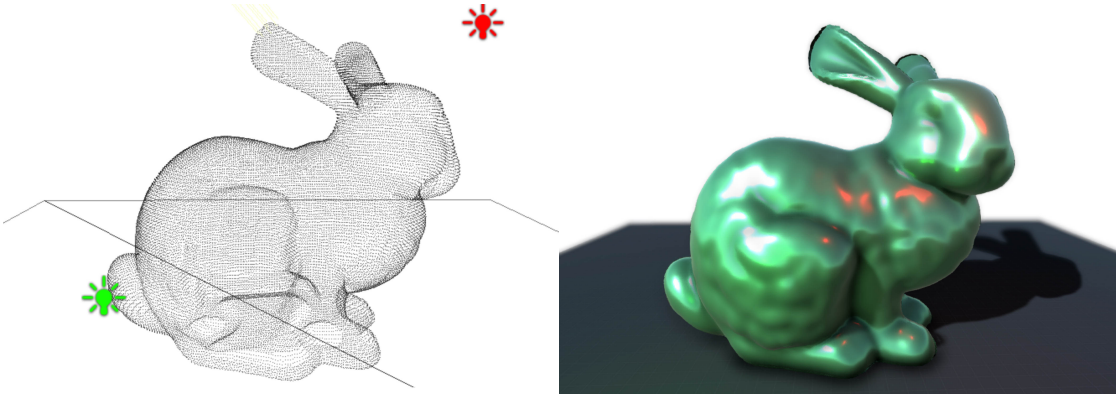
(a) High poly mesh, 36K vertices and 70K faces



(b) Low poly mesh, 7K vertices and 14K faces

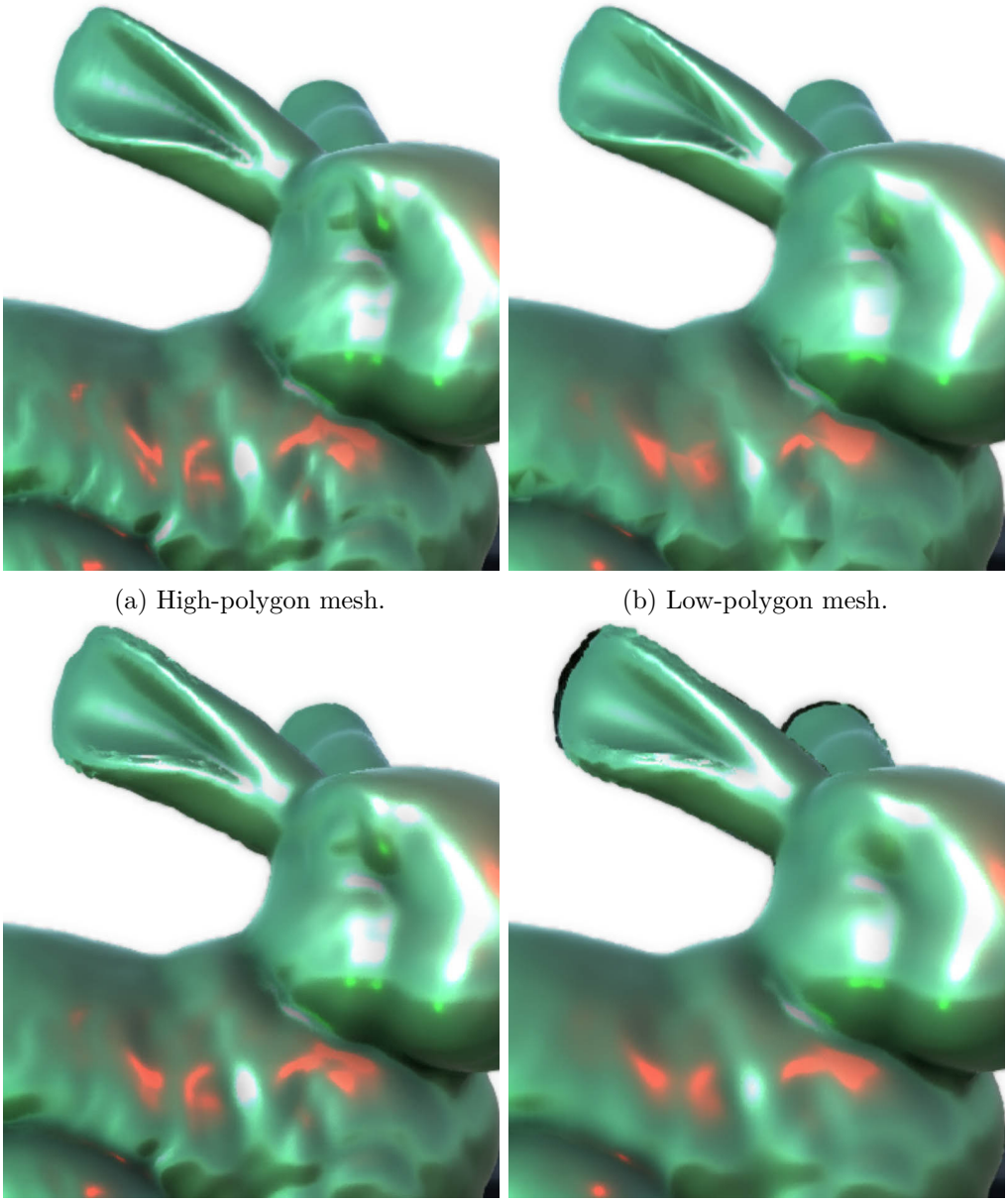


(c) Point cloud with per-point surface normals, 36K surfels



(d) Point cloud with reconstructed surface normals, 36K points

Figure 6.1: Rendering of the triangle mesh representation compared to the point cloud of the Bunny dataset.



(a) High-polygon mesh.

(b) Low-polygon mesh.

(c) Oriented point cloud.

(d) Reconstructed point cloud.

Figure 6.2: Details of the rendered bunnies from Figure 6.1. (a) displays a detail of the high-poly mesh, rendered using the *Unity* default renderer. (b) Is a simplified version of (a) with a fifth of the original polygon count. (c) is rendered with our smooth renderer using per-point normals calculated in an offline pre-processing step. In (d) the normals are reconstructed using our real-time reconstruction stage and our smooth surface renderer.

(a)

(b)

Figure 6.3: Filtering of noisy depth maps. (a) Reconstruction of noisy sensor output. Flickering irregularities on actual smooth surfaces. (b) Reconstruction of filtered sensor output, using median and bilateral filter. Recovering smooth surfaces but loosing sharp details.

(a)

(b)

Figure 6.4: Relighting of reconstructed head. (a) Left: Virtual light source matching real light source direction. (a) Right: Relighted scene with a virtual light source illuminating the face from the right side. (b) Left: Glossy material, with enhanced real light source. (b) Right: Virtual light source, relighting the face with a light source from the right side.

(a)

(b)

Figure 6.5: Different materials on reconstructed live stream data. (a) Glossy material with high specular value. Enhancing specular highlights and creating a glazed effect. (b) Reflective material with no color, environmental reflections and multiple point light sources.

already captured color image [FHD02], for applications where lighting constraints are undesirable.

Additionally to photo-realistic representations, our high-quality rendering component allows the usage of *Unity's* built-in physically based materials. These predefined materials allow to set color, metallic and smoothness properties. Figure 6.5a depicts the recorded head with mapped color texture and material with a high smoothness and metallic factors. The surface obtains a digitally added glazed appearance. Further abstractions can be created by using artificial colors and surreal material properties. Figure 6.5b (left) depicts

how multiple colored and dynamically moving light sources interact with the highly reflective material. For photo-realistic results, environmental reflections are essential, the effects of the environment on a mirror-like material can be seen in Figure 6.5b (right).

6.3 Performance Comparison

Performance is a crucial aspect in real-time rendering applications. This section evaluates the performance of our visualization pipeline on two different platforms. The first platform represents a high-end consumer system, using a desktop PC running Windows 10 with an Intel i7 CPU and a NVIDIA GTX1060 GPU. The second device is a Google Pixel XL Android smartphone with a quad-core processor and an Adreno 530 GPU.

The performance measurements are narrowed down to the actual GPU processing time of the custom processing stages in milliseconds, which are computed every frame. These stages encompass view dependent point culling, normal estimation and smooth surface rendering. Loading times for the specific file formats or stream upload times were neglected. This leaves us with the performance evaluation of the algorithms in the culling, reconstruction and rendering stage. Existing normals on some datasets were ignored to evaluate the per-frame real-time reconstructed stage. All performance evaluations were executed with the same material properties. The neighborhood radius and splat radius were chosen separately to best represent each dataset and render it with perceptibly correct normals and without holes in the surface.

6.3.1 High-end desktop PC

Table 6.2 list the measured GPU times for the high performance system using a NVIDIA GTX1060 with a rendering resolution of 1920×1080px. The table lists the dataset by name with the corresponding number of points, the point count after the culling process and the respective processing times. The visible points are obviously view dependent and change if the camera or the point cloud is moved. This leads to different computation times depending on the viewing angle and the distance of the point cloud to the camera. The measured times were acquired with the point cloud positioned in full view of the camera at a suitable distance to fully fit into the rendered image. The statuette dataset (Figure 6.9) has an elongated form leading to an effectively small projection on the wide-screen view port. Therefore, we rendered the statuette again by aligning its main axis horizontally, allowing to position the statuette closer to the camera resulting in a larger representation. The larger projection of the same dataset is listed as *Statuette Portrait*.

The measurements show that the proposed visualization pipeline performs in real-time, even on very large point clouds with 5 million points. However, the performance of the normal estimation stage and the rendering stage is largely dependent on the chosen radii. This can be seen in Table 6.2. Rendering the Asian Dragon dataset is 1.8ms faster than the Dragon and Statuette dataset, despite having roughly 50% more visible points. This

Dataset	Points	Visible	Cull	Reconstr.	Render	Overall
Bunny	36K	34K	0.12ms	1.80ms	2.00ms	**3.92ms**
Live Stream	307K	145K	1.07ms	4.89ms	1.60ms	**7.56ms**
Dragon	438K	215K	0.32ms	7.40ms	5.80ms	**13.52ms**
Asian Dragon	3610K	364K	2.62ms	13.30ms	4.00ms	**19.92ms**
Statuette	5000K	271K	3.90ms	8.10ms	5.75ms	**17.75ms**
Statuette Portrait	5000K	628K	4.10ms	17.10ms	7.70ms	**28.90ms**

Table 6.2: Performance evaluation of the GPU time of the culling, reconstruction and rendering stage on a NVIDIA GTX1060 at 1920 × 1080px.

results from the actual point distribution of the points in the dataset. The Asian Dragon dataset contains very dense and regular sampled surface points, needing smaller splat radii to render a closed surface, which is more efficient due to reduced splat overdraw. Dense and regular points are also an advantage for the reconstruction stage. However, the large number of visible points requires longer normal estimation computations (13.3ms), leading to an overall longer reconstruction time compared to the Dragon (7.4ms) or Statuette (8.1ms) dataset (Table 6.2).

A notable difference in computation time can be observed between the Statuette and the Statuette Portrait dataset. The culling time stays roughly the same since the number of points in both executions is equal. The largest difference to notice is the pronounced change in the reconstruction processing time regardless of the same neighborhood radius. This again shows how the number of visible points in the reconstruction stage affects the overall performance. This is especially challenging with increasing resolution, as more points are visible in the final rendering.

6.3.2 Mobile Device

The second evaluation was performed on a mobile device. We used a Google Pixel XL smartphone with an Adreno 530 GPU and a 5.5 inch display. The native resolution of this device is 2560 × 1440px with a pixel density of roughly 534 dots per inch (DPI). The mobile application uses the same scene and material settings as the high-end evaluation on the desktop PC. Because of the limited computation power of the mobile GPU, combined with the high native resolution, we solely used the Bunny dataset, with 35K points (Table 6.1), for the mobile evaluation. The point cloud is rendered at different lower resolutions and subsequently upscaled for the output on the device display. *Unity* provides fixed resolution rendering for mobile devices by setting the desired DPI. Table 6.3 shows the resulting measurements on different DPI scaling levels using the Bunny dataset. The 180 DPI pixel density on the 5.5 inch screen results in an effective rendering resolution of 823 × 463px, where we achieve almost real-time rendering capabilities. Higher resolutions, however, increasingly demand longer computation times for the reconstruction and rendering stage, as these stages use multiple full-frame fragment shader passes. Rendering the dataset at native resolution, 2560 × 1440px (534 DPI) is

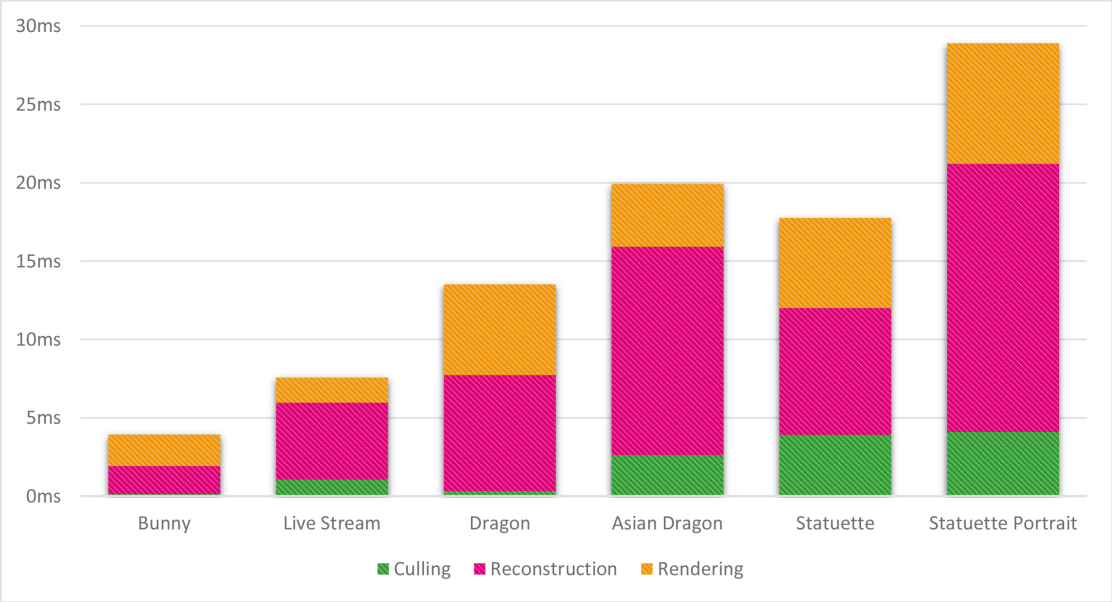


Figure 6.6: Chart of the GPU processing time with color-coded measurements for the specific pipeline stages. The data corresponds to the measurements in Table 6.2. Most notably, one can observe that the Asian Dragon dataset has faster rendering times despite having more visible points than the Dragon or Statuette datasets. This results from the more regular structure of the captured point cloud.

not feasible using our current implementation (as seen in Table 6.3). Nevertheless, even without special optimizations for mobile devices, the evaluation shows that the proposed pipeline is capable to run on multiple platforms and has the means to bring point cloud rendering to next-generation smartphones.

Rendering Resolution	**Visible**	**Cull**	**Reconstr.**	**Render**	**Overall**
823 × 463px (180 DPI)	28K	1.30ms	27.70ms	32.30ms	**61.60ms**
914 × 514px (200 DPI)	29K	1.40ms	32.70ms	37.30ms	**71.40ms**
1143 × 643px (250 DPI)	31K	1.60ms	47.40ms	51.60ms	**100.60ms**
2560 × 1440px (534 DPI)*	35K	2.40ms	195.60ms	112.40ms	**310.40ms**

Table 6.3: Performance evaluation on the Google Pixel XL Android smartphone reconstructing and rendering the Bunny dataset on different rendering resolutions. The result is automatically upscaled to the device's native display resolution at 2560×1440px (534 DPI).
* Native display resolution.

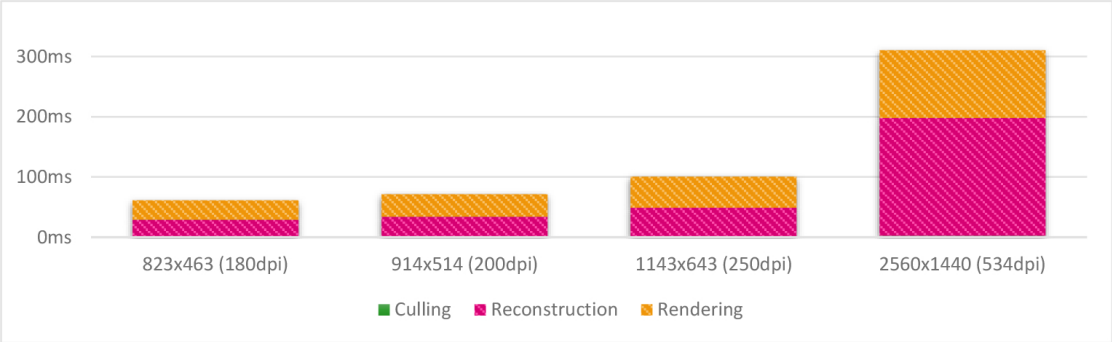


Figure 6.7: Chart of the measured computations times of the visualization pipeline executed on a mobile device. The actual GPU times are listed in Table 6.3

6.4 Applications

In this chapter, we introduce some use cases which benefit from our contribution. The integration of our processing and visualization pipeline in a wide spread multi-platform engine facilitates the handling and integration of point clouds in a variety of applications. This enables the usage of high-quality point cloud renderings in combination with classic geometry representations. As point clouds are no longer visually distinguishable from high-poly triangle meshes, we can exploit their specific advantages. For example, the lack of connectivity between neighboring points in a point cloud simplifies geometry modifications, object merging and morphing, without the need to maintain manifold meshes.

Figure 6.8 depicts two renderings of different, physically based, materials on the Asian Dragon dataset. The final scene is additionally enhanced using temporal anti-aliasing and state-of-the-art post-processing effects, namely SSAO and depth of field. In addition, large point clouds, with more than 5 million points, are rendered with interactive frame-rates (Figure 6.9).

Apart from the high-quality renderings, the modularity of the proposed work-flow allows to substitute all stages with different components, depending on the provided input, the available performance and the expected quality of the visualization. A simplified rendering might be used for previews or data visualizations, where the visual quality is less important than the inherent information stored in the point cloud. A sample implementation is shown in Figure 6.10, where the sensor depth values of the raw Teddy dataset are visualized with a predefined color scheme. Figures 6.11 displays four different representations of the same dataset. All visualizations were achieved with no additional coding, only configuring the components on the point cloud *GameObject* in the *Unity* integrated development environment (IDE). The teddy point cloud was culled using the depth culling planes, in order to remove unnecessary background points. Figure 6.11 shows the raw points with color-coded depth (a) and sensor infrared intensities (b). The result of the surface reconstruction is highly dependent on the chosen neighborhood

radius, a small radius preserves more edges while being more prone to noise (Figure 6.11c), larger radii create a smoother surface but tend to reduce sharp details (Figure 6.11d).

A very promising future prospect is the usage of point clouds in AR and MR applications. Especially since AR goggles like the HoloLens [Hol] or current mobile devices like the iPhone X [iPh] incorporate depth sensors to capture point clouds. We implemented an exemplary Android mobile AR app using the *Unity* engine with AR marker detection and tracking provided by the Vuforia SDK [Vuf]. The application uses a book cover as the tracking marker and places the bunny dataset on top of the book as soon as it is detected (Figure 6.12). The bunny dataset used in the mobile app already provides normals, since the concurrent tracking and point cloud reconstruction are currently not feasible and require further optimizations.

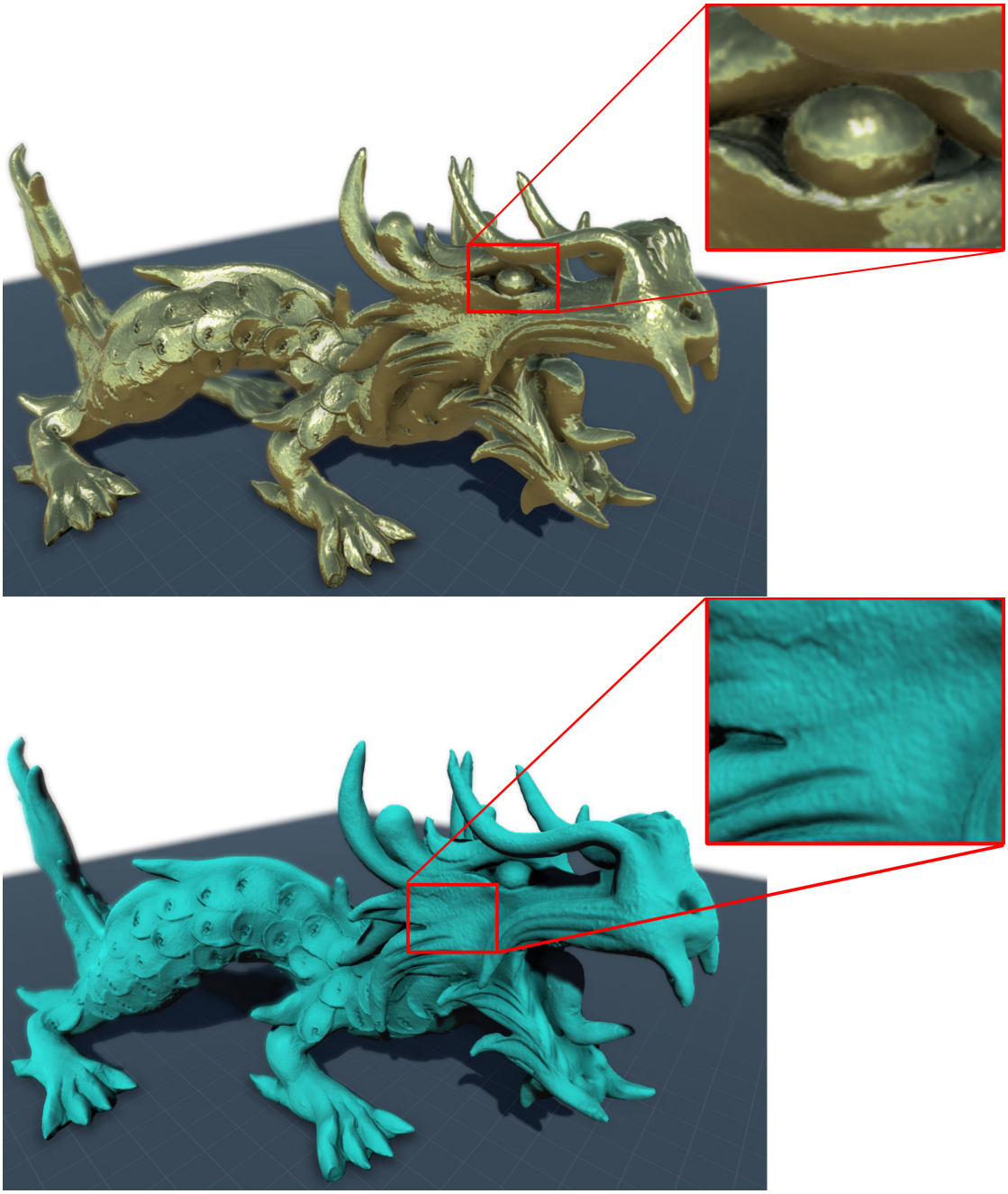


Figure 6.8: Smooth surface renderings of the Asian Dragon dataset with two different, physically based materials. In the rendering on top, the dragon has a metallic, gold-like appearance. The surface has high-detailed environmental reflections, this can be best observed in the dragon's eye. The rendering on the bottom results from the same point cloud. This time, a diffuse turquoise stone-like material is used to display the dragon. The rough surface is nicely represented with high fidelity. This is especially visible on the structured area below the eye. Both renderings were enhanced with temporal anti-aliasing, SSAO and depth of field post-processing effects.

Figure 6.9: The surface of large point clouds can be reconstructed and rendered in real-time, using our proposed smooth surface renderer components for *Unity*. The reconstructed surface of the Statuette dataset, with 5 million points, is rendered in less than 18ms (exact rendering times in Table 6.2)

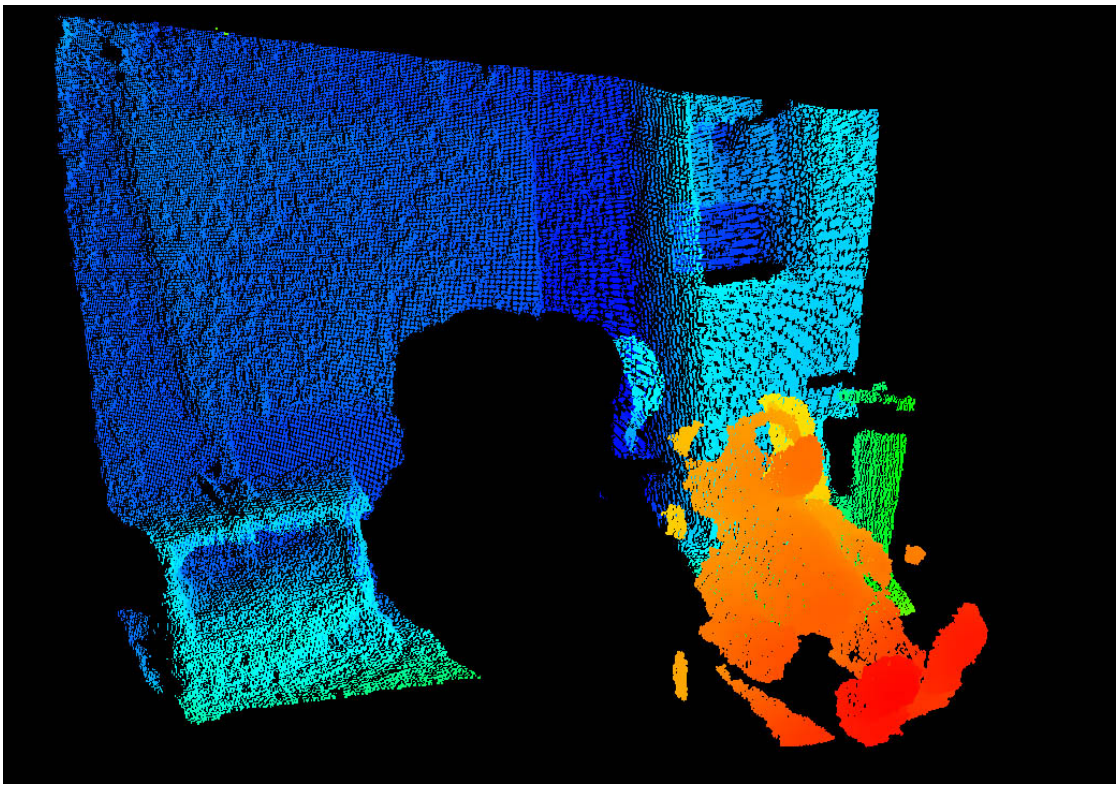
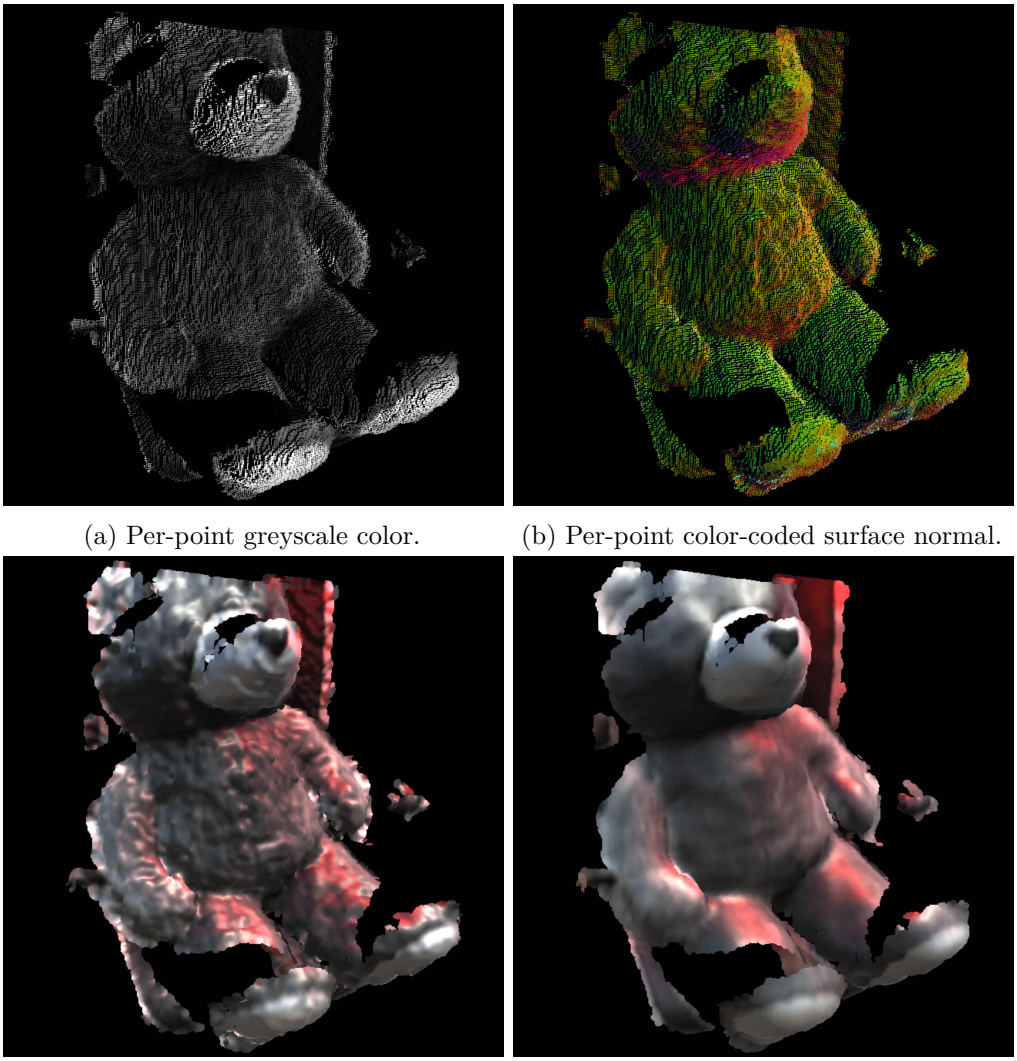


Figure 6.10: The Teddy dataset was captured with a single RGB-D camera. This rendering visualizes a sample application previewing the sensor output directly projected to the 3D space, using a predefined color scheme to encode the depth values retrieved from the sensor. Red points are very close to the sensor, whereas, far away points are colored in blue. This interactive visualization allows emphasizing a variety of properties of the captured scene, in real-time. This type of visualization could be used in robotic and automotive applications, where the real-time preview of the perceived environment can help to gain useful new insights.



(a) Per-point greyscale color.

(b) Per-point color-coded surface normal.

(c) Hight-detail reconstruction.

(d) Smooth recontruction.

Figure 6.11: Visualizing a subset of the Teddy point cloud with different rendering modules and visualization settings. Figure (a) and Figure (b) are rendered using the Simple Point Renderer component (Section 5.6.1). Using the mapped greyscale value (a) and the reconstructed *surfel* normal (b) to color each point. Figure (c) and Figure (d) are using the Smooth Surface Renderer component (Section 5.6.2). With a small (c) and larger (d) kNN-radius for the surface normal estimation (Section 5.5.1). The scene is illuminated with a white directional light from the top and a red point light from the top right.

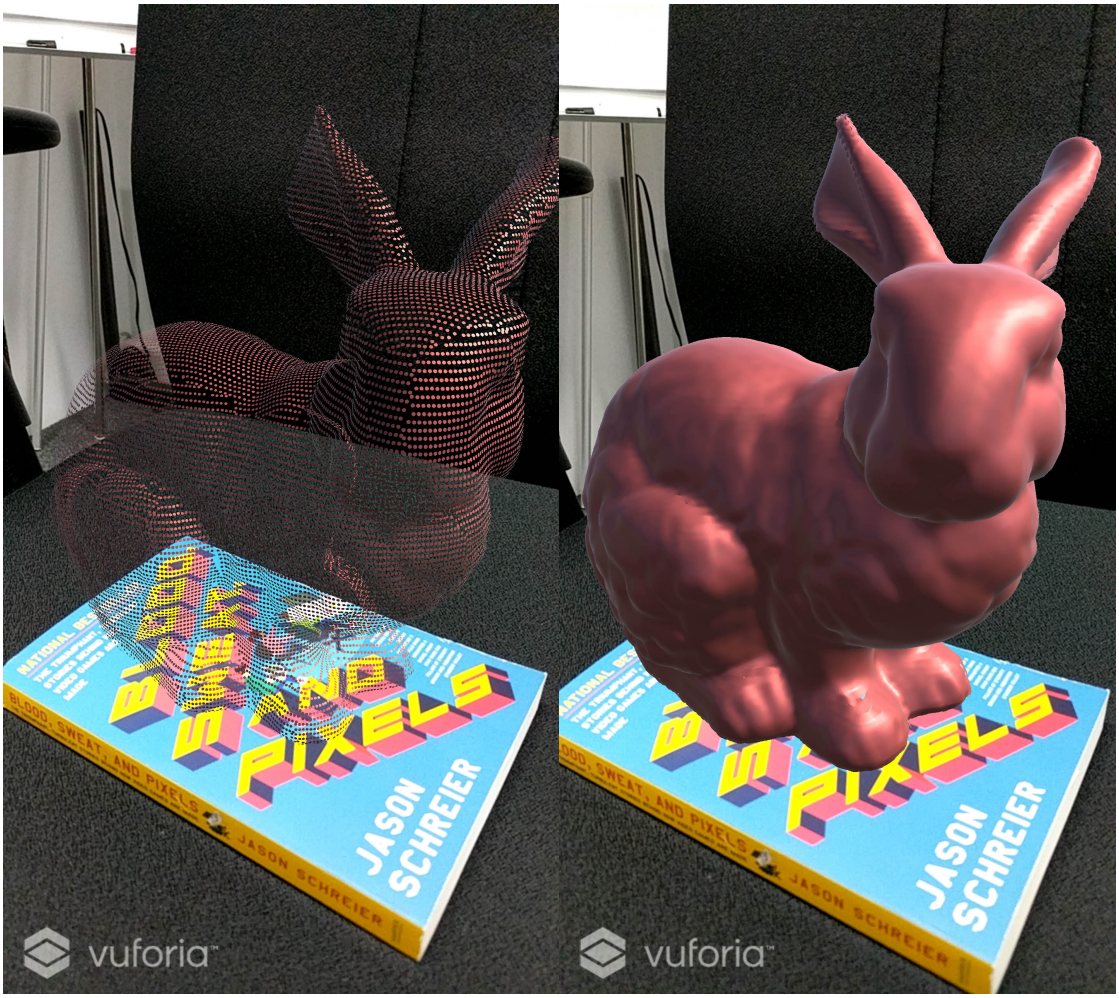


Figure 6.12: The proposed point cloud rendering pipeline in an AR application using the Vuforia SDK. The book cover was used as a tracking marker. The Bunny dataset is rendered with its preprocessed normals. Without the reconstruction stage the tracking and rendering was performed in real-time on the Google Pixel XL with a resolution of 2560×1440px.

CHAPTER 7

Conclusion and Future Work

We presented a visualization pipeline for static as well as dynamic point clouds, capable of reconstructing and rendering the originally sampled surface in real-time. The pipeline was implemented as an extension of the widely used *Unity* game engine, which enables the fast development of a variety of applications for multiple platforms. The configurable pipeline components can be used directly in the *Unity* editor and require no additional coding. The pipeline is separated in four stages, using state-of-the-art point processing methods and algorithms optimized for modern GPUs. The modular implementation of the pipeline as *Unity* components allows substituting single stages to change the behavior and adapt to a wide range of possible applications.

The input handler stage provides a common interface for the heterogeneous input sources and manages the data access for following stages. This allows the pipeline to access point cloud information from various file sources as well as continuous input streams, either from a local device or from a remote network connection. The second stage dynamically reduces the processing overhead by culling hidden or invisible points. Furthermore, depth maps from sensor inputs can be enhanced by applying 2D image processing filters, effectively reducing sensor noise. The surface reconstruction stage implements a normal estimation algorithm based on AutoSplats. The algorithm estimates the surface normal for each visible point in screen space, optimized for parallel execution on modern programmable GPUs. This stage is required if the surface needs to be reconstructed dynamically and offline preprocessing steps are not applicable. The last stage uses the results from the previous stages to render the point cloud. The visual appearance of the point cloud can strongly differ, depending on the use case in the application. This ranges from scientific visualizations and fast sensor previews to photo-realistic renderings with closed surfaces, virtual lighting, environmental reflections and dynamic shadows. High-quality renderings of point clouds are achieved by rendering the points with the GPU accelerated surface splatting technique.

We showed that the pipeline is capable of visualizing photorealistic 3D point clouds, which are nearly indistinguishable from classic polygonal meshes. The point clouds are fully integrated in the lighting and shadow system of the game engine and take full advantage of its platform-independence. Furthermore, we showed that the complete pipeline performs in real-time, even for large data sets with 5 million points. The end-to-end system, featuring live reconstructed, relighted and rendered data from a RGB-D camera represents the main contribution of this work.

In the future, we want to continue our work on the proposed pipeline to further expand the field of application and increase the visual quality as well as the performance. This includes the implementation of pipeline components especially optimized for mobile devices, to facilitate the extensive use of larger point clouds in AR applications. Furthermore, an extension for optimized concurrent handling of multiple point clouds from various input sources in a single scene is desirable. A possible improvement for the high-end visual quality reconstructions is the full implementation of the AutoSplats algorithm with adaptive per-point radii. Further, we are looking into newly added techniques and functionalities in next-generation *Unity* versions. Especially the scriptable rendering pipelines and the multi-threaded job-system look promising to increase the performance of our visualization pipeline.

List of Figures

3.1 Depth capturing process in the RealSense SR300 [Int16] 15

4.1 Flowchart of the visualization pipeline stages and their execution order. . 17
4.2 Flowchart of the *Input Handler Stage* and its input/output interface. The flowchart outlines the internal workflow of the stage, from the provided input, the main processing steps and the final output. The vertical lanes differentiate between the two fundamentally different data types, the point based data and the raw sensor image data. The input for the *Input Handler Stage* is heterogeneous and requires customized implementations for each source, while the output is reduced to the two main types, point arrays and sensor textures. 19
4.3 Flowchart of the *Data Culling Stage* and its input/output interface. The flowchart outlines the internal workflow using the provided input resulting in two distinct outputs. The two lanes separates our two main data types, point and sensor image data. Note that the texture input and output are defined in different spaces. The image input lane expects raw depth values in sensor space while the image output consists of screen space attribute textures (Similar to a G-Buffer [AMHH08]). Furthermore, this stage allows to transform the data type to adapt the output for subsequent stages. . . 20
4.4 Flowchart of the *Surface Reconstruction Stage* and its input/output interface. The flowchart outlines the internal workflow, the required input and the expected output. Vertical lanes separate the point data from the screen space attribute textures. The internal processing for this stage was depicted using the kNN plane fitting approach [PJW12, Sch16], but could vary depending on the implemented algorithm. 22
4.5 Flowchart of the *Rendering Stage* and its input/output interface. The flowchart outlines the internal workflow of the stage, from the provided input, the main processing steps and the final output. Contrary to the previous flowcharts (Figures 4.2, 4.3, 4.4), the lanes in this figure do not separate the input by type (points or images) but instead by the information they contain. Specifically *surfel* input, containing positions and surface normals, and input containing plain point positions. Note that the additional surface information can always be omitted or used for simplified visualization paths. 24

5.1 Flowchart of the implemented visualization pipeline stages, their execution order and possible combinations. Each connected set of modules (from left to right) represents a valid pipeline setup. 26

5.2 Live streamed data from a depth sensor and color camera. The streams are handled in a native plugin and directly uploaded to the GPU. The visualized frustum represents the view of the sensor in a virtual 3D coordinate system. The yellow area highlights the region of interest, points outside this frustum are discarded. The surface of the persons head is reconstructed and rendered in real-time. The sun icon shows the position of the virtual light source, used to recalculate the lighting of the captured scene. 27

5.3 View frustum (grey) of a depth sensor. The yellow area marks the area of interest while everything else is culled. The color used to render this simplified visualization encodes the depth value, from green on the near culling plane to blue on the far culling plane. 30

5.4 Unprocessed raw depth map (left) and the same depth map after distance culling and filtering (right). 31

5.5 Different visualization paths depending on the provided or processed surface information. Starting from the raw point cloud, either present as point positions or depth values defined in sensor space, we can visualize the point cloud as it is, with additional color-coded information, or as an alternative we use the existing or estimated normals to reconstruct a closed surface with photo-realistic shading. 37

5.6 Surface splats made visible by decreasing the splat radius. Point attributes like color and *surfle* orientation are propagated across the splat and blended with overlapping neighboring points. 38

5.7 Dynamic real-time shadow casting and receiving on the dragon dataset with reconstructed normals. The scene shows the combination and interaction between point cloud objects and classic triangles mesh based geometry. The dragon point cloud is placed on a marble pedestal inside a wooden cage. The gold-like material has a polished appearance with characteristic environmental reflections. The cage casts shadows onto the pedestal as well as the point cloud, at the same time is the dragon casting shadows onto the environment and itself. Figure (a) depicts the scene illuminated with a virtual light source projecting the dragon silhouette onto the cage, Figure (b) and (c) show the same scene with light coming from different directions. Additional post-processing effects, like SSAO and depth of field, improve the final result considerably. 40

5.8 UML class diagram of the pipeline inheritance structure. 44

6.1 Rendering of the triangle mesh representation compared to the point cloud of the Bunny dataset. 48

6.2 Details of the rendered bunnies from Figure 6.1. (a) displays a detail of the high-poly mesh, rendered using the *Unity* default renderer. (b) Is a simplified version of (a) with a fifth of the original polygon count. (c) is rendered with our smooth renderer using per-point normals calculated in an offline pre-processing step. In (d) the normals are reconstructed using our real-time reconstruction stage and our smooth surface renderer. 49

6.3 Filtering of noisy depth maps. (a) Reconstruction of noisy sensor output. Flickering irregularities on actual smooth surfaces. (b) Reconstruction of filtered sensor output, using median and bilateral filter. Recovering smooth surfaces but loosing sharp details. 50

6.4 Relighting of reconstructed head. (a) Left: Virtual light source matching real light source direction. (a) Right: Relighted scene with a virtual light source illuminating the face from the right side. (b) Left: Glossy material, with enhanced real light source. (b) Right: Virtual light source, relighting the face with a light source from the right side. 51

6.5 Different materials on reconstructed live stream data. (a) Glossy material with high specular value. Enhancing specular highlights and creating a glazed effect. (b) Reflective material with no color, environmental reflections and multiple point light sources. 52

6.6 Chart of the GPU processing time with color-coded measurements for the specific pipeline stages. The data corresponds to the measurements in Table 6.2. Most notably, one can observe that the Asian Dragon dataset has faster rendering times despite having more visible points than the Dragon or Statuette datasets. This results from the more regular structure of the captured point cloud. 55

6.7 Chart of the measured computations times of the visualization pipeline executed on a mobile device. The actual GPU times are listed in Table 6.3 . 56

6.8 Smooth surface renderings of the Asian Dragon dataset with two different, physically based materials. In the rendering on top, the dragon has a metallic, gold-like appearance. The surface has high-detailed environmental reflections, this can be best observed in the dragon's eye. The rendering on the bottom results from the same point cloud. This time, a diffuse turquoise stone-like material is used to display the dragon. The rough surface is nicely represented with high fidelity. This is especially visible on the structured area below the eye. Both renderings were enhanced with temporal anti-aliasing, SSAO and depth of field post-processing effects. 58

6.9 The surface of large point clouds can be reconstructed and rendered in real-time, using our proposed smooth surface renderer components for *Unity*. The reconstructed surface of the Statuette dataset, with 5 million points, is rendered in less than 18ms (exact rendering times in Table 6.2) 59

6.10 The Teddy dataset was captured with a single RGB-D camera. This rendering visualizes a sample application previewing the sensor output directly projected to the 3D space, using a predefined color scheme to encode the depth values retrieved from the sensor. Red points are very close to the sensor, whereas, far away points are colored in blue. This interactive visualization allows emphasizing a variety of properties of the captured scene, in real-time. This type of visualization could be used in robotic and automotive applications, where the real-time preview of the perceived environment can help to gain useful new insights. 60
6.11 Visualizing a subset of the Teddy point cloud with different rendering modules and visualization settings. Figure (a) and Figure (b) are rendered using the Simple Point Renderer component (Section 5.6.1). Using the mapped greyscale value (a) and the reconstructed *surfel* normal (b) to color each point. Figure (c) and Figure (d) are using the Smooth Surface Renderer component (Section 5.6.2). With a small (c) and larger (d) kNN-radius for the surface normal estimation (Section 5.5.1). The scene is illuminated with a white directional light from the top and a red point light from the top right. . . . 61
6.12 The proposed point cloud rendering pipeline in an AR application using the Vuforia SDK. The book cover was used as a tracking marker. The Bunny dataset is rendered with its preprocessed normals. Without the reconstruction stage the tracking and rendering was performed in real-time on the Google Pixel XL with a resolution of 2560×1440px. 62

List of Tables

5.1 Different data provider implementations. The data type defines how the data needs to be rendered, and can be used to check if the renderer component is compatible with the specified type. Not all components are inherently able to provide surface normals, e.g. *DepthToObjectPoints* solely process depth maps to compute 3D points. Surface normals can then only be gained with further processing steps, e.g. with subsequent stages extending the *BaseSurfaceProcessor* component. The priority of a data provider tells the rendering component which component is the most relevant, if multiple data providers are present. 41

6.1 Used datasets for the evaluation process. 46
6.2 Performance evaluation of the GPU time of the culling, reconstruction and rendering stage on a NVIDIA GTX1060 at 1920×1080px. 54
6.3 Performance evaluation on the Google Pixel XL Android smartphone reconstructing and rendering the Bunny dataset on different rendering resolutions. The result is automatically upscaled to the device's native display resolution at 2560×1440px (534 DPI). 55

Acronyms

API application programming interface.

AR augmented reality.

CPU central processing unit.

DPI dots per inch.

FPS frames per second.

GLSL OpenGL shading language.

GPGPU general-purpose computation on GPU.

GPU graphics processing unit.

HD high definition.

HLSL high level shading language.

HMD head mounted display.

IDE integrated development environment.

kNN k nearest neighbors.

MR mixed reality.

PCL point cloud library.

RGB red green blue.

RGB-D RGB and depth.

SDK software development kit.

SSAO screen space ambient occlusion.

SVD singular value decomposition.

VGA video graphics array.

VR virtual reality.

XR Extended reality, general term describing VR, AR and MR.

Bibliography

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008.

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

[Bou09] Christian Boucheny. *Visualisation Scientifique de Grands Volumes de Données: Pour une Approche Perceptive*. PhD thesis, Université Joseph-Fourier-Grenoble I, 2009.

[Bou18] Paul Bourke. Automatic 3D reconstruction: An exploration of the state of the art. *GSTF Journal on Computing (JoC)*, 2(3), 2018.

[Bra00] Gary Bradski. The opencv library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 25(11):120–123, 2000.

[BRS+16] Daniele Bonatto, Ségolène Rogge, Arnaud Schenkel, Rudy Ercek, and Gauthier Lafruit. Explorations for real-time point cloud rendering of natural scenes in virtual reality. In *Proceedings of International Conference on 3D Imaging (IC3D)*, pages 1–7. IEEE, 2016.

[BSK04] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In *Proceedings of 1st Conference on Point-Based Graphics*, pages 25–32. Eurographics, 2004.

[CBHH17] Dimitris Chatzopoulos, Carlos Bermejo, Zhanpeng Huang, and Pan Hui. Mobile augmented reality survey: From where we are to where we go. *IEEE Access*, 5:6917–6950, 2017.

[Dira] Microsoft® Developer Network, Compute Shader Overview. `msdn.microsoft.com/library/ff476331.aspx`. Accessed: 2018-08-26.

[Dirb] Microsoft® DirectX® API. `msdn.microsoft.com/library/windows/desktop/ee663274.aspx`. Accessed: 2018-08-26.

[DKD+16] Mingsong Dou, Sameh Khamis, Yury Degtyarev, Philip Davidson, Sean Ryan Fanello, Adarsh Kowdle, Sergio Orts Escolano, Christoph Rhemann, David Kim, and Jonathan Taylor. Fusion4d: Real-time performance capture of challenging scenes. *ACM Transactions on Graphics (TOG)*, 35(4):114, 2016.

[DRL10] Petar Dobrev, Paul Rosenthal, and Lars Linsen. An image-space approach to interactive point cloud rendering including shadows and transparency. *Computer Graphics and Geometry*, 12(3):2–25, 2010.

[Emg] EmguCV. `www.emgu.com`. Accessed: 2018-08-26.

[eSEO12] Renan Machado e Silva, Claudio Esperança, and Antonio Oliveira. Efficient hpr-based rendering of point clouds. In *Proceedings of the 25th Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 126–133. IEEE, 2012.

[FHD02] Graham D. Finlayson, Steven D. Hordley, and Mark S. Drew. Removing shadows from images. In *Proceedings of the 7th European Conference on Computer Vision*, pages 823–836. ECCV, Springer Berlin Heidelberg, 2002.

[GTKK13] Marcin Grzegorzek, Christian Theobalt, Reinhard Koch, and Andreas Kolb. *Time-of-Flight and Depth Imaging. Sensors, Algorithms and Applications: Dagstuhl Seminar 2012 and GCPR Workshop on Imaging New Modalities*, volume 8200. Springer, 2013.

[Hof89] Christoph M. Hoffmann. *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann Publishers Inc., 1989.

[Hol] Microsoft® HoloLens. `www.microsoft.com/hololens`. Accessed: 2018-08-26.

[Int16] Intel®. *RealSense™Camera SR300 Product Datasheet*, 6 2016. Revision 1.

[iPh] Apple® iPhone X. `www.apple.com/iphone-x`. Accessed: 2018-08-26.

[Kaj09] Vladimir Kajalin. Screen space ambient occlusion. *Shader X*, 7(413):24, 2009.

[KBR+12] Julius Kammerl, Nico Blodow, Radu Bogdan Rusu, Suat Gedikli, Michael Beetz, and Eckehard Steinbach. Real-time compression of point cloud streams. In *Proceedings of International Conference on Robotics and Automation (ICRA)*, pages 778–785. IEEE, 2012.

[KTB07] Sagi Katz, Ayellet Tal, and Ronen Basri. Direct visibility of point sets. *ACM Transactions on Graphics (TOG)*, 26:24, 2007.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 163–169. ACM, 1987.

[Lib] Intel® RealSense™SDK on GitHub. www.github.com/IntelRealSense/librealsense. Accessed: 2018-08-26.

[LZZ13] Charles Loop, Cha Zhang, and Zhengyou Zhang. Real-time high-resolution sparse voxelization with application to image-based modeling. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 73–79. ACM, 2013.

[McG08] Morgan McGuire. A fast, small-radius GPU median filter. *Shader X*, February 2008.

[Met] Apple® Metal® API. developer.apple.com/metal. Accessed: 2018-08-26.

[Mon] Mono Project. www.mono-project.com. Accessed: 2018-08-26.

[NET] Microsoft® .NET. www.microsoft.com/net. Accessed: 2018-08-26.

[NFS15] Richard A Newcombe, Dieter Fox, and Steven M Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *Proceedings of Conference on Computer Vision and Pattern Recognition*, pages 343–352. IEEE, 2015.

[OERF+16] Sergio Orts-Escolano, Christoph Rhemann, Sean Fanello, Wayne Chang, Adarsh Kowdle, Yury Degtyarev, David Kim, Philip L Davidson, Sameh Khamis, and Mingsong Dou. Holoportation: Virtual 3D teleportation in real-time. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 741–754. ACM, 2016.

[Ope] OpenGL™ API. www.opengl.org. Accessed: 2018-08-26.

[Paj03] Renato Pajarola. Efficient level-of-details for point based rendering. In *Proceedings of Computer Graphics and Imaging*, pages 141–146, 2003.

[PJW12] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. Auto splats: Dynamic point cloud visualization on the GPU. In *Proceedings of EGPGV Symposium on Parallel Graphics and Visualization*, pages 139–148. Eurographics, 2012.

[PZVBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 335–342. ACM Press/Addison-Wesley Publishing Co., 2000.

[RC11] R. B. Rusu and S. Cousins. 3D is here: Point cloud library (PCL). In *Proceedings of International Conference on Robotics and Automation*, pages 1–4. IEEE, 2011.

[REH06] Fabio Remondino and Sabry El-Hakim. Image-based 3D modelling: a review. *The photogrammetric record*, 21(115):269–291, 2006.

[Ric] Riccio Christophe Blog: How bad are small triangles on GPU and why? `www.g-truc.net/post-0662.html`. Accessed: 2018-08-26.

[Sch16] Dominik Schörkhuber. Fast kNN in screenspace on GPGPU. Bachelor's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2016.

[SEE+12] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of RGB-D SLAM systems. In *Proceedings of International Conference on Intelligent Robot Systems (IROS)*, pages 573–580, 2012.

[Sta] The Stanford 3D Scanning Repository. `graphics.stanford.edu/data/3Dscanrep`. Accessed: 2018-08-26.

[Str] Structure Sensor - 3D scanning, augmented reality, and more for mobile devices. `www.structure.io`. Accessed: 2018-08-26.

[TADB+16] Donny Tytgat, Maarten Aerts, Jeroen De Busser, Sammy Lievens, Patrice Rondao Alface, and Jean-Francois Macq. A real-time 3D end-to-end augmented reality system (and its representation transformations). In *Proceedings of Applications of Digital Image Processing XXXIX*, volume 9971. International Society for Optics and Photonics, 2016.

[TCF15] Dorina Thanou, Philip A Chou, and Pascal Frossard. Graph-based motion estimation and compensation for dynamic 3D point cloud compression. In *Proceedings of International Conference on Image Processing (ICIP)*, pages 3235–3239. IEEE, 2015.

[TM98] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Proceedings of Sixth International Conference on Computer Vision*, pages 839–846. IEEE, 1998.

[Uni] Unity 3D. `www.unity3d.com`. Accessed: 2018-08-26.

[UPR] Unity 3D Fast Facts. `www.unity3d.com/public-relations`. Accessed: 2018-05-12.

[VM02] G. Varadhan and D. Manocha. Out-of-core rendering of massive geometric environments. In *Proceedings of Conference on Visualization*, pages 69–76. IEEE, Nov 2002.

[Vuf] Vuforia Augmented Reality SDK. `www.vuforia.com`. Accessed: 2018-08-26.

[Vul] Khronos™Group Vulkan® API. www.khronos.org/vulkan. Accessed: 2018-08-26.

[YGX+17] Tao Yu, Kaiwen Guo, Feng Xu, Yuan Dong, Zhaoqi Su, Jianhui Zhao, Jianguo Li, Qionghai Dai, and Yebin Liu. Bodyfusion: Real-time capture of human motion and surface geometry using a single depth camera. In *Proceedings of International Conference on Computer Vision (ICCV)*, pages 910–919. IEEE, 2017.

[ZPVBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Surface splatting. In *Proceedings of 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 371–378. ACM, 2001.