FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jaroslav Vozár

Surfel-cloud rendering

Department of Software and Computer Science Education

Supervisor of the master thesis: RNDr. Josef Pelikán

Study programme: Computer Science

Specialization: IPGVPH

Prague 2024

I would like to thank my thesis supervisor for helping me with this thesis and for general guidance during both the research and implementation process.

Title: Surfel-cloud rendering

Author: Jaroslav Vozár

Department / Institute: Department of Software and Computer Science Education

Supervisor of the master thesis: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract:

Rendering optimization is very important, especially for low-power and high-demand devices such as AR/VR headsets, where a scene has to be rendered for each eye at a high frame rate and with low latency. Many algorithms have been created for this problem, each with its own advantages and disadvantages.

This project firstly compared existing rendering optimization techniques, mostly focusing on the Level of Detail approach. Then, based on specific needs, the best suited algorithm was selected and implemented. The implementation process also involved several improvements and adjustments.

Keywords: surfels, Level of Detail (LoD), rendering, 3D model sampling

---

Názov práce: Surfel-cloud rendering

Autor: Jaroslav Vozár

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedúci diplomovej práce: RNDr. Josef Pelikán, Katedra softwaru a výuky informatiky

Abstrakt:

Optimalizácia renderovania je veľmi dôležitá, najmä pre zariadenia s nízkou spotrebou a vysokými nárokmi, ako sú AR/VR headsety, kde sa scéna musí vykresliť pre každé oko pri vysokej snímkovej frekvencii a s nízkou latenciou. Pre tento problém bolo vytvorených mnoho algoritmov, z ktorých každý má svoje výhody a nevýhody.

Cieľom tohto projektu bolo najprv porovnať existujúce techniky optimalizácie renderovania, zamerané predovšetkým na Level of Detail prístup. Potom na základe špecifických potrieb bol vybraný a implementovaný najvhodnejší algoritmus. Proces implementácie zahŕňal aj viaceré vylepšenia a úpravy.

Kľúčové slová: surfels, Level of Detail (LoD), rendering, 3D model sampling

Contents

1 Introduction

This chapter explains the **main goals** of the Master thesis, the problems it tries to solve, and the motivation behind it all.

1.1 Background

In computer graphics industries such as data visualization or 3D modeling, **Level of Detail (LoD)** is a technique used to **optimize rendering** time and computational and memory requirements. The main idea is that objects positioned further away from the observer (/virtual camera) can be represented in **lower detail** than objects closer to it. This is simply because the further away the object is from the camera, the smaller its projection on the screen. This does not apply to all projections (such as orthographic), but it does apply to perspective projection, which is the most common in computer graphics applications.

1.2 Motivation for the thesis

LoD can be achieved through numerous approaches, each with advantages and disadvantages. These must be thoroughly analyzed for the specific use case, requirements, and constraints.

The end goal of this thesis is both the **selection and implementation** of rendering optimization technique suitable for, but not only, FataMorgana – an AR/VR (augmented/virtual reality) platform for remote collaboration developed by a smaller start-up company Pocket Virtuality.

1.3 Problem statement

AR and VR headsets are notorious for requiring more computational power than traditional rendering to standard screen for several reasons:

1. There is a need to render the same content to both eyes but from slightly different positions (**stereo rendering**).
2. While users of standard computer applications (such as video games, 3D modeling software, …) can be satisfied with rendering 25-30 frames per second (FPS), a wearer of a VR headset needs at least **60 FPS** to avoid motion sickness.

3. **Latency** also has to be considered. Motion sickness can occur if it takes longer than **20-25ms** from head movement to the new frame being displayed in the headset (Hou, et al., 2004) (Wilson, 2016).
4. Displays in VR headsets are placed mere centimeters away from the eyes. Therefore, they should provide **higher resolution** to keep visible pixels per degree as high as possible and make viewed images sharper.

Reasons **2**, **3**, and **4** also applies to AR headsets but are **not so strict** due to the constantly visible real environment. However, most AR headsets (such as Microsoft HoloLens) are portable, which means they cannot utilize the powerful hardware of desktop PCs (like connected VR headsets can). They are also expected to last at least a few hours on an embedded battery and be lightweight and sleek. All this results in most AR headsets having **performance comparable to modern smartphones**.

Therefore, using LoD is especially beneficial for AR/VR applications.

1.4 Overview of the FataMorgana platform

FataMorgana is an enterprise software system developed by Pocket Virtuality. Its purpose is to provide an easy way for **training and remote assistance**, mainly for industry (manufacturing, energy, maintenance, …).

There are numerous separate parts in this system. The best way to describe them is through a **typical workflow**. It starts with getting all the required 3D models of the scene, tools, etc. If a 3D model is unavailable, the scene can be scanned using a 3D scanner or Microsoft HoloLens AR headset. A server-side app, **FMBrain**, merges scene data from various sources. Then, a desktop program called FMStudio can be used to create scenarios for training, adjust models, see people using HoloLens in real-time in the correct place relative to the model, communicate with them, etc.

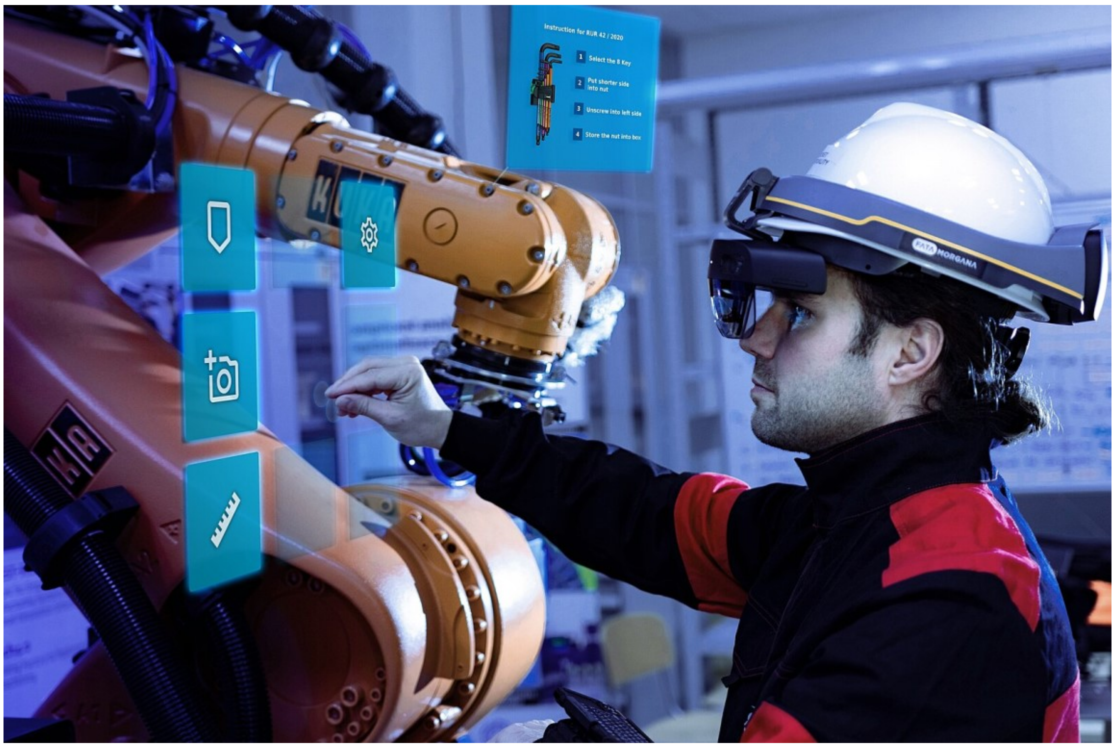


Figure 1 FMVoyager displaying additional information including maintenance steps, over the real environment. Courtesy of Pocket Virtuality

A client app on HoloLens called **FMVoyager** allows communication back to FMStudio and other FMVoyager users. Besides that, it shows steps in case of training, markers in the real world with text/image info, avatars of other HoloLens users (in case they are connected remotely), and more (see Figure 1). FMVoyager runs on Microsoft HoloLens and various VR devices.

This whole process, from the scanning of the environment to displaying the result in AR/VR, is depicted in Figure 2.

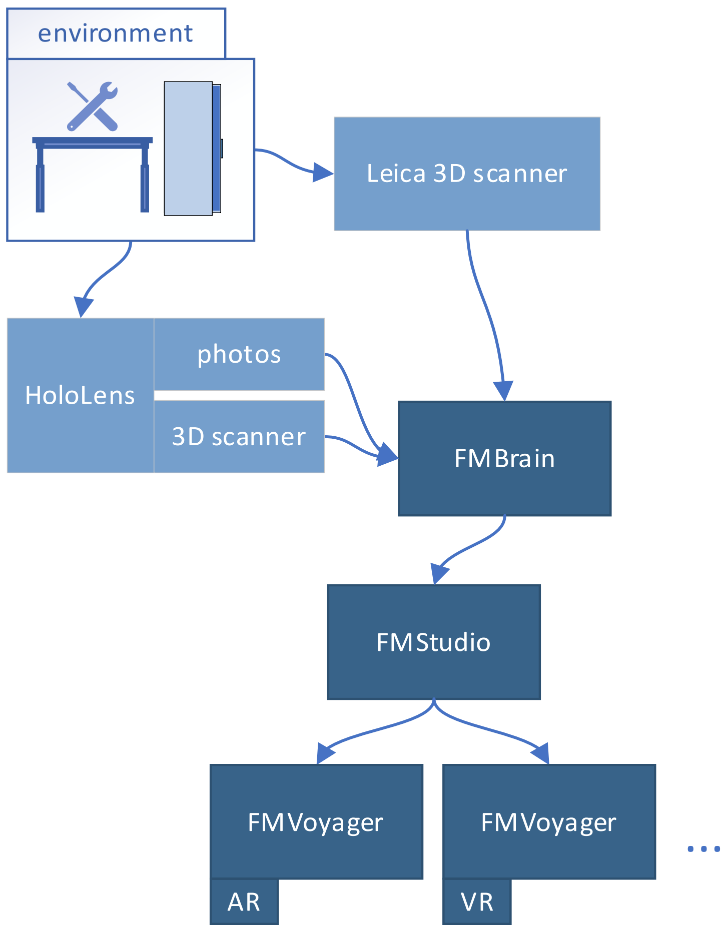


Figure 2 A simplified overview of a typical workflow in the FataMorgana platform. It starts with HoloLens, Leica, and other devices scanning the environment, which is then fused in FMBrain, adjusted and utilized in FMStudio, before being displayed in AR/VR headsets using FMVoyager.

1.4.1 Problem in terms of FataMorgana

Models imported into the FataMorgana system are huge CAD models with **hundreds of millions of triangles**. It is not rare that even the tiniest parts of these models, such as screws and nuts, are fully modeled to the smallest detail.

These models are **not optimized** for low-power portable devices such as Microsoft HoloLens. But they also cause problems on VR headsets, where rendering is significantly more expensive and demanding than a single 2D display.

Besides that, these models might be in **various formats** – not just typical triangle meshes. This project aspires to fix the issue of rendering large and complex scenes in systems such as FataMorgana.

1.5 Scope and limitations of the research

Optimization of rendering is a very complex, broad, and long-studied topic. Research of existing techniques in this project was bounded by techniques applicable to input models in various formats (not only meshes) that were reasonably **scalable** and **novel** (for a more detailed list of requirements, see Chapter 2.2). Many older techniques were researched but primarily only as a necessity for understanding newer research based on them.

The field of rendering optimization is still very active. Therefore, it is good to note that **several newer techniques were presented** while this project was being worked on. Some of them might be mentioned later in this text, but it is out of this project's scope to keep up with all possible new techniques.

This project **should not be used as a meta-analysis** of rendering optimization techniques since many research papers were thrown away right away after reading the abstract and realizing they do not meet the constraints of this project. Such papers will most probably not be mentioned in this project.

1.6 Structure of the thesis

Right after this **introductory chapter** (1), there is an **analysis of various LoD techniques** (2) followed by a more **detailed description** (3) of the technique that was **chosen as the best** for this case.

Then, **the implementation process** (4) is described, which, besides other things, also includes the reasoning behind chosen technologies and libraries. The whole **architecture** (5) is described in the next chapter. That is followed by a chapter with **implementation highlights** (6), such as reasoning for selecting low-level algorithms and approaches.

At the end, there is a **presentation of results** (7), **discussion** (8), standard **conclusion** (9), **references** (10), **lists of figures** (11) and **abbreviations** (12), and lastly, **attachments** (13).

2 LoD techniques analysis

Dozens of LoD techniques were researched throughout the history of computer graphics. This chapter will **explain** different approaches to LoD, **compare** them, and **defend** the selection of the LoD technique, which was implemented as the second part of this thesis.

2.1 LoD in general

Computer games were probably the first part of the computer graphics industry that used LoD. This is due to their nature of displaying large and detailed scenes in **real time** and high frame rates on consumer-grade devices of various power.

The most basic approach to LoD is to create **several polygonal meshes** representing the same object manually but with varying number of polygons (see Figure 2). This follows the basic principle – if the object is far away, the low-poly version of mesh is displayed, and vice versa.

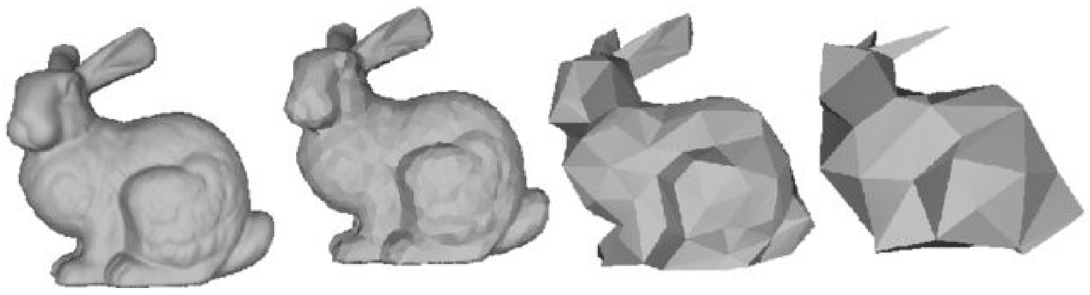


Figure 3 Comparison of different LoD levels on the Stanford bunny model. Source: YouTube[1]

2.1.1 Parameters for selection of LoD level

Distance from camera to model in a scene is the most common, easily available, and very reliable way to select which LoD level of the mesh should be used for an object in a scene.

[1] https://www.youtube.com/watch?v=mIkIMgEVnX0

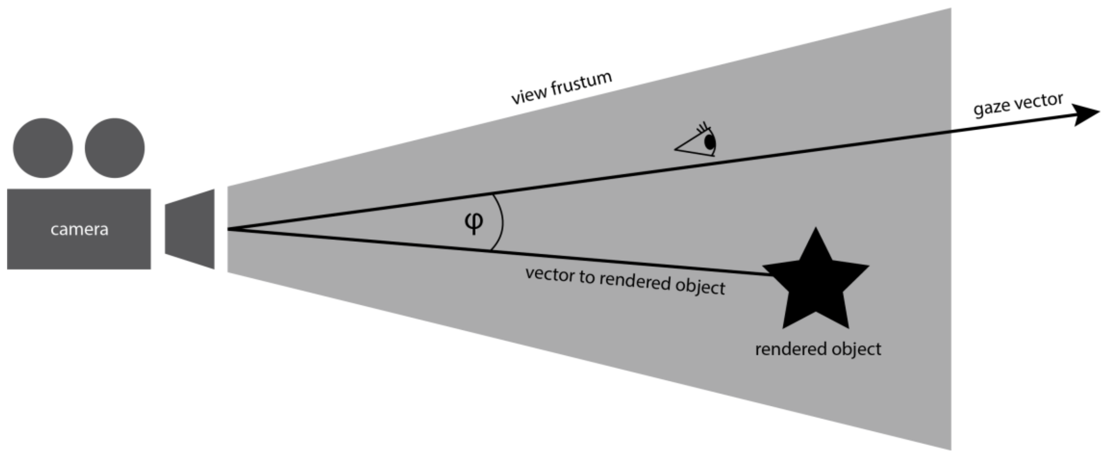


Figure 4 Illustration of the angle between gaze vector and vector to rendered object.

However, it is not the only parameter for this selection. The **angle between the user's gaze vector and the object** (depicted as φ in Figure 4) is also a usable parameter. Suppose the angle between the gaze vector from the camera center and the line from the camera center to the object is large. In that case, it means the object is in the observer's **peripheral vision**, and therefore, it does not require such a detail compared to an object in the middle of the camera. The human eye's anatomy shows fewer rods and cones on the retina's periphery, which results in fewer stimuli available compared to the middle.

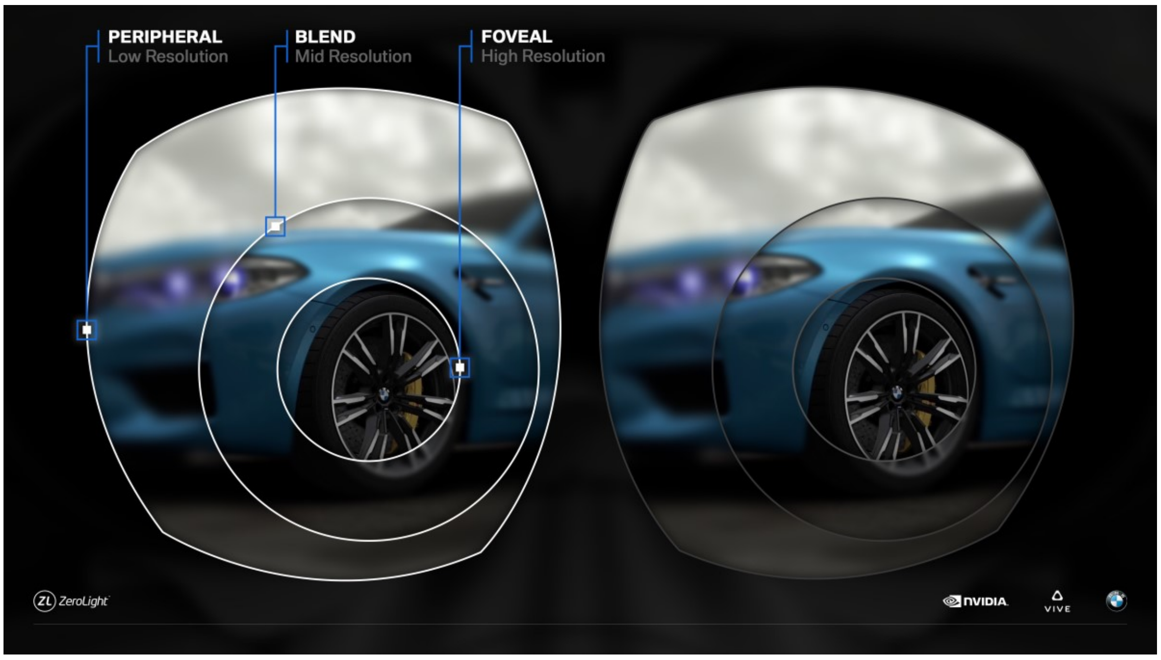


Figure 5 Depiction of the principle of foveated rendering. Source: Article "Foveated Rendering on the VIVE PRO Eye" on LinkedIn[2] by Chris O'Connor from ZeroLight

[2] https://www.linkedin.com/pulse/foveated-rendering-vive-pro-eye-chris-o-connor/

Since users can look anywhere on a standard monitor and due to the general lack of any widely used **eye-tracking system**, this is not usable in most common scenarios. Seeing models of lower quality on the edges of the monitor might look uncanny. This changes with modern AR/VR headsets, which can track the user's gaze using internal cameras. The technique in which the user's gaze direction is used to optimize rendering is called **foveated rendering** (see Figure 4).

Foveated rendering is not restricted to only displaying low-poly models in the area of peripheral vision. Such an area is also a good candidate for utilization of lower quality (but faster) shaders or even calling shaders less often than every frame. The second-mentioned technique is usually referred to as **variable rate shading**.

A good system would consider both distance and angle to the rendered model in selecting the LoD level.

There are other parameters for selecting LoD level, but they might be too subjective to particular cases or software. One of them could be a **priority of the object**. If software is used to, for example, present a car, the environment around it can be rendered in lower resolution.

On **battery-powered** devices, it might be a good idea to lower the rendering quality to keep the device **running longer**. Lower rendering quality also leads to **lower thermal output** of portable devices. This might be desired when the device gets overheated, e.g., due to the influence of the environment.

2.1.2 Continuous and discrete LoD

No matter how the importance of the model is calculated (distance to camera, angle, …), it is usually a **continuous value**. The camera can move closer to a model by a small amount, and the LoD system has to react somehow to this scenario.

If only a **discrete** amount of LoD models is available, changing from one level to another might cause undesirable visual effects, referred to as **"popping"**. Advanced systems can utilize techniques to mask this transition. Such approaches are for example:

- **Alpha Blending** - One LoD level decreases in opacity, and the other increases, as seen in Figure 5. This requires both LoD levels of the model to be rendered simultaneously, even though only for a relatively short transition time.

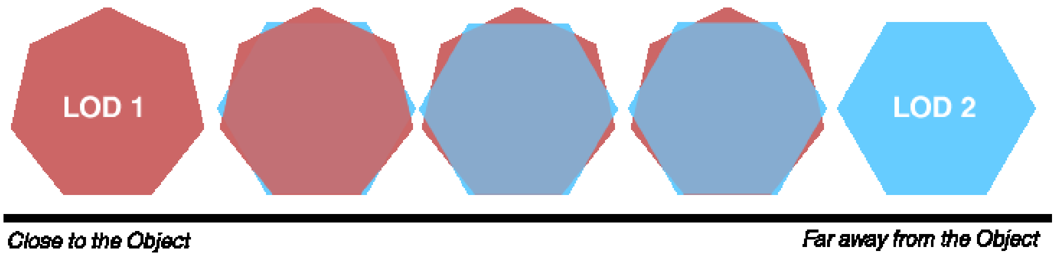


Figure 6 A blending for transition between 2 LoD levels. Source: Wikipedia - Popping (computer graphics)[3]

- **Geomorphing** - Directly changing mesh from one model to another is another technique (see Figure 6). Real-time geomorphing can be computationally expensive.

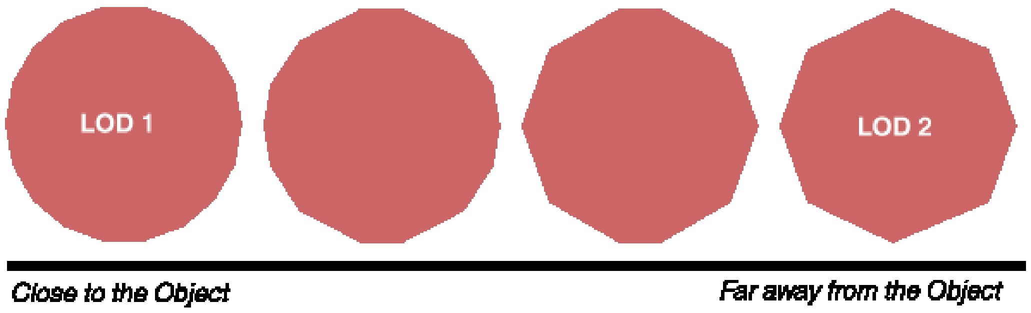


Figure 7 An illustration of geomorphing for transition between 2 levels of detail. Wikipedia - Popping (computer graphics)[3]

Other techniques for more **continuous LoD** can include, for example, adjusting the amount of rendered points/surfels in very small steps. This approach usually scales very well.

Discrete LoD is a traditional approach for this problem, as first presented by (Clark, 1976). In comparison, the term **continuous LoD** is sometimes called “**progressive LoD**” after (Hoppe, 1996).

2.1.3 Polygons vs. surfels vs. points

Manual creation of sets of **polygonal** meshes requires a lot of artists' time. Naturally, this led to the development of automated systems to simplify polygonal meshes. These algorithms usually involve some kind of **clustering of neighboring vertices** while trying to maintain the overall shape of the 3D model. Information about neighbors for each vertex is essential, and in case of its absence, expensive

[3] https://en.wikipedia.org/wiki/Popping_(computer_graphics)

pre-processing is required. Even after that, manual fixes and adjustments of meshes are often needed.

These disadvantages led to the analysis of **other approaches** to LoD, such as the use of points and surfels.

Points are usually handled as sets, referred to as “**point clouds**”. Their rendering is cheap and a good alternative to low-poly meshes. However, gaps between points must be handled by calculating an adequate size for each point. Moreover, visual effects such as **shadows** are often problematic to achieve.

More generalized primitives to points are so-called **surfels** (as first defined in (Pfister, et al., 2000)), from “surfel” = surface element. They are usually represented as points with associated sizes and normal vectors (their orientation). Visually, they can be rendered as oriented circles, ellipses, squares, or other planar primitives positioned in 3D space.

2.2 Criteria used for selection of LoD technique

As briefly mentioned in Chapter 1.5, there are several requirements for the selection of the LoD technique for this project:

1. **Input format** – Many formats of input models are already used in the FataMorgana system. And there are expected to be only more in the future. This is not just about various formats for 3D meshes but also non-standard representations such as output from 3D scanners and similar devices.
2. **Scalability** – Support for devices with strongly varying performance is critical since this technique is expected to be used on low-power standalone headsets such as HoloLens or Meta Quest, but also on powerful desktop PCs. Continuous LoD is a good candidate to fulfill this requirement.
3. **Novelty** – It is important to check how old the specific research is. The older it is, the higher the probability that newer research that improves it exists. On the other hand, if there is no progress in the field for a long time, it might mean it is a dead end, and research is focused on different techniques. In both cases, checking any newer research referencing these old ones is a good idea.

2.3 Review of relevant literature

The following subchapters contain brief **descriptions**, **advantages,** and **disadvantages** of researched techniques. Not all researched papers/articles are mentioned here. Some of them were found not to be relevant enough even to mention them as potential techniques.

2.3.1 Image-based rendering

Other than LoD, several techniques for optimizing rendering based on simplification of rendered scene were also considered - for example, **image-based rendering** and **sprites**.

These approaches somewhat resemble LoD because they show simplified versions of objects when they are too far from the camera. But instead of simplifying meshes, they show 2D images in 3D space, sometimes called **imposters** or **billboards**.

Most, if not all, simple image-based techniques suffer from low quality and visual artifacts. Several methods and data structures were created to address these issues. One of the most known are **Layered Depth Images** (LDI) described in (Shade, et al., 1998), later extended into structures such as **LDI Tree** (Chang, et al., 1999) and **Layered Depth Cube** (Lischinski, et al., 1998).

However, these techniques are rarely used nowadays due to a significant lack of quality and the generally complex nature of structures. Research in this field has **not advanced** anywhere in the last 20 years.

2.3.2 Progressive meshes

One of the older techniques for LoD is called **progressive meshes**. It was first described in (Hoppe, 1996). In simple terms, **progressive mesh** is a data structure containing the simplified version of input mesh and hierarchy of decimation operations, which lead to this simplified representation. Then, to get a better-quality mesh, these decimation operations are applied in reverse.

There were several attempts to implement this technique **on GPU**, such as (Hu, et al., 2009) and (Derzapf, et al., 2012).

It offers very good granularity of quality. On the other hand, disadvantages include **higher memory consumption** and, most importantly, strict requirements for input mesh. Not only does the input have to be a polygonal mesh, but it also has to contain information about the **connectivity of neighboring vertices**.

2.3.3 Point clouds

The point-cloud-based LoD methods have a significant advantage in final rendering since the **rendering of points is cheap**. The resulting quality can vary, but it should not be a huge concern since points would be used only on objects far from the camera. Alternatively, advanced **blending of points with polygonal meshes** could be utilized (Cohen, et al., 2001).

The notable disadvantage is the significantly more complicated handling of **shadows**, **occlusion culling**, and other techniques primarily developed for polygonal meshes. Also, converting polygonal mesh to a point cloud **requires a suitable sampling** method. However, sometimes data are already in the form of the point cloud, such as output from 3D scanners.

LoD techniques based on point clouds focus on the **decimation/simplification** of a set of points. This includes, for example (Pajarola, 2003) or (Shi, et al., 2011), where *k*-means clustering is used. This simplification process creates multiple versions of the same point cloud with coarser and coarser details.

A good simplification process for point clouds should consider the model's properties, such as **curvature**. More points should be preserved in high curvature places (compared to flat areas) since that usually signals more detail. This is done by, for example (Pauly, et al., 2002).

Even though the rendering of points is cheap, the whole rendering process might be challenging. This is because output is usually represented as **complex hierarchies** (such as point-octree), which need to be handled during rendering time - increasing complexity and performance hit for rendering.

2.3.4 Surfels

Surfels as rendering primitive were first introduced by (Pfister, et al., 2000). Since then, several techniques for their rendering and use in LoD have been presented.

For example, a combination of **surfels and billboards** (image-based impostors) in a hierarchical LoD structure was proposed (Holst, et al., 2007). This structure can also be combined with triangle-based LoD structures.

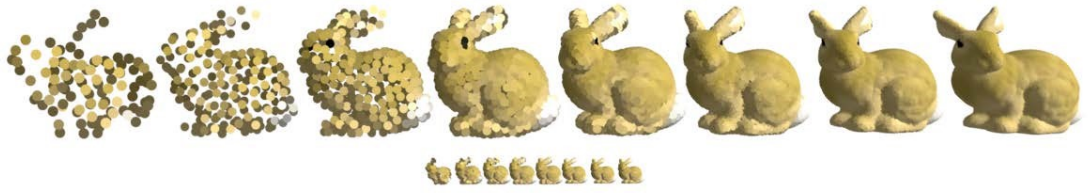


Figure 8 Model of Stanford Bunny consisting of varying amount of Blue Surfels. Note that the size of the surfels is not representative of actual rendering. Source: (Jähn, 2013)

A significantly simpler LoD structure with a sampling of meshes was introduced in (Jähn, 2013) as **Blue Surfels**. Thanks to complex sampling, the resulting structure for rendering is a simple **1D array of surfels**. The renderer decides how many surfels (how large prefix of this array) should be rendered based on LoD parameters. This can be seen in Figure 7. The sampling process itself was later described in more detail in (Brandt, et al., 2019) [Visibility-Aware Progressive Farthest Point Sampling on the GPU] and rendering with LoD selection in (Brandt, et al., 2019) [Rendering of Complex Heterogenous Scenes using Progressive Blue Surfels]. **Blue surfels are explained in more detail in Chapter 3.**

Gaussian splats

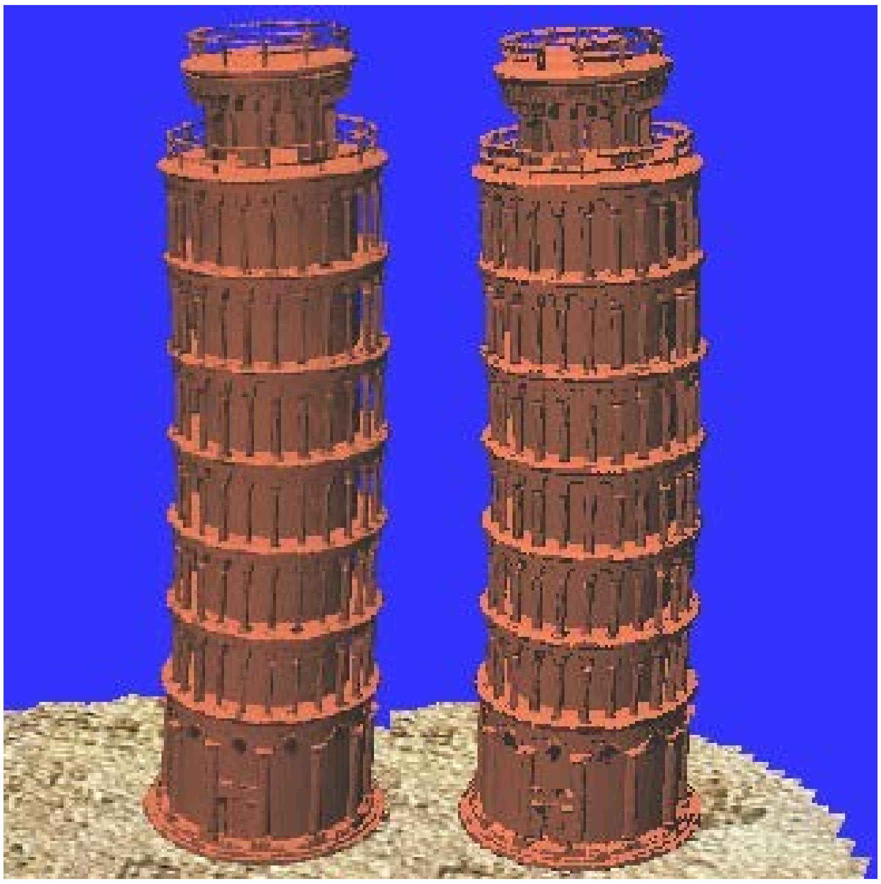


Figure 9 Comparison of Gaussian splats (left) and opaque squares (right). Source: (Coconu, et al., 2002)

Similar to surfels (or even their version) are primitives commonly referred to as Gaussian splats. As the name suggests, they use **Gaussian distribution** – more specifically, this distribution is used for the gradient of the alpha channel from the center of the splat to its edges. This technique **blends splats with others around them** to create a more natural look compared to opaque surfels.

However, this comes with higher rendering complexity due to alpha blending and non-standard depth testing. This was attempted to be solved using the old **A-buffer** technique (Carpenter, 1984), **z-offsetting** (Rusinkiewicz, et al., 2000), or complex hierarchical structures enabling rendering in **paint order** like the painter's algorithm does (Coconu, et al., 2002).

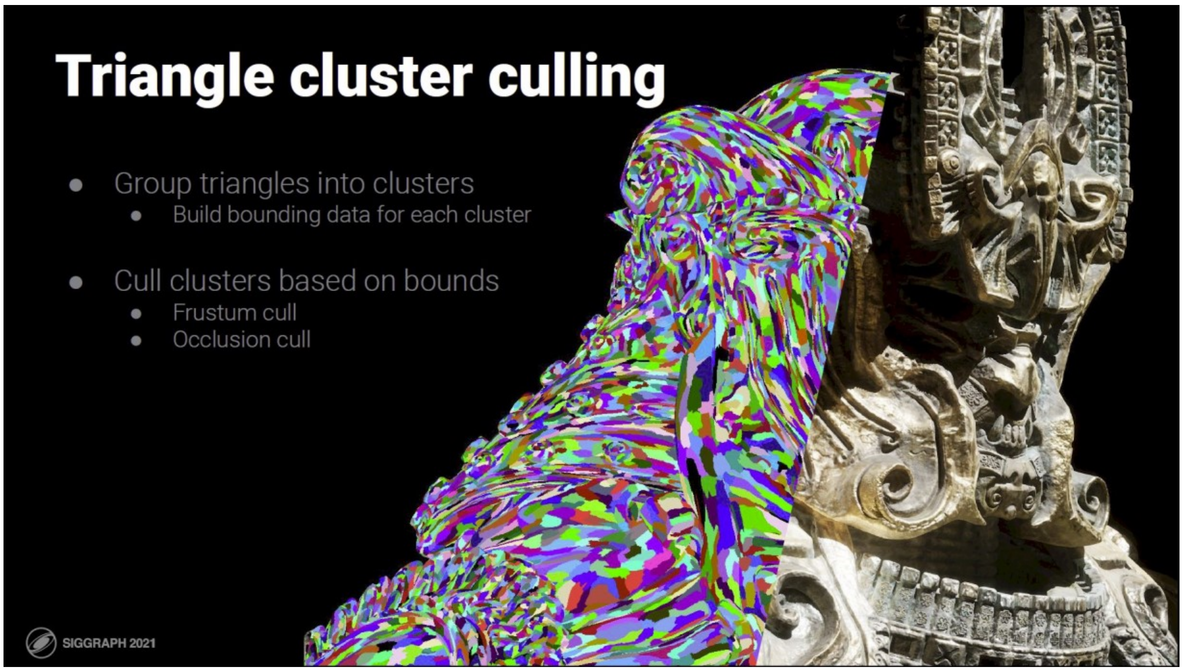
An increase in quality using Gaussian splats compared to primitive opaque surfels can be seen in Figure 8.

EWA filtering

Another technique trying to improve opaque surfels is called **Elliptical Weighted Average** (EWA) surface splatting (Ren, et al., 2002). It does require 2 render passes, however.

2.3.5 Virtualized geometry

A novel approach to LoD is virtualized geometry, with the most famous example being **Nanite**[4]. This system takes polygonal mesh on input and breaks it down into hierarchical clusters of triangle groups. These clusters are then swapped on the fly to provide varying levels of detail.



Triangle cluster culling

- Group triangles into clusters
  - Build bounding data for each cluster
- Cull clusters based on bounds
  - Frustum cull
  - Occlusion cull

SIGGRAPH 2021

Figure 10 Nanite and its clustering of triangles. Source: Nanite – A Deep Dive presentation at SIGGRAPH 2021.[5]

It provides a **considerable performance boost**, which makes it possible to display models with a very high number of triangles. Besides that, it also **compresses the mesh** itself, resulting in lower memory requirements.

It does have several drawbacks, however. First, it is implemented only in **Unreal Engine 4** - though there will undoubtedly be other implementations soon.

[4] https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/

[5] https://www.highperformancegraphics.org/slides22/Journey_to_Nanite.pdf

The **Unity version** of Nanite was crowdfunded through Indiegogo[6]. At the time of writing, Nanite works only with **DirectX 12** (and PlayStation 5 and Xbox Series S|X, but those are irrelevant to this project).

Other drawbacks include **no support** for **double-sided faces**, Android, and mobile chips (including Meta Quest), which means it **might not support HoloLens** either. Nanite also does not work with all materials and **translucent objects**. It works poorly with **long thin objects** (hair, grass, vegetation in general, etc.) and **can be used only on static meshes**.

Many rendering techniques are not currently supported by Nanite. The most important one (for this project) is the **lack of support for stereo rendering**.

2.3.6 Micro-Meshes

A technique called Micro-Mesh, developed and recently shown by Nvidia[7], is based on splitting each triangle of the base mesh into **micro-triangles**. These micro-triangles are later displaced to create a highly detailed model. Thanks to these micro-triangles, this technique offers an **inherit continuous LoD** – the number of rendered micro-triangles for each base triangle can be adjusted on the fly through blended decimation.

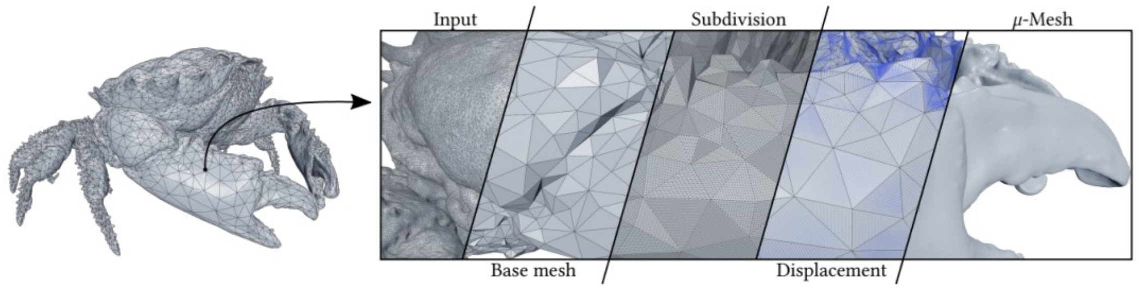


Figure 11 A visualization of conversion from input model to Micro-Mesh. Source: (Maggiordomo, et al., 2023)

Unlike Nanite, this technique works on **animated meshes**. Micro-triangles can also be easily **generated on the fly**, for example, for procedurally generated meshes. However, this requires the **latest generation of GPUs** – more specifically, GPUs ray-tracing capabilities are needed for the native generation of micro-meshes

[6] https://www.indiegogo.com/projects/the-unity-improver-nano-tech#/

[7] https://developer.nvidia.com/rtx/ray-tracing/micro-mesh

(Maggiordomo, et al., 2023). Same as Nanite, Micro-Meshes can also **compress** the original mesh to a fraction of its original size.

This approach's disadvantages include the necessity for the latest generation of GPUs. Micro-meshes also requires **input** to be in the form of a **polygonal mesh**.

Non-standard meshes or meshes with **errors** might cause problems for this algorithm. This includes, for example: open edges, non-manifold surfaces, overlapping UVs for textures, and similar. This will likely improve soon since research in this field is very new and active.

2.4 Discussion of analyzed techniques

The previous chapter reviews many relevant LoD techniques based on stated criteria. It discusses their advantages, disadvantages, and limitations.

There are also depictions of **research trends** for LoD algorithms. As can be seen, image-based LoD algorithms and progressive meshes went into the background of research, while the industry focused on **Micro-Meshes** and **virtualized geometry**.

The introduction of Nanite was a giant leap forward in a field where only minor progressive improvements were made in the last years. Its implementation into industry-level game engines such as **Unreal Engine** will surely bring more focus into this area of research.

Based on this literature analysis and considering all the advantages and disadvantages, selecting the **LoD algorithm** for this project was relatively easy. This selection is discussed and justified in the next chapter.

2.5 Justification for the chosen technique

The selected LoD technique, hereinafter referred to as **"Blue Surfels"**, fulfills all requirements set in Chapter 2.2. It brings even something extra to the table. This algorithm provides **scalable LoD** with extreme granularity, is novel, and is prepared to work with inputs in arbitrary format. Plus, there is a source code available. As a bonus, it is also tested with VR headsets.

On a more **personal note**, using a rasterization pipeline in a very non-standard way and compute shaders (basically a **GPGPU**) is an exciting topic for me. This is not a crucial decision factor. It is still motivating to explore such an exciting approach to graphics hardware.

2.5.1 Hybrid remote rendering

The extremely high granularity and output format of the Blue Surfels algorithm allow for something that can be referred to as hybrid remote rendering.

“Remote” in this case means that the hard work (conversion of input model to surfels) is done on a powerful machine (*server*). This is then sent to a low-power device (*client*).

“Hybrid” refers to the fact that the client still has to render something (surfels in this case), unlike traditional remote rendering where the server sends only final RGB images (and optionally depth maps). Rendering surfels is, however, still significantly cheaper than rendering huge models with millions of triangles. There are also no complex structures and computations used for rendered primitives – only a single 1D array of surfels.

High granularity and an iterative approach to Blue Surfels allow the server to send more and more surfels gradually as they become **available**. Or, as there is more bandwidth for that – surfels can be sent one by one, and they still provide an improvement individually to the rendered scene on the client device. That allows to use this technique even with very slow and **bad connection**.

For example, sending huge models through a slow internet connection might **not be possible**. Sending only a part of the 3D model is not practical (and might not be possible at all, depending on the format of the 3D model). On the other hand, sending just a **few Blue Surfels is way better than having no model at all** (or having only a placeholder, for example, in the form of a bounding box).

It is worth noting that no matter what type of remote rendering is used, some things should be **rendered locally** – such as hands. They usually require very little computational power (low triangle count and simple shading), their geometry and position change every frame, and they should be as low latency as possible while being resistant to problems caused by networks.

3 Blue Surfels

This chapter talks about **Blue Surfels** algorithm as presented by

- Claudius **Jähn** (from DeepL GmbH, Cologne, Germany),
- Sascha **Brandt**,
- Matthias **Fischer**,
- and Friedhelm Meyer auf der **Heide** (the last 3 are from Heinz Nixdorf Institute, Paderborn University, Germany)

in these 3 papers:

- Progressive Blue Surfels (Jähn, 2013),
- Rendering of Complex Heterogenous Scenes using Progressive Blue Surfels (Brandt, et al., 2019),
- and Visibility-Aware Progressive Farthest Point Sampling on the GPU (Brandt, et al., 2019).

More specifically, the first part of this chapter describes the working of this algorithm as presented in 3 papers. This should be a **replacement for reading** all these papers – at least for a high-level understanding of the algorithms used in this project.

The second part of this chapter mentions the **existing implementation** of the Blue Surfels algorithm into the university's rendering platform. It also provides arguments on why a complete rewrite of the algorithm was required.

3.1 Overview of papers

All aforementioned research papers talk about the same problem from **different views**. The main idea of the algorithm, as mentioned in Chapter 2.3.4, is the **uniform sampling of a 3D model** to create a surfel representation of it. These surfels are ordered in a 1D array in a way that rendering longer and longer prefix results in more details while maintaining the uniformity of rendered surfels' distribution.

3.1.1 The origins

The first mentioned paper, **Progressive Blue Surfels** (Jähn, 2013), is a concise introduction to this problem. Most of the mentioned processes and algorithms are described on a very **high level**. Others are very **primitive versions** of algorithms with notes that they should be **improved/replaced** in future work. No benchmarks, comparisons, or analyses are provided.

Since the other 2 papers cover and build upon everything in this paper, it is **irrelevant to this project**.

3.1.2 Rendering

Rendering of Complex Heterogenous Scenes using Progressive Blue Surfels (Brandt, et al., 2019) paper does mention **progressive sampling**, but only the significantly **simplified version**. It consists of a **greedy permutation** (maximization of the minimal closest distances between neighboring surfels) with a heuristic to make it faster while keeping the sampling quality reasonably good.

The main focus of this paper is the actual rendering of surfels. This includes primarily **calculation of array prefix length** for desired **surfel size**. Subsequently, another formula shows how to calculate **surfel size** based on the **rendering time** of the last frame. An amount of rendered surfels (array prefix length) is expected to be updated as often as every single frame.

The rest of the paper consists of performance **benchmarks** and a **quality comparison** of rendered surfels to baseline mesh rendering.

3.1.3 Sampling

The last paper, **Visibility-Aware Progressive Farthest Point Sampling on the GPU** (Brandt, et al., 2019), is the most important one since it describes an advanced **sampling** process with blue noise characteristics (more in the next chapter).

Note that this sampling algorithm is **different** and especially significantly more advanced than the one briefly mentioned in the **previously discussed paper**.

Besides the detailed description of the algorithm, this paper also contains a very deep **analysis** of results, including differential domain analysis, timings compared to reference algorithm, statistics, etc. There is also a comparison of various values for algorithm **parameters**.

No rendering algorithms are discussed in this paper. There are only mentions of LoD as one of the applications for the sampling algorithm.

3.2 Sampling algorithm

This whole project is based on the advanced **sampling process** described in the Visibility-Aware Progressive Farthest Point Sampling on the GPU (Brandt, et al., 2019) paper. Therefore, this chapter is dedicated to **description of this algorithm** as presented in the paper.

The overall **goal** of this sampling algorithm is to sample the visible surface of the 3D model so that the resulting samples have blue noise characteristics for every prefix of the output sequence of samples.

"Blue noise" in this context means that samples are distributed **uniformly** (in spatial dimension) with **low regularity** (no visible patterns). In computer graphics, blue noise is also frequently used for applications such as dithering, as can be seen in (Ulichney, 1988).

All this is done with a focus on utilizing the **highly parallel** architecture of modern GPUs, gaining a huge **performance advantage** compared to CPU-only algorithms.

Note that this algorithm, as presented in the paper, is **not one-to-one** compared to the algorithm actually implemented in this project. Also, parts of the algorithm in the following subchapters are mentioned in a relatively **high-level** approach since low-level workings are explained in detail in the rest of this thesis. Similarly, implementation-specific details are excluded from this chapter.

The following **flow-diagram** in Figure 12 shows overview of individual stages of algorithm and main loop. It also illustrates how compute and rasterization pipelines are utilized in a "zig-zag" manner.

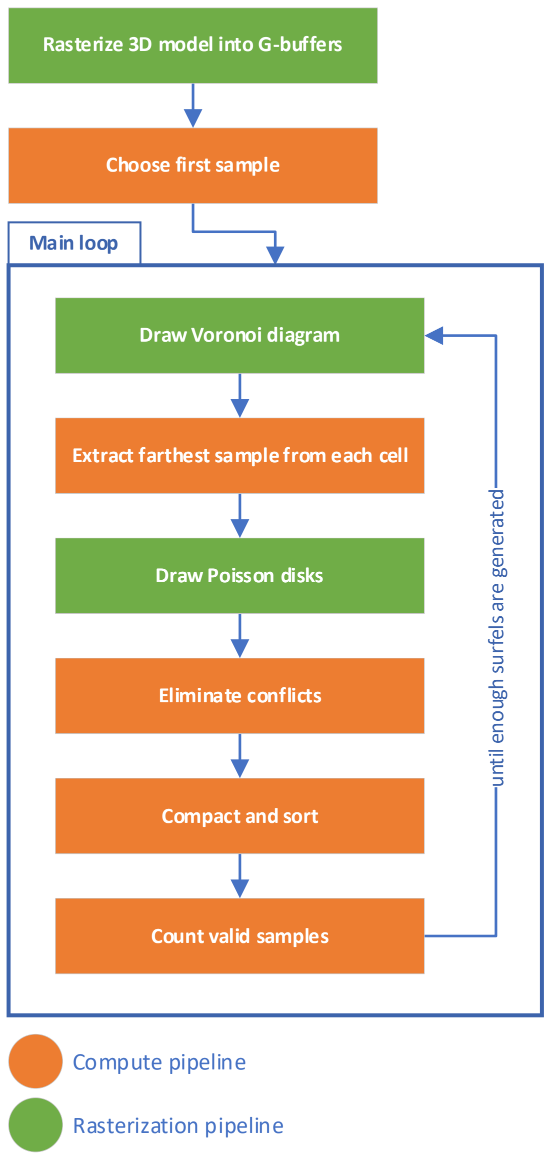


Figure 12 A flow-diagram representing individual stages of the whole algorithm. Orange stages use compute shaders and green ones run on rasterization pipeline (vertex, geometry, and fragment shaders).

3.2.1 Rasterization

The algorithm starts with an input model being **rasterized from various directions/views** – originally, pre-generated positions of cameras on the sphere around the input model are used.

This resembles deferred rendering since the result of rasterization is a set of **G-buffers**. Specifically, 3 buffers containing **position**, **normal**, and **color** for each pixel. This can be seen in Figure 13. G-buffers are stored as **layered textures**, where each layer corresponds to 1 camera.

No representation of the original input model is used in the algorithm from this point onward.

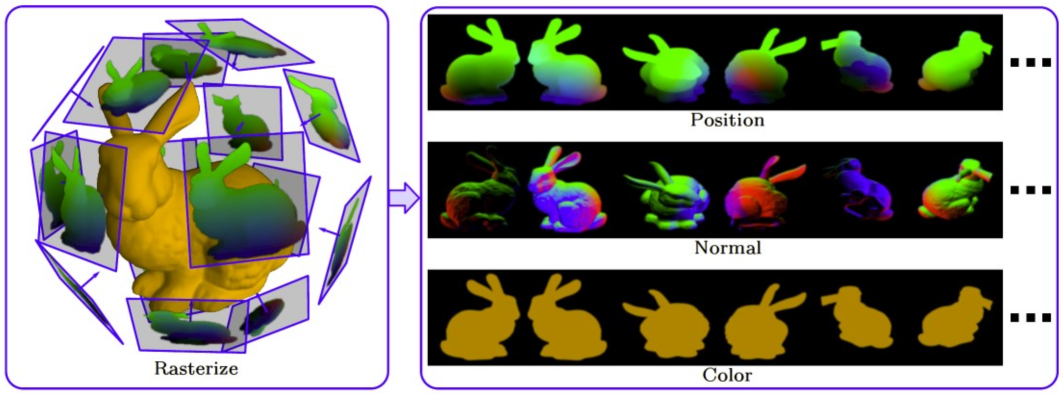


Figure 13 Rasterization of the object from several directions (left) and storage of results into various G-buffers (right). Source: (Brandt, et al., 2019)

3.2.2 Initial sample selection

After rasterization, 1 random **sample is selected** from G-buffers. Sample, in this case, means a reference to a **valid** (= non-background) **pixel** of G-buffer texture. This is not a reference to a particular pixel but rather a position of pixel applicable to all G-buffers since each G-buffer represents different parameters of the same rasterized object.

This sample is stored in a **sequence of samples S**.

3.2.3 Sample format

Each sample is represented as a **single 64-bit float**. The sample **radius** is stored in the 32 most significant bits, while the rest is used for **texture coordinates**. These coordinates are stored in the format of **12:12:5:3** bits as follows:

- **2 x 12** bits for **UV coordinates**,
- **5** bits for the **texture layer**,
- and the last **3** bits for the **MIP level**.

Having the 32 most significant bits represent a sample radius (while the whole sample is a single 64-bit float) has one significant advantage – **easy sorting**. The **progressive** nature of output is achieved by sorting of the samples by the **decreasing sample radius**. Therefore, this format allows a simple comparison of just these 32 bits of each sample to get correctly sorted output. **Radix sort** is ideal for cases like this. Furthermore, the whole 64-bit number is directly comparable using a **single compare operation**.

Invalid samples are stored as a 64-bit representation of the **number 0**. This is also a default value for the whole sequence S.

3.2.4 Main loop

After the initial sample is selected, the algorithm's main loop starts. It loops until **no more samples** can be extracted or the **desired number of samples** is generated (one of the input parameters).

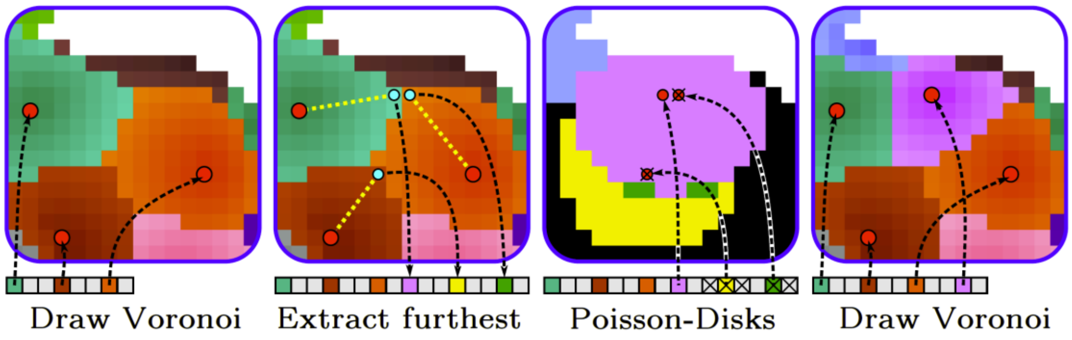


Figure 14 Simplified visualization of a single main loop iteration: from drawing of Voronoi diagram, extraction of the farthest samples, and elimination using Poisson disks. The right-most picture represents an updated Voronoi diagram used for the next loop iteration. Source: (Brandt, et al., 2019)

Voronoi diagram

For each camera view, a **discrete 3D Voronoi diagram** is calculated.

The idea is that each **sample** in sequence S acts as a **Voronoi site** (a generator for Voronoi cell). A **disk is rasterized** (directly on layers of G-buffers; for details, see Chapter 3.2.5) for each sample. The **distance** to the Voronoi site is subsequently written into the **depth buffer**. This utilizes a highly optimized depth buffer to assign a given pixel to the **closest sample**, creating a Voronoi diagram. This algorithm is based on (Ip, et al., 2013).

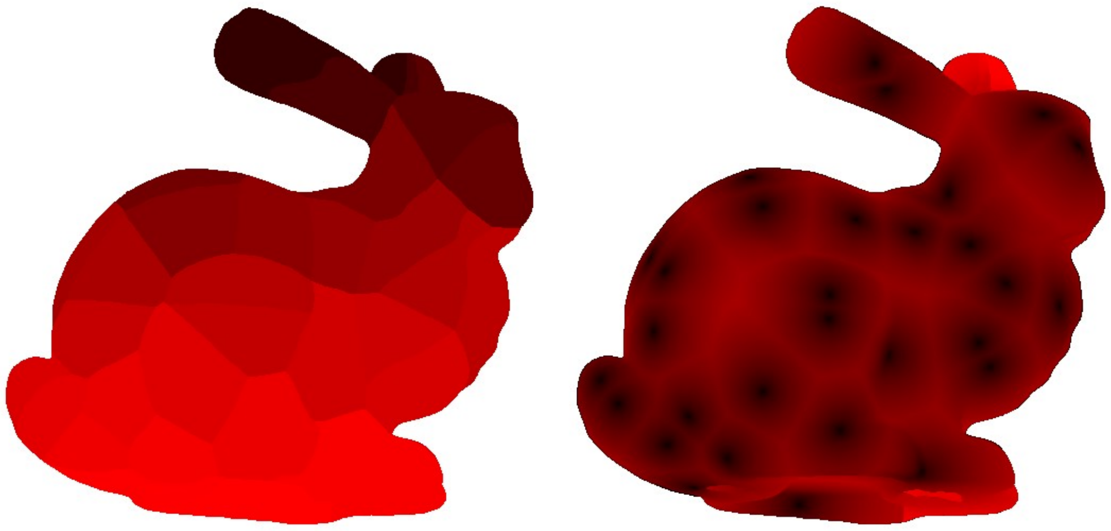


Figure 15 G-buffers for Voronoi diagram algorithm stage. Each shade of red in the left picture represents a single Voronoi cell. The right picture then depicts the distance of each pixel from its Voronoi site on a scale of black (nearest) to red (farthest). The centers of the black blobs are the Voronoi sites. Used model: Stanford Bunny[8]

The output of this stage consists of 2 G-buffers. One stores the **ID of the Voronoi cell** for each pixel (left picture in Figure 15). This ID corresponds to the sample's index in sequence S, which the Voronoi cell was created from. The second G-buffer stores the **distance to the closest Voronoi site** (right picture in Figure 15).

The left-most part of **Figure 14** shows Voronoi cells created from samples, while the right-most picture shows a Voronoi diagram with an **additional cell** created. The diagram with the added cell is used in the next loop iteration.

Farthest sample extraction

When the Voronoi diagram is created, it is time to find each cell's locally **farthest discrete point** (pixel of G-buffer). This can be seen in the second picture in Figure 14. This farthest point becomes a sample. And since each initial sample from the sequence S creates exactly 1 new sample, the total number of samples is **doubled**.

This also means that the corresponding **new sample** for each site is at a **strictly defined index** in sequence S. For the initial sample with index `i`, the new sample's index would be `(m + i)`, where `m` is the current number of samples in S.

[8] https://graphics.stanford.edu/data/3Dscanrep/

The farthest point is extracted in a single pass of a compute shader. G-buffers (containing Voronoi cell ID and distance to site) from the previous algorithm part are used. An **atomic max** operation is performed to select a sample with the largest distance to its Voronoi site. This sample is then written into sequence S at index `(m + i)`, as mentioned above.

Conflict removal

Not all new samples extracted from Voronoi cells are usable. For example, if multiple new samples are **too close to each other** – this can be seen in the second and third picture of Figure 14. Of 3 new samples (turquoise circles), 2 are **rejected** as being too close to the remaining one.

This "conflict removal" is done by rasterizing **Poisson disks** (similarly to disks used to create the Voronoi diagram) for each new sample. The disk's radius is equal to the distance from the new sample to the Voronoi site it from created from.

When **overlapping disks** appear, the one with the **largest radius** wins. This is illustrated in the third picture of Figure 14. The selection of the disk with the largest radius is done using a **depth buffer** – the same way as with the distance to the Voronoi site in the "Voronoi diagram" part of this algorithm, just with different clear value and comparison function for the depth test.

Once indices of Poisson disks are written into the G-buffer, a compute shader is used to go through new samples in sequence S and mark them **invalid** if the **neighbor's Poisson disk overlaps** them.

Compact and sort

The previous part of the algorithm causes **gaps of invalid samples** in sequence S. Furthermore, new valid samples are **not sorted**. To achieve progressive sampling, sorting by decreasing the sample radius is required. A "compact and sort" algorithm is performed to fix these issues.

Thanks to the **format** of samples (both valid and invalid; see Chapter 3.2.3), compacting (removal of gaps) and sorting can be done effectively on GPU using **parallel radix sort**. The inner working of this sorting algorithm is directly based on (Satish, et al., 2009).

After sequence S is compacted and sorted, a **count of valid samples** is updated. This number is needed to decide whether to **quit or continue** the main algorithm loop.

3.2.5 Rasterizing disks

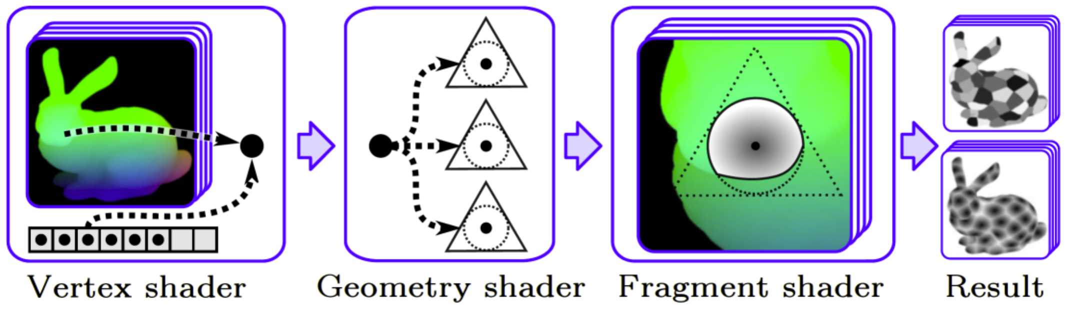


Figure 16 Process of rasterization of disks for purposes of Voronoi diagram. Vertex shader creates a point for each sample (using G-buffers), geometry shader creates a triangle from each point, and fragment shader discards fragments, leaving only an inscribed circle behind. The right-most pictures show resulting G-buffers – they are analogous to Figure 15. Source: (Brandt, et al., 2019)

Both **Voronoi diagram** creation as well as conflict removal using Poisson disks require the rasterization of disks. This is done using the standard rasterization pipeline of modern GPUs. The process can be seen in **Figure 16**, while a more detailed description is available in the rest of this chapter.

Vertex shader

The sequence of samples S is used as a **vertex buffer**. Vertex shader takes each sample, and based on texture coordinates stored in it, **position** and **normal vector** are read from G-buffers. These parameters are passed to the next stage.

Geometry shader

Geometry shader takes each sample as a point and creates an **equilateral triangle**, whose inscribed circle:

- has **center** at the **sample's position**,
- **rotation** perpendicular to the **sample's normal vector**,
- and **radius** identical to the **sample's radius**.

Fragment shader

Lastly, the fragment shader **discards** all fragments that are further away from the inscribed circle center than the circle radius, leaving only **fragments representing the circle**. A similar process is also visualized in Figure 17.

3.2.6 Output

The very important notable thing about this algorithm is the form of its output. It consists of a single **1D array of samples**. These samples are arranged in a way that **any prefix** of this array creates a **progressive sampling** of the input model. For an illustration of this, see Figure 8. As a result of that, the rendering itself can be done in a **single draw call** – just with a variable prefix length.

The number of rendered samples (prefix length) depends on general parameters for selecting the LoD level – for more details, see Chapter 2.1.1. It is important to note that the output of this algorithm is usable as **continuous LoD** with **extreme granularity**. Prefix length can be increased/decreased by as little as a **single sample** in each frame without processing overhead. Only the **surfel size has to be updated**. Fortunately, this value is the same for all the surfels at a single draw call, and its calculation is computationally trivial.

Samples can be easily **converted to surfels** as part of preprocessing or **rendered directly**. Direct real-time rendering of samples is less convenient since it requires having all relevant G-buffers bound, and reading from them will result in unnecessary **performance hit**.

3.3 Existing implementation

PADrend[9] (**P**latform for **A**lgorithm **D**evelopment and **Rend**ering) is a software system for virtual walkthroughs in 3D scenes developed at Heinz Nixdorf Institute, University of Paderborn, Germany. Authors of Blue Surfels papers implemented surfel generator and renderer as a **plugin** into this system.

The whole implementation is written in a combination of **EScript** and **C++**. EScript (different from eScript, a scripting version of Erlang) is a scripting language for controlling C++ applications explicitly developed for PADrend. The use of this language and the fact that implementation is very **tightly integrated** into PADrend were good reasons to completely rewrite the implementation rather than use the existing one and try to adjust it.

PADrend's implementation of Blue Surfels also uses **OpenGL** as a graphics API. This API is rather **old**, **lacks options** for deep **optimization**, and its **support**

[9] https://www.padrend.de/

will only **worsen** in the upcoming years. It is understandable to use OpenGL in an academic environment since it is easy to learn, use by students, and prototype ideas quickly. But there are better alternatives for professional applications nowadays.

The **lack of comments** and documentation also did not favor using the existing implementation.

Therefore, a decision to **completely rewrite** this program was made. Papers describing Blue Surfels were good enough to write a new implementation from scratch. The existing implementation code was still used in a few places **as a reference**, especially the **shaders**. The architecture of **CPU code** was created completely **from scratch** without any influence from the existing implementation.

4 Implementation process

The implementation process, from the **selection of software**, **testing hardware**, and **integration** into FataMorgana, as well as a discussion of **technical difficulties**, are all presented in this chapter.

4.1 Selected technologies

This subchapter discusses the reasons for this project's technologies - **IDE**, **programming language**, **APIs**, **libraries**, etc. For a quick summarization of the following subchapters, skip to Chapter 4.1.8.

4.1.1 Programming language

No critical computationally intensive part of this project is performed on the CPU. Therefore, the selection of CPU programming language was **not crucial** and came down to support for the binding library for selected graphics API (more in Chapter 4.1.5), ease of use, and **compatibility with the rest of the FataMorgana** system.

C/C++ is the best for working with all relevant graphics APIs without the need for any binding library.

C# was chosen mainly due to its simplicity (both in writing code and setting up a project with multiple files) but most importantly due to the ease of **integration into FataMorgana**. The core of this system is written in C#, and therefore, any interoperability is as easy as possible.

The **version of .NET** in the FataMorgana system is frequently updated to the most modern and stable version available. It also has minimal impact on interoperability between projects. Due to all these reasons and the fact that newer versions of .NET frequently bring a lot of useful and relevant improvements (for example, into C# language), it was decided that this project will be written using **.NET 6** - the latest stable version at the time of the beginning of this project. Once **.NET 7** was released in a stable version, the project was upgraded to it.

4.1.2 Development environment

Visual Studio comes up naturally as the #1 IDE for working with any C# project, especially larger ones (which might require profiling and other features unavailable in Visual Studio Code and similar IDEs). Therefore, the selection of IDE was easy.

4.1.3 Source control

Git was selected as source control software simply because the FataMorgana project is already hosted in a Git repo.

To make work on this project possible, even after I left Pocket Virtuality, the project was moved from the company's Git repo to a private one. A simple remote or clone change was impossible since the company's repo contained all FataMorgana projects. Project files had to be copied manually. This led to the loss of Git history, but it did not cause any issues.

4.1.4 Graphics API

The selection of graphics API in an application like this is **crucial** since most of the computation is **performed on GPU**, and the differences between available APIs are enormous. If single-purpose and non-Windows graphics APIs are excluded (such as Metal for Apple products and PSGL for Sony PlayStation), there are only so many options.

OpenGL

OpenGL[10] is **multiplatform**, **easy to use** (at least as far as graphics APIs go), but rather **old**. The Khronos Group (the group behind OpenGL) is increasingly shifting focus to their newer API called Vulkan. Therefore, support for it is lacking behind and is not expected to improve. Besides that, OpenGL does not support low-level optimizations and techniques that modern APIs offer.

Vulkan

Vulkan[11] is **modern** and **multiplatform** graphics API. Its advantage is the ability to optimize code on a **very low level** by providing graphics driver (precisely,

[10] https://www.opengl.org/

[11] https://www.vulkan.org/

Vulkan runtime in this case) as much information about provided data structures and operations as possible. Vulkan also allows programmers to utilize the multithreaded nature of modern computers fully. Its main disadvantages are a **steep learning curve**, lack of proper tutorials/troubleshooting guides, and hard-to-read official documentation.

DirectX

DirectX[12] is a graphics API with almost 30 years of history. It is developed by Microsoft and works **only on Microsoft products**, such as Xbox gaming consoles and Windows OS. The most commonly used major version nowadays are 11 and 12. In the case of DirectX, **versioning is very misleading** – while differences between DirectX 9, 10, and 11 are substantial, the core concept is still the same, and programmers can quickly adapt to newer versions. The difference between versions 11 and 12 is almost the same as the difference between OpenGL and Vulkan. DirectX 12 works on entirely different core principles and exposes many low-level features, similarly to Vulkan.

Translation layers/emulators

Translation layers and emulators, which allow users to play games written in **graphics API not supported on their system**, are improving. A massive boom in this field was recently caused by the popular Steam Deck gaming console, which runs on a Linux-based operation system called Steam OS. This naturally required a good emulator since most games are written in DirectX. However, these emulators are still far away from being perfect.

Decision process

Initially, **DirectX 11** seemed like the best candidate for graphics API for this project because it is already extensively used in the FataMorgana system, so various libraries could be **reused in other projects**. Besides that, DirectX 11 is relatively widespread, with many tutorials for it.

Unfortunately, in the middle of development, DirectX 11 proved unusable because it **does not support one critically required instruction in compute shader** – *atomic max on 64-bit floating-point numbers*. Workarounds, such as locks and

[12] https://learn.microsoft.com/en-us/windows/win32/directx

memory barriers, were considered and tested, but the performance hit was too big (by several orders of magnitude) – see Chapter 6.14.

The usage of **CUDA** or other similar GPU-driven libraries for high-performance computing was also considered. Still, interoperability between these libraries and graphics API causes performance bottlenecks and general problems with development and troubleshooting.

Graphics APIs that support this critical atomic operation are OpenGL, Vulkan (through extension), and DirectX 12. The use of OpenGL was quickly dismissed due to a lack of low-level optimization and a gradual loss of support. Choosing either **Vulkan** or **DirectX 12** meant completely **rewriting existing code** and no benefit of sharing libraries with existing projects from the FataMorgana system. Also, both APIs are more or less **equally difficult to learn**, and none of them were used by any employees of Pocket Virtuality before. One significant decision parameter was multiplatform support. **Vulkan is the winner** in this aspect as well whole decision process for graphics API.

Windows 10 does support DirectX 12; however, it is **not the version** that supports atomic max on 64-bit floats. To get this functionality, it is required to use DirectX 12 with Feature Level 12.2 and Shader Model 6.6. This requires WDDM 2.9, which is technically possible to use in Windows 10 but only in builds available through Dev Channel. Builds in Beta and Release Preview Channels were not usable. A few development builds of Windows 10 with support for WDDM 2.9 were tested, but they were all very unstable and generally unusable for everyday work. Moreover, selecting Dev or Beta Channel as Windows Insider in Windows 10 nowadays forces installation of Windows 11. Getting Windows 10 with all the required features through normal means is impossible. This would further **limit usable platforms** in case DirectX 12 was chosen.

The new dedicated **Intel Arc** GPUs are also worth considering in the equation. These GPUs are based on entirely new architecture, and the focus was obviously on **new graphic APIs** such as DirectX 12 and Vulkan. There are known issues with support and performance in games running older versions of DirectX and OpenGL as they are internally translated/emulated.

4.1.5 Graphics API binding library

Vulkan’s API is written natively in **C language**. This means that a **binding library** for C# was required. Binding libraries are low-level enough (at least between C# and C/C++) not to cause significant performance bottlenecks. And since all of the performance-critical work is computed in shaders on GPU, usage of binding library compared to native calls to C API does not hurt the final performance of this application in any noticeable way.

There are 2 major C# binding libraries for Vulkan. **Vortice** and **Silk.NET**. Other libraries, such as **vk** and **VulkanCore**, have not been in active development for several years and, therefore, were ignored.

Vortice is developed primarily by a **single person** with few contributors. **Silk.NET**, on the other hand, is officially developed under the umbrella of the **.NET Foundation**. This means significantly more contributors, more end users, and more frequent releases. Both libraries' quality and ease of use seemed to be more or less the same, and both are open-source projects with the same licenses (MIT). Due to these advantages of Silk.NET over Vortice, the final decision was made to **use Silk.NET as a binding library**.

While the project was still developed in DirectX 11 (which also offers only native C-based API), **SharpDX** was used as a binding library from C#. This choice was made purely from the standpoint of SharpDX being **already used** in FataMorgana. Otherwise, using this binding library is not recommended, as it was abandoned in 2019. Silk.NET would probably be a good alternative as it offers all the above advantages and provides DirectX bindings. However, no further research was done in this field as it was irrelevant.

4.1.6 Shader language

There are 2 major shading languages: **GLSL**, used in OpenGL, and **HLSL**, used in DirectX. Vulkan, however, relies only on low-level shader code representation called **SPIR-V**. There are compilers for both GLSL and HLSL into SPIR-V. The decision between GLSL and HLSL unlimitedly boils down to personal preference since the vast majority of extensions are supported. That is, unless the shader code is supposed to be shared with OpenGL or DirectX, which was not the case.

GLSL was chosen primarily for personal preference and partially due to the availability of a better GLSL to SPIR-V compiler plugin for Visual Studio.

4.1.7 Graphics debugger

The selection of a graphics debugger was not straightforward and required a bit of trial and error. This was mostly due to **high demands for support** of advanced features and extensions.

Visual Studio Graphics Debugger is great for **quickly** inspecting the rendering process and results but **lacks advanced tools**. But most importantly, it does **not support Vulkan**. However, it was used when the project was written in DirectX 11.

Regarding actual graphics debuggers (not just profilers) that support Vulkan, there are only 2 options: **RenderDoc** and **Nvidia Nsight Graphics**. Both are **advanced**, and both were used to some extent. However, RenderDoc appeared to be **more suitable** for debugging most of the cases in this project. Some advanced tools needed for this project's development include a step-by-step shader debugger and the ability to work with layered textures. RenderDoc fully supports both. Besides that, it is also open-source, with great support from the community and the creator.

RenderDoc API

Graphics debuggers usually expect the standard **structure of the rendered frame** – 1 or more Draw/Compute commands with Present command denoting the end of frame (tells GPU that the current frame can be presented to the screen). However, this project uses a rasterization pipeline in a very unusual way – it works iteratively and completely **off-screen** with **no Present command** involved.

To properly debug situations like this, RenderDoc **provides API** that (besides other features) allows apps to mark the start and end of rendered frames. In this project, a single iteration of the algorithm acts more or less as a single frame and was set this way.

4.1.8 Recapitulation

To summarize previous subchapters, selected technologies are as follows:

- Development environment: **Visual Studio** (standard solution with projects) and **Git**

- Languages: **C#** for CPU side and **GLSL** for shaders
- Graphics API: started in DirectX 11 but later changed to **Vulkan** through the **Silk.NET** binding library
- Graphics debugger: primarily **RenderDoc**, a bit of **Nvidia Nsight Graphics**

4.2 Dependencies

Using this project as part of a commercial product leads to limitations in selecting dependencies. More specifically, **licensing**, probability for long-term **support**, and **development stability** (how probable are large breaking changes to API) had to be considered.

4.2.1 FataMorgana

Plans to use DirectX 11 to get the option to share some of the libraries with the FataMorgana system could not be fulfilled. Also, there was no code using Vulkan in this ecosystem. Due to these reasons, there are absolutely **no dependencies** on the FataMorgana system, and this project can currently be used as a **standalone**.

4.2.2 Libraries

All external dependencies are managed through **NuGet** packages since it is the easiest and most straightforward way to add 3rd party libraries to C# projects.

These packages include:

- **CjClutter.ObjLoader** – loader library for .obj files (containing 3D mesh data),
- **CommandLineParser** – simple utility library to help with parsing command line arguments,
- **Evergine.Bindings.RenderDoc** – C# binding for RenderDoc API,
- **Silk.NET** – various packages related to C# binding to Vulkan API or working with graphics APIs in general, more specifically:
    - **Maths** – math library used for standard graphics APIs operations such as vectors, matrices, etc.,

    - **Vulkan, Vulkan.Extensions.EXT, Vulkan.Extensions.KHR** – Vulkan binding itself and bindings to 1st party KHR (Khronos) and 3rd party EXT extensions,
    - **Windowing, Windowing.Common, Windowing.Glfw, Windowing.Sdl** – packages related to the windowing system used by Silk.NET,
- **SpirVTasks** – a small utility tool that adds tasks into Visual Studio for automatic compilation of GLSL shaders into SPIR-V.

Vulkan frameworks

There are several libraries and frameworks specifically made for Vulkan. However, this project **does not use any of them**. The decision to write everything **from scratch** and use pure Vulkan was made to have complete and low-level access to everything needed.

Also, most of these frameworks are designed for **standard rendering**. This project uses Vulkan in a highly **non-standard approach**; therefore, finding a framework that would provide everything needed for this project was deemed problematic, especially since it would have to be comfortably usable from C#.

Licensing

All aforementioned 3rd party packages are distributed under the **MIT License**. This license permits both private and **commercial use**, **modifications**, as well as distribution of code using these libraries. Therefore, there is no issue with using these packages in commercial systems such as the FataMorgana.

4.3 Integration into FataMorgana

As mentioned in Chapter 4.2.1, this project has no dependencies on any existing part of the FataMorgana system. It can work **entirely independently**.

Integration of this project into FataMorgana (or any other similar system) still **requires much work**. This consists mainly of the workflow to select a model, split it (if necessary), convert it, call this project to generate surfels, and save surfels. Neither the rendering of surfels (see 6.8) nor the selection of the correct number of surfels to render (or the original mesh) are complex.

The only parts of this project that need to be adjusted based on the system around it, are **input and output formats**.

4.3.1 Input

Not only can mesh input be in many different formats, but the input does not have to be a mesh at all. Output from a **3D scanner** or similar device can be used as input to this project. This is further discussed in Chapter 6.6. Transfer of input can be done in various ways as well – as a **file** on disk/database, a **pointer** (/handle) to memory on CPU or GPU, etc.

4.3.2 Output

The same applies to an output. It can be saved to a file or transferred to GPU memory for maximum performance advantage. Vulkan offers **interop** functionality for the vast majority of graphics APIs, which could save unnecessary copying. But again – it is highly dependent on an external system using this project.

Creating a universal **system for handling input and output** in as many ways as possible is a **long-term commitment** whose usefulness heavily depends on the number of external systems using this project. This algorithm was created primarily with just a single external system in mind.

4.3.3 Conclusion

The actual **integration into the FataMorgana** system is outside this project's scope. It was planned to be done shortly after this project was finished. However, these plans have now changed since I no longer work for Pocket Virtuality at the time of writing. Therefore, integration into FataMorgana and other works related to it will have to be **done by another company employee.**

4.4 Testing hardware

All development and testing were done primarily on desktop PC running Windows 11 and the following configuration:

- **AMD Ryzen 9 3900X**,
- 32 GB RAM (3600 MHz),
- **Nvidia GeForce RTX 3070 Ti** (8 GB of dedicated GPU memory).

All relevant software, such as OS, Vulkan SDK, and GPU drivers, were kept reasonably **up-to-date** during development.

4.4.1 AMD and Nvidia

Besides the aforementioned main testing Nvidia GPU, this project was also tested on a dedicated AMD graphics card. More specifically, **AMD Radeon RX 6600**. This testing was brief and done only to check changes in the **behavior of Vulkan drivers** from different GPU vendors.

As for the result, no differences between AMD and Nvidia graphics cards were encountered in program output. There was also **no difference in errors/warnings/debug messages** produced by AMD and Nvidia drivers.

4.4.2 Limitations of AMD cards

The AMD card used for testing is modern but very low-end (the cheapest previous generation AMD card available in shops at the time). This **caused 3 problems**.

Firstly, this application requires **1 queue** (capable of graphics, compute, and transfer operations) for **generating surfels** and **another one** (with graphics, transfer, and presentation capabilities) for **debug renderer**.

AMD Radeon RX 6600 supports a **single queue capable of graphics workload** (“`GRAPHICS_BIT`”). Therefore, using a debug renderer on this AMD card was impossible. For comparison, the Nvidia RTX 3070 Ti supports 5 different queue families of various capabilities, each containing up to 16 queues (28 queues in total across all families). Surprisingly, AMD card does not support more than **a single queue**.

The second issue was **the maximum number of views for multiview rendering** (a property called “`maxMultiviewViewCount`” in Vulkan Hardware Capability Viewer). While the tested Nvidia card supports up to **32 views** (plenty for sampling cameras), the AMD card supports a mere **6 views**.

Lastly, this AMD card does **not support 64-bit unsigned integers** (`VK_FORMAT_R64_UINT`) in any buffers, therefore not even in the vertex buffer, which is required by the application. However, even though standard Vulkan validation layers threw an error regarding the lack of support for this format, everything worked fine. One can only assume that the GPU driver must have used a **fallback behavior** for this case. Still, this should be avoided in general. Not only can it cause undefined behavior or crashes, but a fallback strategy will most likely cause reduced performance. If needed, this issue could be solved using **other 64-bit format** and **casting**. Radeon RX 6600 does support 64-bit signed floating-point numbers (`R64_SFLOAT`) for vertex buffers.

After more research, these limitations showed up to be more of an issue of AMD rather than low-end vs high-end cards. For example, the exactly same issues occur between competing cards of comparable price, Nvidia RTX 3050 and AMD RX 7600. While Nvidia offers **32 multiview views** and plenty of fully-equipped queues, AMD offers **only a fraction** of that.

Strangely, the same limitations occur in the best consumer-grade AMD cards of the current generation, such as RX 7900 XTX. Its maxMultiviewViewCount is only 6, and while it offers 9 queues in total, there is still only a **single fully equipped queue** available. It is also the only queue with essential graphics capabilities.

4.5 Encountered technical issues

Vulkan has a handy feature called **API layers**. These small programs intercept calls to Vulkan functions before they reach the GPU driver. Layers can be used for **profiling**, **validation**, and similar things. Vulkan SDK contains the **Vulkan Configurator** app, which provides many different layers – everything from validation of synchronization, thread safety, and object lifetime to hardware-specific performance suggestions.

Standard **validation layers** were turned on throughout the development to catch issues immediately. Unfortunately, some edge cases are undetected even with all available 1st party validation layers turned on. And with GPU programming being notorious for complex troubleshooting, resolving some issues took considerable time.

4.5.1 Device Lost error

One such issue was the hard crash of an application with nothing but a **"Device Lost"** error provided by Vulkan. This issue is extremely difficult to debug since it gives no additional information.

Khronos acknowledges this and recently added a way for GPU driver manufacturers to specify what caused the Device Lost error. This can be done through extensions `VK_EXT_device_address_binding_report` and

`VK_EXT_device_fault`. These extensions were not available before the problem was resolved.

Nvidia provides an SDK for **collecting GPU crash dumps**. Unfortunately, it is available only for C/C++. Therefore, other methods were tested before trying to call this SDK from C# code (which might be very difficult or even impossible due to the nature of this low-level SDK).

After some trial and error, it was discovered that this problem was most probably **caused by the Nvidia driver** (specifically, version 512.15). It also happened only on testing laptop but not desktop PC. Since newer drivers seemed to fix this issue, it was not investigated further.

4.5.2 Attempted to read or write protected memory error

Another issue, most probably caused by the faulty driver, resulted in throwing an **"Attempted to read or write protected memory"** error when trying to create a pipeline using the `vkCreateGraphicsPipelines` command. Validation layers reported no other errors, nor was there a more detailed description of why this error was thrown.

This issue was even more baffling since the call to this command was at the beginning of the whole program run - in a phase of setting up devices, pipelines, command buffers, etc. No actual work was being done on GPU yet. This error also seemed to depend on the number of **views for multi-view rendering** (internally `ImageArrayLayersCount` variable).

Fortunately, this issue occurred only in drivers 522.25 and 526.47. No other drivers produced this error message.

4.5.3 Gaps in results

The more serious issue, however, was causing **gaps in the resulting array of surfels**. Meaning that there were surfels with all parameters **set to 0** (default value) in the middle of the resulting array. This started to happen after several iterations of the algorithm (at least the first few thousand surfels in the output array were correct). The algorithm was internally designed the way this should not happen and was therefore considered a **faulty output**.

Investigation of this issue involved Nvidia drivers released in the last several months from when this issue was detected. Both Game Ready and Studio drivers and Beta versions of drivers were tested.

A weird behavior was detected - drivers from 512.15 up to 516.40 expressed the same problem with gaps in output, but driver version 516.59 caused differences in reading the same buffer from RenderDoc and directly from code (by copying this buffer to CPU memory where it could be read the standard way). Later driver versions did not have this issue but still produced **incorrect output**. All available drivers up to 516.94 were tested.

This all seems like undefined behavior, so the investigation was re-focused on a different approach. In the end, problems with incorrect output were caused by an **incorrectly set memory barrier** – something that is usually detected by validation layers, but this time, it was not.

5 Architecture

This chapter explains the architecture of the whole app. That includes:

- **organization** of the source code,
- **class hierarchy** (inheritance, base classes, interfaces, …),
- **functionality and usage** of most classes/structs and their properties/methods,
- and rationale behind **implementation decisions**.

5.1 Organization of solution, projects, and folders

The root folder of this project contains:

- **FMsurfelsDebugTools** folder
    - everything related debug tools project
    - for more info, see Chapter 6.9
- **FMsurfelsDirectX** folder
    - abandoned DirectX version of the main project and everything related
        - this project is not part of any solution anymore
- **FMsurfelsVulkan** folder
    - Vulkan version of the main project and everything related
- **Master Thesis** folder
    - this Master Thesis document and its assets
- **FMsurfels.sln** file
    - solution which contains projects `FMsurfelsVulkan` and `FMsurfelsDebugTools` projects
    - does **not** contain the `FMsurfelsDirectX` project anymore
- <miscellaneous files such as gitignore and Directory.Build.props>

All C# project files are in the roots of their respective folder trees. They are all **self-contained** – without cross references or dependencies.

FMsurfelsVulkan folder further contains `Program.cs` with an entry point (method `Main`), class diagrams (generated in Visual Studio; mainly to provide pictures for this document), and a hierarchy of folders containing the rest of the source code. With few exceptions, each class/struct has its **own .cs file** named after it.

The majority of folders are named after the **classes/interfaces** they contain. For example, the Buffer folder contains `BufferBase.cs` file (+ files with specific classes inheriting directly from it), Staged Storage Buffers subfolder (with `StagedStorageBufferBase.cs` inheriting from `BufferBase` class; + more classes inheriting now from `StagedStorageBufferBase`), and so on.

Most of this chapter is organized similarly, with subchapters and subtitles as the aforementioned folder hierarchy.

5.2 Main method

The entry point for the whole program is the `Main` method in the `Program.cs` file.

5.2.1 Options

The first thing the `Main` method does is the parsing of command line arguments from `string[] args` parameters.

`Options` class represents a collection of program parameters used for that. It uses the **Command Line Parser Library for CLR and NetStandard** installed from the NuGet package. This library provides an easy way to specify options, their default values, whether they are required, and other useful features using straightforward attributes. It then takes care of actual parsing in the `Main` method.

These options are used to specify:

- **input OBJ file**,
  - path to a file used as an input,
- **dimensions** of sampling viewports,
  - size of sampling camera viewport in **pixels**
  - in general, values between 512x512 and 2048x2048 seems to have an ideal quality to performance ratio with **1024x1024** being a default value
- **maximum number of surfels**,
  - **upper limit** of surfels to be generated

    - there is **no guarantee** that this many surfels will be generated since the algorithm will finish also when no more surfels can be extracted from the model
    - depends extremely on the model itself and the **requirements of the external system** using this algorithm
    - expected to be in orders of **thousands** up to **hundreds of thousands** for big models
    - a default value of **10 000** seems like a golden mean for debug purposes
- number of **views**,
    - number of **cameras** to use for the Sampling stage
    - limited by the value of `maxMultiviewViewCount` from `VkPhysicalDeviceMultiviewProperties` and `VkPhysicalDeviceLimits.MaxImageArrayLayers`
    - a default value is **16**, while the minimum should not be lower than **6** (to keep decent quality), and most GPUs nowadays are limited by `maxMultiviewViewCount` being **32** (see Chapter 4.4.2 for more details)
- and **debug options**
    - whether to use **surfel renderer**
    - file path for output **PLY file** with surfels
    - options related to **RenderDoc API**, such as file path for capture files.

None of these options are mandatory to specify in the command line argument since they all have default values.

5.2.2 Setup

After parsing, the `Main` method sets up instances of 2 important classes: `SurfelGenerator` and `VulkanAppBase`. The second mentioned one is actually an abstract class. The concrete selected implementation is based on the value of `Options.UseDebugRenderer`.

- If **true**, `SurfelRenderer` is used to show debug visualization of surfels.

- If **not**, `BasicVulkanApp` is created instead.

In either case, `SurfelGenerator` uses this created `VulkanAppBase` to get everything needed from Vulkan – `Instance`, `Device`/`PhysicalDevice`, `Queues`, `CommandPool`, `AllocationCallbacks`, …

Both `SurfelGenerator` and `VulkanAppBase` are initialized solely using options parsed from the command line (or their default value).

5.2.3 Run

The actual run of the algorithm from the perspective of the `Main` method is relatively simple. It calls `Run` methods on both `SurfelGenerator` and `VulkanAppBase`. These methods return `Tasks`, which are then waited for using `Task.WaitAll` method.

5.2.4 End

If `Options.OutputPlyFileName` property is specified, `SurfelGenerator` saves generated surfels into **PLY file** on disk.

After that, the only remaining thing to do before the program exits is proper **disposal** of instances of both `SurfelGenerator` and `VulkanApp`.

5.3 Vulkan Apps

At the core of this application is a hierarchy of **"Vulkan App"** classes. These classes were developed with the intention of providing a flexible **Vulkan backend** for the majority of applications. The whole hierarchy might seem too generic and broad. The reason is that such a system of classes could be used in **any other project** requiring Vulkan in the company.

Base class

As with most complex classes in this project, there is an abstract base class at the bottom of this hierarchy - `VulkanAppBase`. It takes care of all **core** things Vulkan requires, such as initialization of `Instance`, choosing `PhysicalDevice`, creating (logical) `Device`, `Queues`, `CommandPool`, etc. It also checks and enables desired device and instance extensions as well as layers. And, of course, it ensures all of this is appropriately disposed of (using an `IDisposable` interface). It is expected that many methods of `VulkanAppBase` will be overridden by classes that inherit from it; therefore, they are marked as **virtual**.

There is **no rendering** done in this class. There are also no calls to any other Vulkan commands. This adds flexibility since it allows the creation of various apps on top of this class – be it **standard rendering** or **non-standard** ones like this project.

This class also does **not take care** of creation or management of **windows** or even **swapchain**. No windows and swapchains mean no need for a present `Queue`. Therefore, this class can be used on platforms that do not support this type of `Queue`.

5.3.1 Render-less Vulkan App

The simplest implementation of the aforementioned base class is `BasicVulkanApp`. It provides a way to check for device features easily. This class is directly used by `SurfelGenerator` in case there is no need for a debug renderer.

Same as `VulkanAppBase`, this class still does not do any work after initialization. It belongs to the **render-less branch of Vulkan Apps**. Its `Run` method returns `Task.CompletedTask`. `BasicVulkanApp` is just used to provide all **core Vulkan objects ready and initialized** for whatever system above it needs them.

5.3.2 Renderers

There are **3 renderer classes** in total. 2 of which are abstract and 1 concrete. This might seem unnecessarily complex, but it creates a foundation for various renderers – both off-screen and on-screen/window renderers.

RendererBase

The **branch of rendering Vulkan Apps** starts with the `RendererBase` class. Like all Vulkan Apps, it too inherits from `VulkanAppBase`. On top of that, it takes care of initializing and providing **renderer-related Vulkan objects** such as `Viewport`, graphics `Pipeline` (with color blending, input assembly, rasterizer, …), `RenderPass` but also `DepthBuffer`, and `DescriptorPool`.

Unlike BasicVulkanApp, this class is actually expected to do something besides the initialization of Vulkan objects. Its Run method calls MainLoop, and it even provides the GetMainRenderCommands method, which calls Vulkan commands for binding of Pipeline, DescriptorSets, vertex/index buffers, and finally calls CmdDraw/CmdDrawIndexed.

WindowedRendererBase

WindowedRendererBase then extends RendererBase with properties and methods required for **managing the window** and its **surface**. Thanks to the Silk.NET.Windowing namespace, the managing of windows is **platform-independent**.

Besides inheriting from RendererBase, it also implements the IPresentQueueProvider interface (adds present Queue). This class does not take care of just surface-related Vulkan objects KhrSurface and SurfaceKHR, but also synchronization objects (Semaphores and Fences) used to signal which Framebuffer is being presented and which is being rendered to. This is needed since the WindowedRendererBase class supports advanced present modes (chosen by a class inheriting from this base class). In older graphics APIs, these present modes are referred to as **double/triple buffering**. Present modes in Vulkan are significantly more complicated – the important part is that they require multiple Framebuffers.

Lastly, WindowedRendererBase handles all work related to Swapchain. This includes its creation, disposal, and, very importantly, a re-creation of the swapchain in case of an error. Such an error might be, for example, KhrSwapchain.AcquireNextImage method returning Result.ErrorOutOfDateKhr, which usually indicates that the window was **resized**.

Note – Swapchain class mentioned in this part is not directly a Vulkan object but rather a custom class (for more details, see Chapter 5.3.3).

SurfelRenderer

SurfelRenderer class extends WindowedRendererBase with everything needed for **debug rendering of surfels**. This includes setting up correct shaders (Surfels.vert/.geom/.frag files), camera, WindowControl, and various parameters for surfels such as their radius and length of prefix of surfels array to draw.

These parameters for surfels and cameras must be **transferred from CPU to GPU**. Therefore, `SurfelRenderer` also sets up Uniform Buffers with descriptors for them (`SurfelParametersUbo` and `SimpleMatricesUbo`).

The non-trivial part of `SurfelRenderer` is a calculation of the length of the surfel array (`RenderableSurfelsBuffer`) prefix to render. The calculation is done through the `ChangeSurfelRadius` method (`ChangeSurfelRadiusCallback` in case of a call from `WindowControl`). Calculations themselves are based on Chapter 4.1 from (Brandt, et al., 2019). `SurfelGenerator` itself has to call the `SetExampleMedianMinimumDistance` method to set the value of the `exampleMedianMinimumDistance` variable based on minimal distances between surfels. This is done only once at least `<examplePrefixLength>` surfels are inside of `RenderableSurfelsBuffer`.

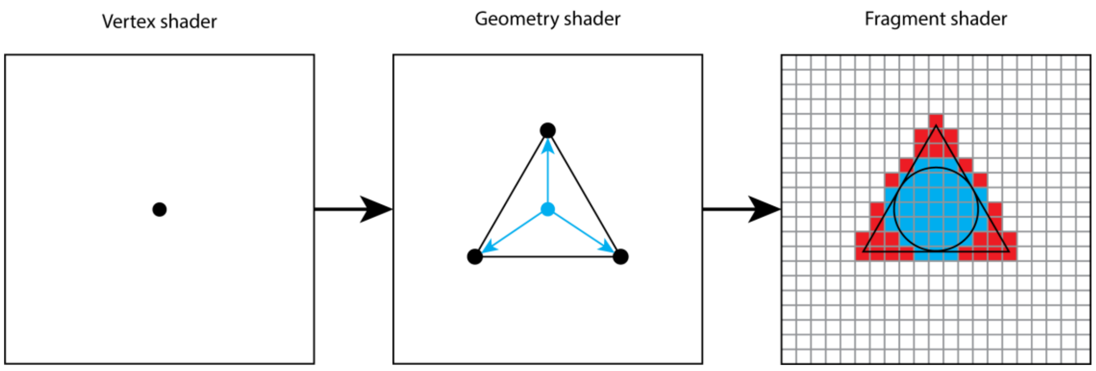


Figure 17 Rendering of surfel from surfel sample. A surfel sample (a single point) is passed from the vertex buffer to the vertex shader, the geometry shader creates an equilateral triangle out of it, and the fragment shader discards fragments outside of its inscribed circle (red fragments). Blue fragments represent the final surfel drawn as a circle.

When these calculations are done, `RenderableSurfelsBuffer` is set as a **vertex buffer**, and the appropriate amount of surfels from it is rendered. Vertex shader does nothing but transfer **color**, **normal**, and **world position** (from `RenderableSurfelsBuffer` acting as a vertex buffer) to geometry shader. It then draws an **equilateral triangle** whose inscribed circle is a surfel - color, normal, and world position are taken from the vertex buffer, while the radius is taken from `SurfelParametersUbo` (it is a constant for all surfels in a single draw call). The fragment shader subsequently **discards** fragments of the triangle that are not part of its inscribed circle and sets simple Lambertian shading for easier visual inspection of debug visualization. The whole process can be seen in Figure 17.

A similar process is used for the **rasterization of disks** to create a Voronoi diagram and a Poisson disk conflict removal, as mentioned in Chapter 3.2.5.

`SurfelRenderer` is used mainly as a **debug renderer** and is not expected to be used directly by an external program using this application.

5.3.3 Vulkan Apps Miscellaneous

This subchapter explains the **purpose** of some **miscellaneous classes** directly related to Vulkan Apps.

Swapchain

Even though it might sound like that, this is not directly a Vulkan object. `Swapchain` class does include instances of both Vulkan `KhrSwapchain` and `SwapchainKHR` but also everything else related to swapchain, such as `Images` themselves (with `ImageViews`, dimensions/`Extent2D`, and their `Format`) and also `Framebuffers`.

This class creates all of these Vulkan objects and properly disposes them. It also provides a way to acquire the next image and queue it for presentation. All including error handling.

The `Swapchain` class is directly used only by the `WindowedRendererBase` class.

VulkanException

This is a bare class inheriting from `System.Exception`. The vast majority of internal code throws this exception instead of `System` exceptions. It is helpful for future expansion, such as logging errors. Note – almost all error throws in this application are meant to be **fatal** for the application and should result in immediate program termination or **hard reset** at best.

QueueFamilyIndices

A struct called `QueueFamilyIndices` holds indices to the `Queue` family to use for all **graphics**, **present**, and **compute** `Queues`. This is directly used by mostly `VulkanAppBase` and `Swapchain`. Indices held by this struct correspond to an array of `QueueFamilyProperties` returned from the `GetPhysicalDeviceQueueFamilyProperties` Vulkan method.

Besides holding these indices, it also provides methods for checking whether all indices are set for a particular case:

- **Normal** case when all **3 families** are required,
- or **windowless** case when the **present** `Queue` family is not needed.

WindowControl

The purpose of the `WindowControl` class is to handle callbacks from **keyboard/mouse** button presses and **cursor** movement to translate/zoom camera. This results in the possibility of moving in the debug window as in any 3D program. It also provides a way to adjust the **size of displayed surfels**.

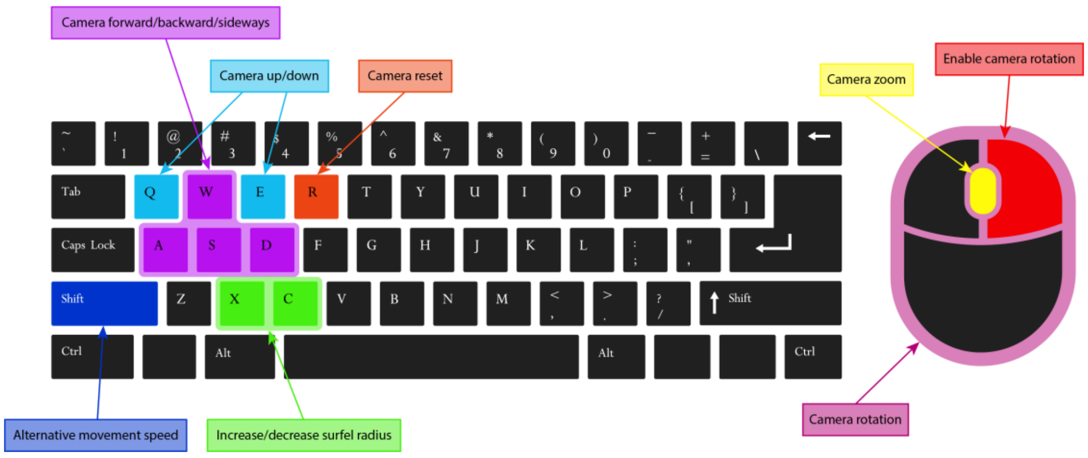

Camera forward/backward/sideways
Camera up/down
Camera reset
Camera zoom
Enable camera rotation
Alternative movement speed
Increase/decrease surfel radius
Camera rotation

Figure 18 Key bindings for debug viewer. Source of vector art: Vecteezy.[13]

Controls are currently hardcoded to the industry standard of **WASD** for forward/backward/sideways movement and **Q** and **E** for camera up/down movement. Additionally, the **X** and **C** keys are used to increase/decrease the surfel radius, and the reset of the camera to its default position is done using the **R** key.

Shift key is then used to speed up all translations. Holding down the **right mouse button** and subsequent cursor movement rotates the camera around the pivot. The **mouse scroll wheel**'s functionality is used to translate the camera alongside its forward vector (primitive "zoom").

The whole key binding scheme is illustrated in Figure 12.

[13] https://www.vecteezy.com/vector-art/4931862-keyboards-computer-with-black-and-white-style-vector-illustration and https://www.vecteezy.com/vector-art/9866961-mouse-icon-vector-mouse-icon-vector-illustration

There are **known issues** with this system in **extreme cases**. However, it works well enough for its purpose – simple camera movement in the debug window. Therefore, there is little to no incentive to improve these window controls furthermore. This also applies to **customizable bindings** for keys/mouse actions, movement speeds, etc.

All **callbacks** are currently handled through the `GlfwWindowing` class from Silk.NET. This is a windowing system selected for modern Windows OS. Unfortunately, Silk.NET **does not provide a platform-independent way** to handle these callbacks. And since all development is done on Windows 10/11, this one is used and hardcoded in the `WindowControl` class. In case it is required, it will not be too difficult to add support for the `SdlWindowing` system as well.

This class is directly used only by the `SurfelRenderer` class.

5.4 Stages

Stages are the most essential part of the CPU code of this project. They take care of creating and managing `Framebuffer`, `RenderPasses`, `FramebufferAttachment`, `Pipeline`, `CommandBuffer`, and synchronization objects for one part of the algorithm. The main program loop is directly calling `Draw`/`Dispatch` methods on stages to submit their `CommandBuffers` into the `Queue` – this is the end of all work in CPU code prior to work in shaders on GPU.

The base class for all stages, `StageBase`, is used mainly to create the skeleton for its implementations and a few standard methods, such as calls to dispose of Vulkan objects.

There are 2 very distinct types of stages:

- **compute stages**, calling compute shaders,
- and **G-buffer stages**, used for rendering into G-buffers (off-screen render targets).

5.4.1 Compute stages

Base class

`ComputeStageBase` is an abstract class for all stages used to call **compute shaders**. That is why it contains compute-shader-specific variables such as sizes of workgroups.

Besides that, it can directly **create a shader stage** (an instance of `PipelineShaderStageCreateInfo`; not related to “stages” in terms of this project) since there is always only a **single shader** in a compute pipeline, unlike a graphics pipeline, which can contain a varying number of shaders.

Before main loop

One of the simpler compute stages is `ChooseFirstSampleStage`, which needs mainly texture with sampled world positions, and `SurfelSamplesBuffer`, where it writes the very first sample. Its shader is relatively simple but with a small caveat mentioned in Chapter 6.3.

Inside of main loop

`ExtractFarthestSamplesStage` is the first compute stage to be called right after G-buffer `VoronoiStage` inside the main program loop. For each view, it creates **1 new sample** (written into `SurfelSamplesBuffer`) from the **farthest pixel** in each Voronoi cell.

Right after G-buffer `PoissonStage`, `RemoveCloseSamplesStage` is called. Its purpose is simple – **invalidate new samples** (in `SurfelSamplesBuffer`) that are too close to each other. This is done by going through all new samples created in the last iteration of `ExtractFarthestSamplesStage` and checking whether the pixel they were created from is covered by the Poisson disk of another sample. In case it is, their sample in `SurfelSamplesBuffer` is invalidated by writing 0 in its place (more specifically 64-bit representation of 0).

The next compute stage, `CompactAndSortStage`, is the most complex one. This stage actually consists of **6 mandatory phases** (`LocalSort`, `ComputeHistogramAndOffsets`, `ScanHistogram`, `ScanBlock`, and `Scatter`) and **1 optional phase** (`TestElementsOrder`) used only for debugging and validation purposes. All these phases run one after another. They are all contained in a single `CompactAndSort.comp` shader (for discussion on why it is in a single shader and not multiple ones, see Chapter 6.2.1). Algorithms in this shader are based on (Satish, et al., 2009). In short, it **fixes gaps of invalid samples** created by the previous stage and then **sorts the new samples** by decreasing the radius (to achieve progressiveness of samples). Sorting is done using parallel radix sort since it is ideal for sorting by only the 32 most significant bits representing radius in the surfel sample and not the whole 64-bit number. Plus, parallel radix sort can be effectively implemented in a compute shader.

And lastly, a simple `CountNewSamplesStage` is called. As the name implies, its shader **counts a number of new valid samples** added in this algorithm iteration after removing close samples, sorting, and compacting them in `SurfelSamplesBuffer`.

Others

The only remaining compute stage to mention is `ConvertSurfelsStage`. It is one of the simpler compute stages as its task is to **convert surfels** represented in the form of `SurfelSample` (data and indices to data in textures, all packed in a single 64-bit number) into the form of `RenderableSurfel` (data ready for straightforward rendering). These representations of surfels are discussed in more detail in Chapter 5.10.1. `ConvertSurfelsStage` is the **most flexible** stage regarding the order of execution in code. Based on the specific needs of the system using this algorithm, `ConvertSurfelsStage` can be called only at the very end of the main algorithm or, for example, each nth iteration. In other words, it is expected to be called only when the outside system requires the actual output from this algorithm – be it partial or final output.

5.4.2 G-buffers stages

The standard **forward rendering** pipeline consists of 1 framebuffer and 1 depth buffer. On the other hand, this algorithm uses several framebuffers as outputs from a single stage and reuses them as input to the following stages. This approach is more similar to **deferred rendering**, which uses several G-buffers to capture various information about the scene at a given pixel (diffuse color, specular color, normal vector, …). That is why the term **"G-buffer"** is also used in this algorithm, and it refers to render target with arbitrary information (not just the color) for pixels.

Framebuffer attachment

In the most general view, **attachments in Vulkan** refer to images with additional information on how to use them as input or (more commonly) output. Output attachments are equivalent to render targets – images containing color (color buffer), depth/stencil (depth buffer), or any other relevant information for the pixel we write into in fragment shader.

To make work with attachments easier, `FramebufferAttachment` class was created. It is used to hold a reference to `Image` itself (together with `ImageMemory` and `ImageView`) but also other relevant information such as `ClearValue`, attachment-specific structs `AttachmentDescription` (information on sampling used, load/store operation, layout, and others) and `AttachmentReference`, and others.

Base class

Abstract `GBuffersStageBase` class holds information and functions used by all G-buffers stages (they inherit from it). This includes (but is not limited to) the creation of `RenderPass`, `Framebuffer` objects (out of `FramebufferAttachments`, which it also provides a way to create), `Pipeline`, `CommandBuffer`, and other objects required for a single pass of rendering using rasterization to multiple off-screen render targets (G-buffers). Each `FramebufferAttachment` (more specifically, the `Image` this `FramebufferAttachment` refers to) is created with a number of layers corresponding to a number of cameras. Subsequently, `RenderPass` is created with multiview rendering enabled (`VK_KHR_multiview`) to utilize the capabilities of modern hardware (see Chapter 6.13 for more details about layered rendering).

Before main loop

There is only a single G-buffers stage before the start of the main program loop: `SamplingStage`. It is technically the same as the first pass of deferred rendering – it renders the scene (from several cameras at once) into several off-screen G-buffers (layers of their `Images`). This is depicted in Figure 13. In this case, these G-buffers contain:

- **color**,
- **normal vector**,
- and the **absolute world position** for each pixel.

Its shaders are similarly straightforward. `SamplingStage` is the **only stage that rasterizes a scene** from its polygonal representation. This fact makes the algorithm even more versatile since there is **no need for representation of the scene** as long as we have the aforementioned G-buffers as inputs (for more details, see Chapter 6.6).

Inside of main loop

The very first stage called in the main loop is `VoronoiStage`. Its job is to **assign each pixel** of the “rendered scene” to the **Voronoi cell** created from surfel samples (the sample acts as a Voronoi site). It does not directly use rendered scene but only the texture of absolute world positions retrieved from the `SamplingStage` (or another system able to provide it). Internally, it works on surfel samples in packed form (as 64-bit floats) bound as its vertex buffer. These samples are unpacked to get radius and texture coordinates, which are used to sample texture with world positions. With this information, the geometry shader creates an equilateral triangle from each sample so that its inscribed circle has a center and radius based on data passed from the **vertex shader**. The fragment shader is then used to discard all fragments outside of the aforementioned inscribed circle (see Figure 16). Besides that, the fragment shader also **calculates the distance of a fragment to its Voronoi site** and stores it in a G-buffer, together with the unique ID of the Voronoi cell. **Distance** to the Voronoi site also has one important role – it is assigned as the **fragment’s depth** (`gl_FragDepth` system variable). This technique assures that the **fragment is assigned to its closest Voronoi site**, therefore creating a valid (discrete) Voronoi diagram. Such a diagram can be seen on the left-most and right-most pictures of Figure 14.

Right between `ExtractFarthestSamplesStage` and `RemoveCloseSamplesStage` is the last G-buffers stage called `PoissonStage`. It works only on newly added surfel samples instead of all of them like the previous G-buffers stage. Its CPU-side code is very similar to `VoronoiStage`, and therefore, it inherits from it. Besides overriding a few properties (depth buffer clear value and comparison operation), it changes `FramebufferAttachments`. More specifically, it uses just 1 `FramebufferAttachment` – to output the **ID of the Poisson disk** for each pixel. It even uses exactly the **same vertex and geometry shaders**. Only the fragment shader is slightly different. Instead of writing the distance to the site into the depth buffer, it writes the radii of the new surfel sample (equal to the distance to their Voronoi site) in there. This results in **larger Poisson disks virtually overdrawing smaller ones** (see the third picture in Figure 14), which affects ID written into the render target for a specific fragment.

`RemoveCloseSamplesStage` later uses these IDs to determine which new samples should be removed (invalidated) and which should be kept.

5.5 Surfel Generator

The heart of the whole algorithm is a class called `SurfelGenerator`. As the name suggests, it is the place where **surfels are generated**. This is done by running stages (calling either `Draw` or `Dispatch` methods) in the correct order and with the proper parameters.

Before that, `SurfelGenerator` has to create and initialize all these stages, as well as all supporting objects such as buffers, sampling cameras, rasterizer, descriptions, etc. To do this, it utilizes `VulkanAppBase` passed as an argument into `SurfelGenerator.Initialize` method.

`SurfelGenerator` also provides various supporting structs for the creation of `VulkanAppBase`. These include `ApplicationInfo`, `PhysicalDeviceFeatures`, validation layer names, instance extensions, and others.

5.5.1 Main loop

After the initialization of `SurfelGenerator`, the outside class (in this case, `Program.Main`) just has to call the `async Task Run()` method, which starts the `MainLoop` method.

This method first calls `SamplingStage` to populate G-buffers for positions, normals, and colors and then `ChooseFirstSampleStage` to select the first sample, which is used as a base for `VoronoiStage`. This is done **before the actual loop**.

After this, the `while` loop begins – until a **number of samples is larger than the `MaxSurfelsCount`** parameter or **no new samples** can be created in an iteration of the loop.

One iteration of the loop starts with a call to `VoronoiStage`, followed by a call to dispatch of `ExtractFarthestSamplesStage` for each view, `PoissonStage`, and `RemoveCloseSamplesStage`. In this order, the only remaining mandatory stages to call are `CompactAndSortStage` and `CountNewSamplesStage`. The last mentioned is needed to obtain both the total number of samples and the number of new samples from this iteration (both are used to check whether the loop should continue with more iterations).

The call to `ConvertSurfelsStage` can be made at the end of the while loop whenever we want new surfels to display. This does not have to be every iteration of the loop and will heavily depend on the actual final usage of the program.

5.5.2 Synchronization between stages

There is a call to `QueueWaitIdle` between each and every call to a stage. Since each subsequent stage is entirely dependent on the end of work of the previous stage (and only a few, if any, pipeline stages can be run in parallel), this was found to be the **easiest** and the **least problematic solution**.

While more sophisticated approaches using `Semaphores`/`Fences` exist, they are significantly **more challenging** to set up correctly. Both `GBuffersStageBase.Draw` and `ComputeStageBase.Dispatch` methods fully support the use of these synchronization objects. Currently, there are unused parameters with default `null` values for all parts of `SubmitInfo` (various `Semaphores`, wait stages, `Fences`, …).

This could be one of the potential improvements in this program. However, it is expected that performance gain from this approach would **not be significant**.

It is important to note that the approach with `QueueWaitIdle` is also **more flexible** since it does not depend on stages, it is in between. The same cannot be said about other mentioned synchronization objects.

5.5.3 Graphics debuggers

Each iteration of the loop (and the short part before the loop) calls RenderDoc API to set the **start and end of the frame** through `RenderDocHelperFunctions`. This is needed since there are **no standard “frames”** in this algorithm (most graphics debuggers consider the end of the frame to be a call to a “present” method or similar). But 1 iteration of this loop was found to be ideal for “frame” in terms of RenderDoc.

5.6 Buffers

Vulkan already contains a class called `Buffer`. However, it is used only as an **opaque handle** without any properties/methods. Buffers in computer graphics are used for a vast variety of features/algorithms and come in many shapes, forms, and ways of usage. Therefore, it was necessary to create a hierarchy of classes that primarily encapsulate the `Buffer` handle and everything else related to it, such as `DeviceMemory` and various flags (`BufferUsageFlags` and `MemoryPropertyFlags`), but also provide methods for easier work.

DepthBuffer

Even though the name might suggest it, the class `DepthBuffer` **does not** contain a Vulkan `Buffer` object but rather an `Image` (among others). Therefore, it does not share any functionality or predecessors (interfaces or base classes) with the rest of the classes with "buffer" in its name. Since the name "depth buffer" is well established in computer graphics and `Image` class can be considered a 2D `Buffer` with additional functionality such as sampling, it should keep this name.

Regarding the purpose of this class, it is a wrapper around stuff related to depth buffering/Z-buffering. This includes, most importantly, `Image` itself, `CompareOp`, methods to find supported formats, and a straightforward way to create structs used for `FramebufferAttachments` (`AttachmentDescription` and `AttachmentReference`) and the creation of pipeline (`PipelineDepthStencilStateCreateInfo`).

Base class

`BufferBase` class serves as a base class for all other buffer classes (except for the aforementioned `DepthBuffer`). It encapsulates the Vulkan `Buffer` class together with `DeviceMemory`, flags, and getters used to retrieve the total size of the buffer and the number of elements in it. An element is a class set as a type `T` since `BufferBase<T>` is a generic class.

This class also takes care of the disposal of allocated Vulkan classes using a standard `IDisposable` interface.

StagingBuffer

Staging buffers, in general, are used as a **middleman between memory** on the **CPU** (*host visible*) and `Buffers` completely stored on the **GPU** (*device local*).

Of course, this does not apply to all architectures, but in general (especially on standard desktop computers), it provides a significant performance benefit.

Data is first copied **from CPU** memory (an array, for example) **to the staging buffer**, and then the staging buffer is copied **to the final device local buffer**. This technique is extensively used throughout this application to get as much performance as possible. `VulkanHelperFunctions` provide the `UpdateBufferUsingStagingBuffer` method to do exactly this.

The `StagingBuffer` class inherits from `BufferBase` and overrides default `BufferUsageFlags` to ones used by all staging buffers: `TransferSrcBit` and `TransferDstBit` to make work with staging buffers easier.

This class also restricts generic type `T` to `unmanaged`. This is required since `System.Buffer.MemoryCopy` is used to copy data from CPU-only memory to the staging buffer, and it requires a **pointer to data** (using a `fixed` keyword). Until C# 11, trying to get a pointer to a managed type resulted in an error. In C# 11, doing so will result in only a warning (CS8500) but still should not be used to avoid any issues. Besides, `T` being `unmanaged` is not a huge restriction in this case. Having a string, array, or other collection as a single element in a buffer makes very little sense. Custom structs with all fields being `unmanaged` are still considered unmanaged themselves, which replaces the need for having a custom class as an element of a buffer.

`StagingBuffer` is primarily used by **Staged Storage Buffers** (see Chapter 5.6.3) but can also be used outside of them.

5.6.1 Interfaces

There are 2 main buffer-related interfaces: `IDataBuffer` and `IIndexBuffer`. They serve as a set of minimal requirements for specific cases.

`IDataBuffer` requires just a `Buffer` object and the number of elements in it. This is, for example, all that the renderer needs to bind the vertex buffer (`CmdBindVertexBuffers`) and draw it (`CmdDraw`).

`IIndexBuffer` inherits from `IDataBuffer` and adds a requirement to declare `IndexType`. A renderer then uses this to bind it (`CmdBindIndexBuffer`).

5.6.2 Uniform Buffers

Uniform Buffers (frequently referred to as UBO = Uniform Buffer Object) are buffers that are relatively **fast to access** (in general, faster than Storage Buffers but slower than Push Constants), but they are **read-only** for a shader and **cannot hold much data** (usually from 16 to 64 kB on the most GPUs; still significantly more than Push Constants).

Most of the Uniform Buffers in this project are small enough to be converted to Push Constants to get even slightly faster access. However, changing the Push Constant's value **requires re-recording** the whole `CommandBuffer`, which is not needed in the case of Uniform Buffer. Therefore, overall performance gain would be negligible. Also, working with Uniform Buffers is more comfortable and easier than Push Constants.

Base class

`UboBase` abstract class serves as a base class for all UBO classes. It provides `DescriptorType`, `DescriptorBufferInfo`, and `BufferUsageFlags`, as well as methods for updating data in these buffers using a `CommandBuffer`. It inherits from a `BufferBase` class.

Specific UBO classes

Most final UBO classes are relatively simple – they just declare structs used inside and set it as their type `T` since the `UboBase<T>` class is generic.

Structs can range from simple `ComputeUbo` (containing `viewIndex` used in several compute shaders) and `SurfelParametersUbo` (holding surfel radius for `SurfelRenderer`) through `SortParametersUbo` (with several variables used in `CompactAndSort` shader) up to `SimpleMatricesUbo` and `MatricesUbo`. The last 2 mentioned contain structs with standard 4x4 `model`, `view`, `projection`, and `modelViewProjection` matrices used by various shaders. Compared to `SimpleMatricesUbo`, `MatricesUbo` has arrays of all of these matrices (except for the `model` matrix) – 1 set of matrices in arrays (at the same index) for each camera view.

5.6.3 Staged Storage Buffers

Storage Buffers in Vulkan are the **most flexible buffers**. They can be both **read and written** to in shaders, and they can hold a **huge amount of data**. As a limitation, they are rather **slow** to access.

The word "staged" in the full name of these buffers refers to the fact that they contain an instance of the `StagingBuffer` class to improve overall performance on most architectures while still allowing writing and reading from these buffers on the CPU side.

Base classes

Abstract `StagedStorageBufferBase` class inherits from `BufferBase` (and also `IDataBuffer` interface). On top of that, it adds an instance of `StagingBuffer<T>` with everything related to it: the creation of this buffer, the update of the main buffer using this staging buffer, and its disposal. The `StagedStorageBufferBase` constructor also requires a `Queue` instance to copy data from the staging buffer to the main buffer.

There is one additional abstract class between `StagedStorageBufferBase` and some of the final classes: `ReadableDataBufferBase`. As the name suggests, it adds functionality to read data on the CPU from the main buffer using the staging buffer through the `T[] GetData()` method. The `StagedStorageBufferBase` class itself does not offer this functionality because it requires an additional usage flag for the main buffer (`BufferUsageFlags.TransferSrcBit`) and could cause unnecessary performance hit.

CompactAndSort buffers

Similarly to Uniform Buffers, many Staged Storage Buffer classes just **declare the structs** they contain and add **overloads of methods** for easier creation, update, and data retrieval based on their structs.

Examples of such simple classes are ones used for `CompactAndSort` shader: `BoundsBuffer`, `HistogramBlockSumBuffer`, `HistogramBuffer`, `SamplesCountsBuffer`, and `OrderedTestBuffer`.

Surfel buffers

More complex buffers in the same category are related directly to surfels.

The first one is `RenderableSurfelsBuffer`, which is used for **debug visualization** in `SurfelRenderer`. Compared to other buffer classes, it adds functionality to **export data** to PLY format for debug purposes, and its usage flags are extended by `BufferUsageFlags.VertexBufferBit`.

The second surfel-related buffer is probably the most important buffer in the whole application – `SurfelSamplesBuffer`. As an addition, it provides methods for getting **testing data** (both precalculated/deterministic and random). And same as `RenderableSurfelsBuffer`, it is used as a vertex shader, which is reflected in its usage flags.

Structs for both of these buffers (`RenderableSurfel` and `SurfelSample`) are explained in detail in Chapter 5.10.1.

Rasterization buffers

The last 2 remaining Staged Storage Buffers are simply called just `IndexBuffer` and `VertexBuffer`. As names suggest, they are directly used for rasterization in the `SamplingStage`.

Since Vulkan's `IndexType` does not provide much flexibility (it technically supports only 16 or 32-bit unsigned integers for standard rasterization), `IndexBuffer` straightaway sets its generic type parameter `T` to 32-bit `uint`. A change to 16-bit `ushort` won't be problematic in case this performance/memory advantage is required. But ~65k indices (accessible through `ushort`) is rarely enough. Therefore, it was deemed unnecessary even to consider this option.

To enable as broad support for vertex formats as possible, `VertexBuffer` uses `byte` as its generic type parameter `T`. This moves the responsibility of providing correct vertex format to `IMesh` with its `byte[] GetVertexBufferArray()` method (for more info, see Chapters 5.10.2 and 5.11) and subsequently the selection of corresponding shader.

Both of these buffers also set appropriate usage flags: `BufferUsageFlags.IndexBufferBit` and `VertexBufferBit`, respectively.

5.7 Constants

Vulkan uses the concept of **Specialization Constants** to change the values of constants in **halfway-compiled** SPIR-V shaders before they are bound to the pipeline stage, as seen in Figure 13. This is a great way to easily adjust simple values in shaders with virtually no performance overhead – unlike Push Constants and various buffers.

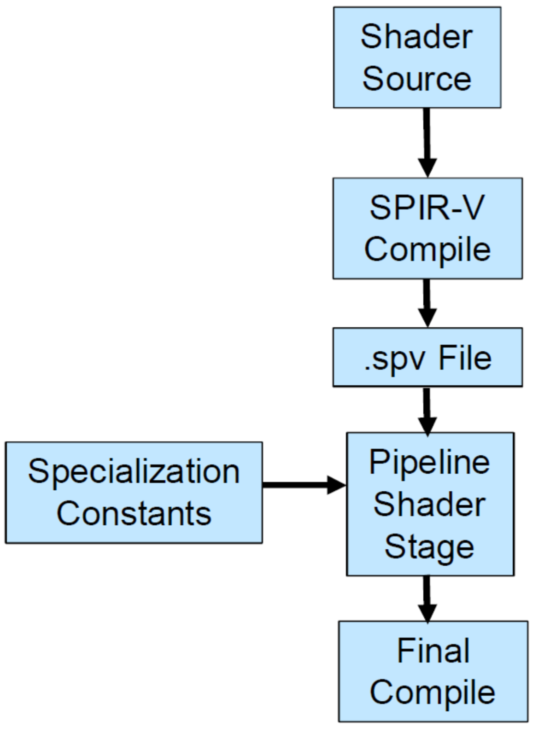


Figure 19 Illustration of how specialization constants are set in the shader compilation pipeline. Source: Vulkan Specialization Constants presentation by Mike Bailey from Oregon State University.[14]

This technique is used in `DimensionsConstants.inc` file, which is included in several shaders. It provides constants for a maximum number of views and dimensions of the viewport.

To make use of Specialization Constants easier, several classes were developed. An `IShaderConstants` interface is at the bottom of this hierarchy, with broader `ShaderConstantsBase` inheriting this interface. Together, they provide the final class with as many automatically calculated values as possible. For example, `SpecializationMapEntry[]` can be automatically created thanks to C# reflection. The final class then only needs to set actual data struct containing constants and provide an implementation for a method to get

[14] https://web.engr.oregonstate.edu/~mjb/vulkan/Handouts/SpecializationConstants.4pp.pdf

`SpecializationInfo` – that is the only struct required by Vulkan itself for the creation of the pipeline shader stage.

5.8 Descriptors

The concept of descriptors (descriptor sets, layouts, pools, …) in Vulkan is somewhat convoluted and complex. The descriptor is usually either an **image** or a **buffer** with additional information. For an image, this additional information (besides `ImageView` itself) would be `ImageLayout` and `Sampler`. And for a buffer, the descriptor contains its size and offset.

Descriptors are always grouped into **descriptor sets**.

Descriptor set layouts provide a description for these sets – saying what type of descriptor is at which binding index and which shader stages it is used in. This layout is bound to the **pipeline**.

Later, a descriptor set (containing references to actual buffers and images) is used in a **call to command buffer**. Of course, any descriptor set that fits the descriptor set layout can be used with that pipeline. This is illustrated in Figure 20.

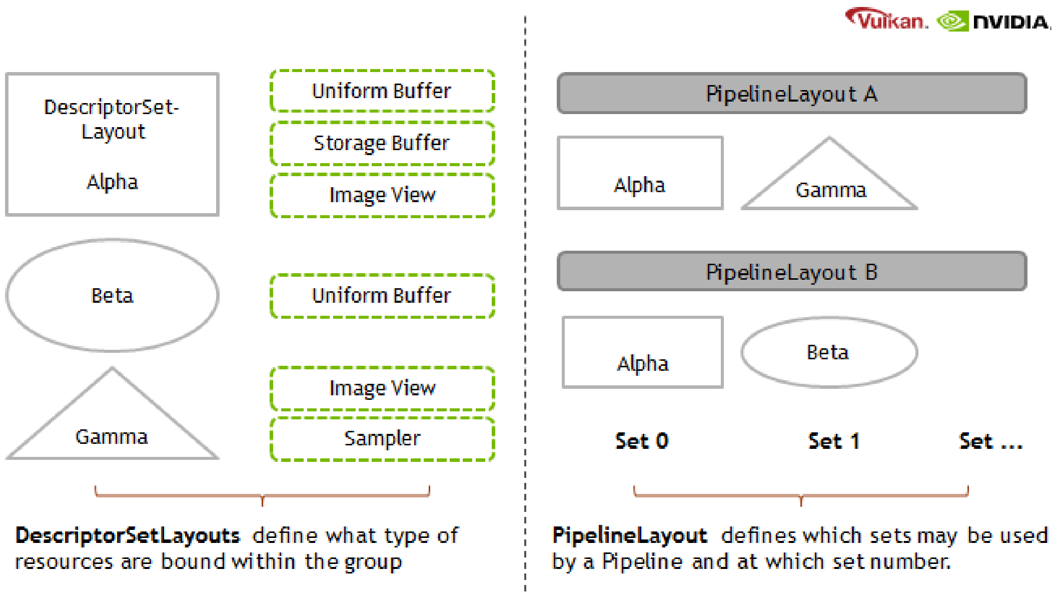


Figure 20 Visualization of relations between descriptors, descriptors sets, descriptor set layouts, and descript pool. Source: Article “Vulkan Shader Resource Binding” on Nvidia Developer website[15]

[15] https://developer.nvidia.com/vulkan-shader-resource-binding

5.8.1 Descriptors Manager

`DescriptorsManager` with several helper classes was implemented with the sole purpose of making the use of descriptors more comfortable. This class creates `DescriptorSetLayout`, `DescriptorPool` (used to allocate descriptors), and `DescriptorSet`. Finally, it updates the `DescriptorSet` using `WriteDescriptorSet` (makes sure descriptor objects point to actual data such as `Buffers` or `Images`).

This solution to wrap the whole concept of descriptors **lacks flexibility** in some cases (such as creating multiple descriptor sets for a single layout or more efficient use of a descriptor pool). But it is **easy to use** (compared to raw handling of sets, layouts, etc.) and should be adequate for **most cases**. Most importantly, it is more than enough for use in this project.

5.8.2 Descriptor Set Elements

To actually do all this work, `DescriptorsManager` needs info about descriptors to use in the first place. This is provided as `IEnumerable<IDescriptorSetElement>`. In this case, "descriptor set element" refers to a single "descriptor" (or "descriptor object" as called in some literature). On top of that is the base class `DescriptorSetElementBase`, which implements methods for getting `WriteDescriptorSet` and `DescriptorSetLayoutBinding`.

The most interesting classes for the final implementor are `BufferDescriptorSetElement` and `SamplerDescriptorSetElement`. As names imply, they are descriptors for buffers and samplers. In Vulkan, it is technically possible to have a descriptor only for `Sampler` or image (`ImageView` with `ImageLayout`). But the most common case is to use `DescriptorType.CombinedImageSampler`, which combines the sampler and image into one descriptor. It is also the easiest way to deal with it.

One more class deriving from `IDescriptorSetElement` is `UboBase` (for details about his class, see Chapter 5.6.2). It is a particular case since it does not inherit from `DescriptorSetElementBase`. Instead, it internally holds a reference to `BufferDescriptorSetElement` – that is also how it implements the `IDescriptorSetElement` interface.

Constructors of all these final classes are as simple as possible. Just provide the necessary info for descriptors, and everything else is deduced from the ordering of `IEnumerable<IDescriptorSetElement>` that goes to the constructor of the `DescriptorsManager` class.

5.9 Cameras

Camera classes (deriving from the `ICamera` interface) have 2 tasks: calculate and provide **projection** and **view** matrices.

Extrinsics/View matrix

The calculation of the view matrix is based on extrinsics of the camera - position, direction/target vector, and orientation (represented by an up vector in this case). Since extrinsics are not dependent on the type of camera, they are handled in the abstract `CameraBase` class through `SetExtrinsics` methods.

Intrinsics/Projection matrix

Intrinsics (required for projection matrix) depend on camera type and, therefore, are handled by the final classes – `CameraOrthographic` and `CameraPerspective`. Respectively, they are set using `SetIntrinsics` methods.

A **perspective** camera is used for debug viewer since that is how **human vision** works and is the most common way to display 3D data naturally.

On the other hand, an **orthographic camera** is an ideal candidate for the **sampling process**. It projects parts of a 3D scene the same, whether in the center of the camera view or on the edge. This type of camera also **ignores the distance** of parts of the scene from the camera. The only thing that matters is whether the scene is in front of the camera or behind it. Due to the nature of the placement of cameras (see Chapter 6.1), the whole model is always in front of each orthographic camera used for sampling. The width and height of these cameras are also set to always capture the entire model.

For a **comparison** of perspective and orthographic projections, see Figure 15.

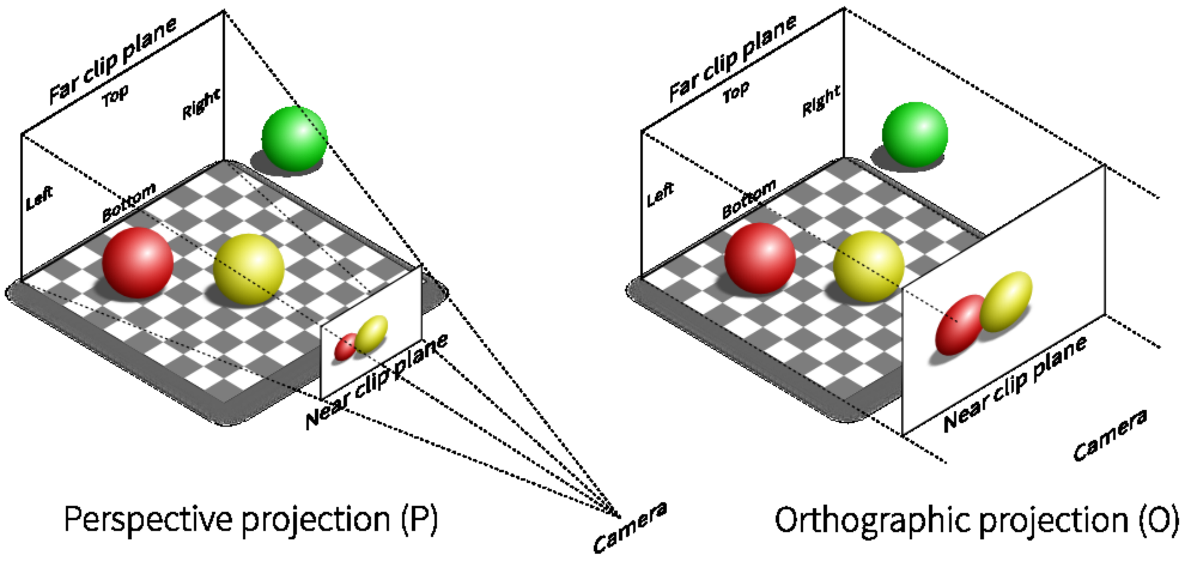


Figure 21 An illustration of the difference between perspective and orthographic projections. Source: StackOverflow question "From perspective picture to orthographic picture" by Raph Schim[16]

5.10 Elements

Structs and interfaces in `FMsurfelsVulkan.Elements` namespace are used directly in vertex buffers. To make serialization into buffers easier:

- they are actually **C# structs** and not classes,
- all data is in the form of **fields** (and properties are used only as getters into these fields), and
- `FormatAttribute` is used to mark these fields with `Silk.NET.Vulkan.Format`.

They consist of various vertices for sampled mesh and 2 structs used for surfels.

5.10.1 Surfels

Two structs for the representation of surfels are needed since they have significantly **different internal representations** and are used for different tasks. They also do not share any interface, even though naming might suggest so.

[16] https://stackoverflow.com/questions/36573283/from-perspective-picture-to-orthographic-picture

`RenderableSurfel` is the more straightforward of these two. It is used for rendering into a **debug viewer**. That is why its simple internal structure is represented by just **position**, **normal vector**, and **color**.

The significantly more important and more complex one is the `SurfelSample`. Internally, it is represented as a single **ulong** (64-bit unsigned integer) containing packed data for **UV texture coordinates**, **texture layer index**, and **radius**. The exact format is (U, V, layer, radius) with **13:13:6:32** bits, respectively. In this packed form, texture coordinates and layer index are `uint` values, while radius is `float`. Note that this is slightly **different** from the format used in (Brandt, et al., 2019) and described in Chapter 3.2.3.

To make utilization of this struct easier, the `SurfelSample` class contains methods for **packing** and **unpacking** its data into separate values. These helper methods are currently used only for debugging since the actual "instances" of these structs are all created and manipulated solely on the GPU side **in shaders**.

Conversion from `SurfelSample` to `RenderableSurfel` happens in the `ConvertSurfels` stage through a compute shader to make it as efficient as possible. This means there is **no need to copy** buffers from GPU to CPU memory and back.

5.10.2 Vertices

Interfaces

As a base for vertices structs, there are 2 simple interfaces. `IVertex` providing nothing but **position** – useful for simple operations such as calculation of bounding boxes. And `IObjParsableVertex` providing a static method to **parse vertex out of the .obj file**. This is needed since different types of vertices have different properties and, therefore, are parsed from .obj in different ways. Parameters for `IObjParsableVertex.ParseFromObj` method (group, face, and face vertex index) are generic enough to interpret this vertex in any way .obj files allow.

Specific structs

For the actual structs used for the representation of vertices, there are `ColorVertex` and `TextureVertex`. Both contain **position** and **normal vector**, and either **color** or **texture coordinates**. It looks like this:

```
public readonly record struct ColorVertex
    : IVertex, IObjParsableVertex<ColorVertex>
{
    [Format(Format.R32G32B32Sfloat)]
    private readonly Vector3D<float> position;

    [Format(Format.R32G32B32Sfloat)]
    private readonly Vector3D<float> normal;

    [Format(Format.R32G32B32Sfloat)]
    private readonly Vector3D<float> color;

    public ColorVertex(
        Vector3D<float> position,
        Vector3D<float> normal,
        Vector3D<float> color);

    public ColorVertex(
        float x, float y, float z, // position
        float nx, float ny, float nz, // normal vector
        float r, float g, float b); // color

    public static ColorVertex ParseFromObj(
        LoadResult result,
        Group group,
        Face face,
        int faceVertexId)
    {
        return new ColorVertex(...);
    }
}
```

These structs are actually **record structs** – which removes the hassle of manually implementing the whole `IEquatable<T>` interface.

These structs for vertices are used primarily for **debugging** and **demonstration** purposes. Therefore, there was no incentive to merge these structs into more generic ones capable of holding various formats vertices. Besides color and texture, this could be various colors (ambient, diffuse, specular), material parameters/coefficients, illumination model, displacement and bump maps, etc. Implementing such a system is significantly outside of the scope of this work.

5.11 Meshes

Interface

The `IMesh` interface is at the bottom of the hierarchy for mesh-related classes. It specifies the used `PrimitiveTopology` (most commonly `TriangleList` for standard triangular meshes), methods for getting **AABB** (axis-aligned bounding box) used for camera placement, and most importantly, methods for retrieving **vertex and index buffer arrays**.

The only part of the algorithm that directly uses meshes is `SamplingStage`.

Base class

Abstract `MeshBase` class (implementing `IMesh`) was created to make working with meshes easier. It fully implements AABB-related methods. This is done thanks to its generic type being restricted to `IVertex`, whose only requirement is getter for a position. `MeshBase` also provides convenient `Lists` for vertices and indices, which enables the implementation of `GetVertexBufferArray` and `GetIndexBufferArray` from the `IMesh` interface.

Specific classes

`TriangularMesh` class then fully specifies `PrimitiveTopology` and further restricts its generic type to `IObjParsableVertex`, which makes it possible to implement the `LoadFromObj` method. This is mainly for **debug** reasons since it is expected to transfer meshes in different ways and formats based on the **architecture of the external program** that would use this app.

However, as mentioned, the `IMesh` interface is all that is needed, and it is generic enough to enable easy implementation from the external program.

The last mesh-related class is `TestingMesh`. It directly inherits from `TriangularMesh`, sets its generic type parameter to `ColorVertex`, and provides static `Lists` of **testing vertices and indices**. This makes it possible to test meshes without any external factors and settings. Therefore, it minimizes potential errors caused by parsed external files.

5.12 Helper functions

There are several static classes containing helper methods that were not suitable to be anywhere else and could be made static.

MathHelperFunctions

One of the simple ones is `MathHelperFunctions`. The majority of math-related operations on vectors and such are handled using the `Silk.NET.Maths` namespace. There was still a need to implement other methods, such as **conversion between coordinate systems** (spherical to cartesian and vice versa), and this class seems ideal for these methods.

MiscHelperFunctions

`MiscHelperFunctions` is reserved for **general-purpose methods** that are too small and unique to be placed into a separate helper class but are still needed to be called from various parts of the code.

RenderDocHelperFunctions

`RenderDocHelperFunctions` class is used to simplify work with **RenderDoc API**, especially using `IDebugProvider` (see 5.13). It also takes care of the interpretation of C-style results from calls to this API.

VulkanHelperFunctions

A similar but way more complex helper class is `VulkanHelperFunctions`. It was created with a job to simplify **tasks related to work with Vulkan** which could be called from more than one place and therefore were not suitable to be placed directly into the caller class. These tasks include creating and updating buffers, images, commands, pipelines, handling memory and allocations, and others.

It also takes care of error handling – processing `Silk.NET.Vulkan.Result` and throwing exceptions in case the negative result of a call to the Vulkan function is non-recoverable. `VulkanHelperFunctions` could be considered a mini framework for Silk.NET Vulkan library since most of these functions would be useful in any program accessing Vulkan through Silk.NET, not just this one.

This is not an uncommon approach when it comes to Vulkan. Monado[17], an open-source OpenXR runtime, uses a similar class with helper functions for the most common Vulkan functions.

ObjParsingHelperFunctions

Another class with wrapper-like helper functions is `ObjParsingHelperFunctions`. It is used to simplify **parsing** of the most common properties (such as position, normal, and color) from the `ObjLoader` library as well as calculating normal vectors in case they are missing.

ElementsHelperFunctions

The second helper class closely associated with elements is `ElementsHelperFunctions`. It helps to **generate Vulkan structures** such as `VertexInputAttributeDescription` and `VertexInputBindingDescription` out of elements (various surfels and vertices).

To do that as easily as possible, it utilizes C# **reflection** and custom **attribute** `FormatAttribute` to denote data fields of elements structs with `Silk.NET.Vulkan.Format`. An example is the `Vector4D<float> color` field in `RenderableSurfel` being marked with `Format.R32G32B32A32Sfloat`.

It might seem like this could be even **more automated** since `Vector4D` shows there should be 4 elements, and `<float>` could be clearly used to determine that individual parts are 32-bit floats. However, it is not as straightforward for all types – for example, the difference between `R8G8B8A8SNorm`, `R8G8B8A8Sscaled`, `R8G8B8A8Sint`, and `R8G8B8A8Srgb` cannot be deduced from C# type alone. It could easily lead to unintended behavior. Also, there are over 300 Vulkan formats; therefore, making this conversion at least somewhat reliable would still require several orders of magnitude more work than just setting simple attributes such as `[Format(Format.R32G32B32A32Sfloat)]` to each field.

[17] https://monado.dev/

5.13 Providers

Various calls to Vulkan API require a lot of different Vulkan objects, most of which are **constant** for the whole duration of the application run. This includes `Vk` (does not exist in the pure C version of Vulkan API; it is just a way to provide global functions in C#), `Instance`, `Device`/`PhysicalDevice`, `AllocationCallbacks`, and others. This results in passing many parameters from one class to another. To make code more readable, flexible, and shorter, a system of **providers** was created.

Providers are classes that implement one of the **provider interfaces**. These interfaces require an implementation of getters for the aforementioned Vulkan objects. As a result, classes implementing these provider interfaces can then be passed as parameters to other classes.

The list of provider interfaces is following:

- `IMinimalVulkanProvider`
  - groups together aforementioned Vulkan objects
- `IDebugProvider`
  - gives access to `ExtDebugUtils` (Vulkan **debug** utilities) and instance of `RenderDoc` (from `Evergine` namespace)
- `IComputeQueueProvider`, `IGraphicsQueueProvider`, and `IPresentQueueProvider`
  - provides access to instances of compute, graphics, and present **queues**, respectively
- `IStandardProvider`
  - only groups together `IMinimalVulkanProvider` and `IDebugProvider`

The class that implements all of these (except for `IPresentQueueProvider`) is `VulkanAppBase`. Higher in the class hierarchy is then `WindowedRendererBase`, which adds `IPresentQueueProvider` to a list of interfaces it implements. Since `VulkanAppBase` is the backbone of everything Vulkan-related (initialization of all these Vulkan objects), it is straightforward for it to provide easy access to all these objects.

Reference to `VulkanAppBase` can be **passed even deeper in the method call** hierarchy without a need to store references to all these Vulkan objects individually in each class on the way to the final API call. As an example, `VulkanHelperFunctions.CreateBufferAndMemory` method takes `IMinimalVulkanProvider` as one of its parameters. Then it takes whatever it needs from this provider and passes it whole to other methods it calls. Some Vulkan objects from this provider interface will be **redundant**, but it keeps the code cleaner and provides no performance hit since everything is passed as a reference anyway.

These interfaces make it easier to **replace** `VulkanAppBase` or **extract initialization** of these Vulkan objects elsewhere if needed.

This approach also makes it possible to check what exactly the passed-in provider is and behave accordingly. For example, as mentioned, the `CreateBufferAndMemory` method takes `IMinimalVulkanProvider` as a parameter. But then it calls the `AssignDebugNameConditional` method to assign debug names for just the created buffer and memory. This method takes just `IMinimalVulkanProvider` as one of its parameters. But if this passed `IMinimalVulkanProvider` is also `IDebugProvider`, its `ExtDebugUtils` method is used to assign a **debug name**. This makes switching between debug and non-debug (release) versions of providers possible.

The approach to **encapsulate several core Vulkan objects** is quite common. For example, Monado has a class `vk_bundle` containing a similar collection of objects. However, it is less flexible since it contains only a single queue.

5.14 Consumers

Directly related to the aforementioned providers is the `MinimalVulkanConsumerBase` class. This class provides a mechanism to simplify code for classes that need to store an instance of `IMinimalVulkanProvider` and easily **access its properties**. To do that, the class can inherit from `MinimalVulkanConsumerBase`, assign `IMinimalVulkanProvider` to its property (through the constructor of `MinimalVulkanConsumerBase`), and get direct access to the underlying properties of this provider through getters from `MinimalVulkanConsumerBase`. For example, instead of accessing `MinimalVulkanProvider.CommandPool`, it can be accessed directly through `CommandPool`.

This mechanism exists only to make code a bit **shorter and cleaner**. Since classes in C# cannot inherit from more than 1 class, this might cause inconvenience if a large refactor of classes is done in the future. It does not cause any known issues right now. `MinimalVulkanConsumerBase` could be changed **from class to interface**, solving the problem with single-class inheritance. However, that would cause a loss of the base constructor, which **forces** inheriting classes to set `MinimalVulkanProvider`.

Also, this class and the whole mechanism can be **removed** without directly affecting the system of providers.

6 Implementation highlights

This chapter mentions and analyzes some of the most critical **decisions** in architecture and low-level **implementation**. It contains a detailed comparison of various approaches/algorithms for the problem and a **justification** for the chosen solution.

6.1 Camera placement algorithms

A small number of cameras (a few dozen at max) used for **sampling** in the first stage must be placed **around the sampled 3D model**. The most universal solution without the need to analyze actual mesh is to place cameras around it **uniformly**. For example, on the surface of a **sphere** around the whole mesh. Cameras, in this case, would be all pointed into the center of this sphere.

Several algorithms for **uniform placement** (in this case, same as sampling) on the sphere were tested. Placement is calculated only once per model, so the speed of actual calculation is not critical.

6.1.1 Hand-picked values

Using saved hand-picked values for placement is probably the **simplest** solution, but it lacks flexibility with a varying number of cameras.

6.1.2 Spherical coordinate system

The standard **cartesian coordinate system** is unsuitable for dealing with coordinates in a sphere and on its surface. For cases like this, the **spherical coordinate system** is a great candidate. It is specified by **3 numbers**: a distance from the sphere center, polar angle, and azimuthal angle (see Figure 16). Based on ISO convention[18], these variables are referred to as $\boldsymbol{r}$, $\boldsymbol{\theta}$, and $\boldsymbol{\varphi}$, respectively.

[18] ISO 80000-2:2019 Quantities and units - Part 2: Mathematics

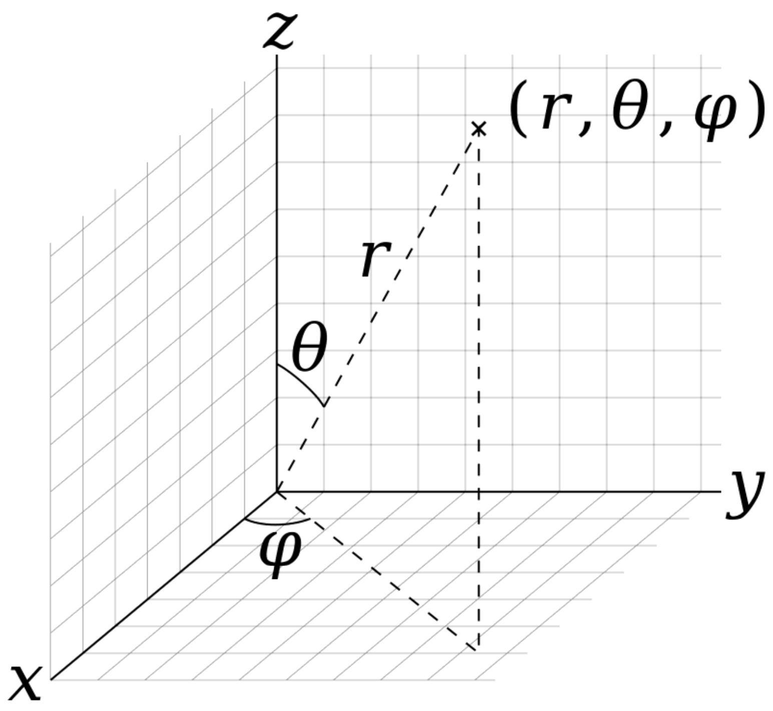


Figure 22 Spherical coordinate system illustration. Source: Wikipedia - Spherical coordinate system.[19]

Since cameras are supposed to be placed on the surface of a sphere, **distance *r* is fixed**. However, a **uniformly random** setting of θ and φ coordinates does not result in uniformly distributed samples. And even worse, with few cameras, results are **highly unpredictable**. An additional check for the quality of samples and their eventual re-calculation would be required.

6.1.3 Fibonacci sphere

One of the solutions for the even distribution of samples on the surface of a sphere is called the **Fibonacci sphere** (sometimes referred to as the **Fibonacci lattice**). The algorithm for its calculation is fully **deterministic**, and the result is **uniform** (see Figure 17).

[19] https://en.wikipedia.org/wiki/Spherical_coordinate_system

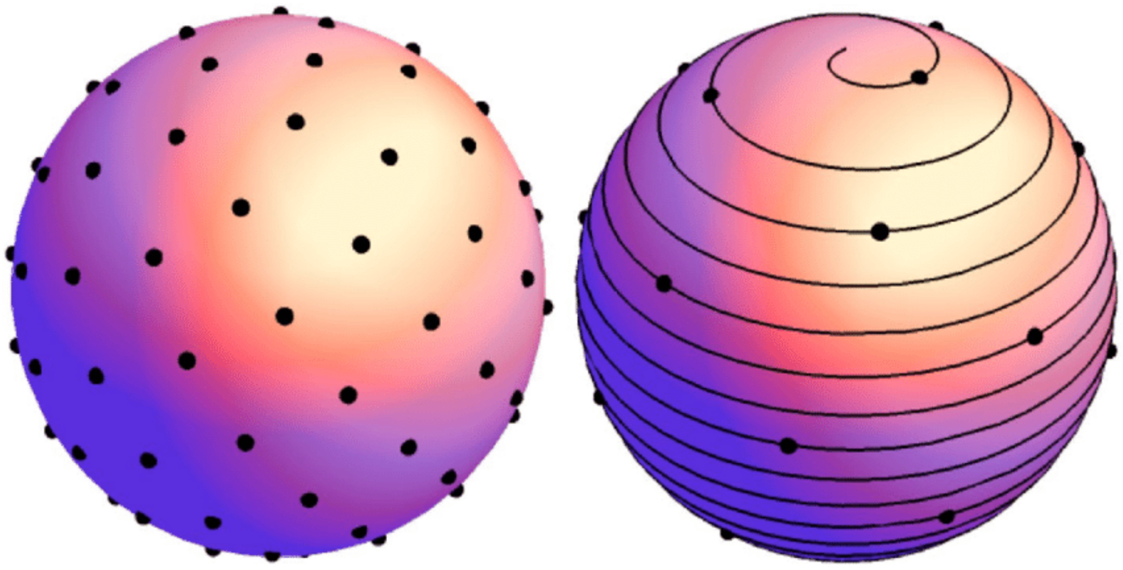


Figure 23 An example of a Fibonacci sphere (left) and one of the spirals it was generated from (right). Source: (Munguba, et al., 2021)

A visible **pattern** in samples will appear with a high number of samples. However, when working with a number of samples on the order of tens (as used in this application), the resulting sampling looks **random** enough while being nicely **evenly distributed**.

Since the Fibonacci sphere algorithm perfectly **meets all requirements** for sampling on the sphere and is very **easy to implement**, it was chosen for the final implementation.

6.1.4 Implementation

Internally, the `SamplingCamerasGenerator` class takes care of **calculating positions** for `ICameras`. `GeneratorTechnique` enum then provides the ability to choose which technique should be used: `RandomPolarCoordinates`, `FibonacciSphere`, or `DEBUG`. The last one is just a set of hand-picked values used solely as a deterministic set for debugging purposes.

6.2 Organization of shaders

The majority of shaders used in this project are short and straightforward enough to be contained in a **single file** each.

The only **shared functions** between shaders are in `PackUnpack.inc` and `DimensionsConstants.inc` files. The first mentioned contains macros for

packing and **unpacking** `uint` into `ivec3` containing texture coordinates (U and V coordinates and texture layer index). The second mentioned contains constants for the **width** and **height** of the viewport in pixels and the maximum **number of views** available.

6.2.1 Compact and Sort stage

The only **exception to short and more-or-less self-contained shaders** is the compute shader for the Compact and Sort stage. It consists of **6 phases** that run one after another.

Quite a few **shared** functions, constants, and bindings exist between them. Therefore, the problem with the division of this large shader arose.

- Splitting phases **each into their file** would cause chaos with **shared parts** of code, which would have to be added using the `#include` directive.
- The other option is to use the so-called **"ubershader"** – put everything into one big shader file and handle it one of the following ways.
    - Usage of GLSL shader **subroutine** (conceptually similar to function pointers in C) would be ideal for this case. Unfortunately, SPIR-V shaders **do not support** this functionality.
    - Vulkan supports setting the shader's **entry-point** (`PName` variable in `PipelineShaderStageCreateInfo` struct). However, this **does not work for GLSL** shaders, which mandate the use of the `void main()` function as an entry-point.
    - Both glslangValidator and glslc (GLSL/HLSL to SPIR-V compilers) support the setting of `--source-entrypoint` parameter, which specifies GLSL/HLSL **entry-point**. Even though glslangValidator seems to log an error when a function other than `main` is used with this parameter, it does seem to work. However, this would require **compiling a shader once for each phase** – just with different entry-points. It would

ultimately lead to an increased memory footprint and a more complex build process.

- Besides that, no matter how various entry-points would be achieved, each phase would now require its **own pipeline** and bind it before the phase is called. Even though the added complexity of code and performance hit would probably be minor, it is good to consider them.

- The most straightforward way is to compile a single copy of the shader the standard way with the `main` function and use **branching** (if/else or switch) to call the appropriate method to handle phases. Plus, since this branching is based on the Push Constant (whose value is pre-recorded in the command buffer), it can be very effectively optimized, and a call to the main function with branching would be equivalent to directly calling the phase function as an entry-point. And even in case this optimization is not utilized by the graphics driver, the performance hit is expected to be relatively insignificant. One of the drawbacks is the potential **bug-prone** missing association between phase indices and actual phases. The only link between these in CPU and GPU codes is a new Push Constant (`uint`), and any change to the order of phases has to be adjusted in several places. However, this is a usual issue when writing CPU and GPU code (shaders).

Due to its simplicity and relatively few drawbacks, the last option (**ubershader** with the `main` function as a single entry-point and **switch statement**) was selected for the organization of compute shader for the Compact and Sort stage.

6.3 Operation atomicExchange vs. atomicMax

The task of the compute shader for the Choose First Sample stage is to select the first sample for the rest of the algorithm. It does not matter which pixel is selected for the first sample as long as it is valid. The body of the shader looks like this:

```
const ivec3 textureUVW = ivec3(gl_GlobalInvocationID.xy, 0);
const    vec4    worldPosition    =    texelFetch(worldPositionsTexture,
textureUVW, 0);

if (worldPosition.w > 0) // pixel is valid
{
    atomicMax(sharedPackedSampleCoordinate,
        packCoordinate(textureUVW));
}
```

This compute shader goes through pixels of `worldPositionsTexture` (at layer 0, since it does not matter which layer is chosen) and checks which pixel has a **value** of `W` coordinate **non-zero**. This is sufficient and the **only condition** for a pixel to be considered **valid**; therefore, it can be chosen for the first sample.

Next, the shader needs to **pack the coordinates** of this pixel and set `shared uint sharedPackedSampleCoordinate` to its value. After proper synchronization of invocations using barriers, this shared variable is written into the global array `surfelSamples.samples[0]` by locally first invocations (it does not matter which specific invocation is used) of each workgroup.

The interesting part is the actual setting of the `sharedPackedSampleCoordinate` variable. This variable is shared between all invocations within a single workgroup. **Atomic operations** are the best candidates for this since they do not require manual synchronization between invocations.

Based on the operation's name alone, `atomicExchange` might seem ideal for this task. However, using `atomicMax` instead provides one very good advantage – **result reproducibility**.

Scheduling of workgroups is entirely up to the driver, and individual invocations in each workgroup would call `atomicExchange` operation in a different order, resulting in **different results** (different first samples) for **each application run**. Function `atomicMax` ensures the same result regardless of the scheduling of workgroups and invocations. This is great for **debugging** purposes. And since the Choose First Sample stage is called only once at the beginning of the algorithm run and the performance difference between these 2 atomic operations is negligible, `atomicMax` is used even outside debug runs.

The **same mechanism** is used at the end of this shader – at the place where locally first invocations of each workgroup set the global value of `surfelSamples.samples[0]` equal to `sharedPackedSampleCoordinate` (packed into surfel sample). Usage of function `atomicMax` will result in the same value in `surfelSamples.samples[0]` each algorithm run, regardless of workgroup scheduling.

Of course, it should be noted that `atomicMax`, in this case, acts as `atomicExchange` only because the `sharedPackedSampleCoordinate` variable is **set to 0** at the beginning of the shader run, and all valid packed coordinates are represented as positive **non-zero** numbers.

The same technique with the same context is used in the `ExtractFarthestSamples` compute shader.

6.4 Unsafe code

Due to the nature of Silk.NET, it is impossible to avoid using **unsafe** C# code in this project. This essentially means using **raw asterisk pointers** (such as `void*`). This is used only when necessary due to undelaying API. Some parts of code also require using `IntPtr` or `nint` to store pointers (for example, to call `Marshal.StructureToPtr` method).

Handling and management of **raw global memory** could not be avoided either. This is, for example, when **chaining several structs** using `void* PNext` of Vulkan structs. `PNext` is a pointer to the next Vulkan struct, which is used to add information, for example, using extensions. Almost all Vulkan structures contain this field, which leads to the possibility of chaining as many structs as needed.

Unfortunately, in C#, this is problematic due to the **automatic garbage collector**, which does not know that the pointer is still used by some `void*`. To fix this, Silk.NET offers the method `SilkMarshal.Allocate` method (and its counterpart `SilkMarshal.Free`) to allocate global memory. This memory can then be populated using `Marshal.StructureToPtr`. This is precisely what `VulkanHelperFunctions.StructToGlobalMemory` method does.

Working with global memory results in extra care needed to be taken by a programmer to avoid **memory leaks**. This is especially true for C# where unsafe code and memory leaks in general are very rare.

6.5 Sampling colors

Right now, the Sampling stage directly takes only a **single sample of color** for a given fragment. In some rare cases, this might cause graphical issues such as **aliasing** (see Figure 18).

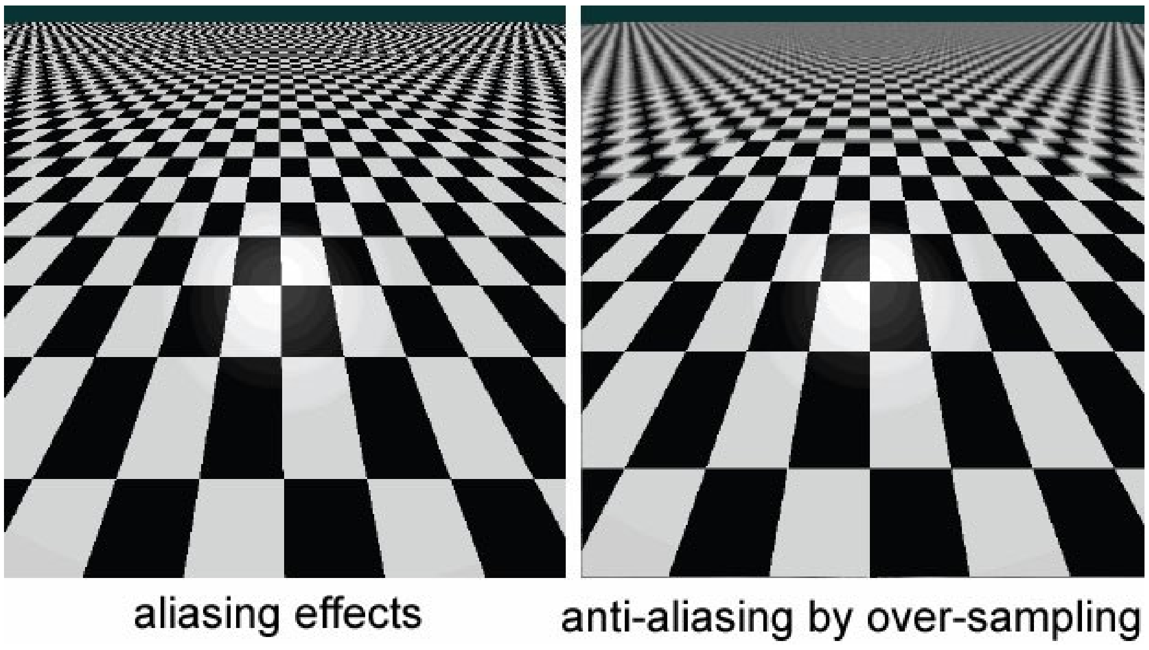


Figure 24 An example of an aliasing in computer graphics caused by insufficient sampling. Source: Spatial Antialiasing - Presented by Tiger Giraffe.[20]

An easy way to improve it would be to take multiple samples from each fragment and average them. This can usually be hardware accelerated since the same technique has been used for **MSAA** (multisample anti-aliasing) in games and other software for many years.

Another more advanced approach would be to take multiple samples not just from a given fragment but from **neighboring fragments** as well. This is more difficult to implement and will result in significantly higher performance hit.

[20] https://mielliott.github.io/index.html

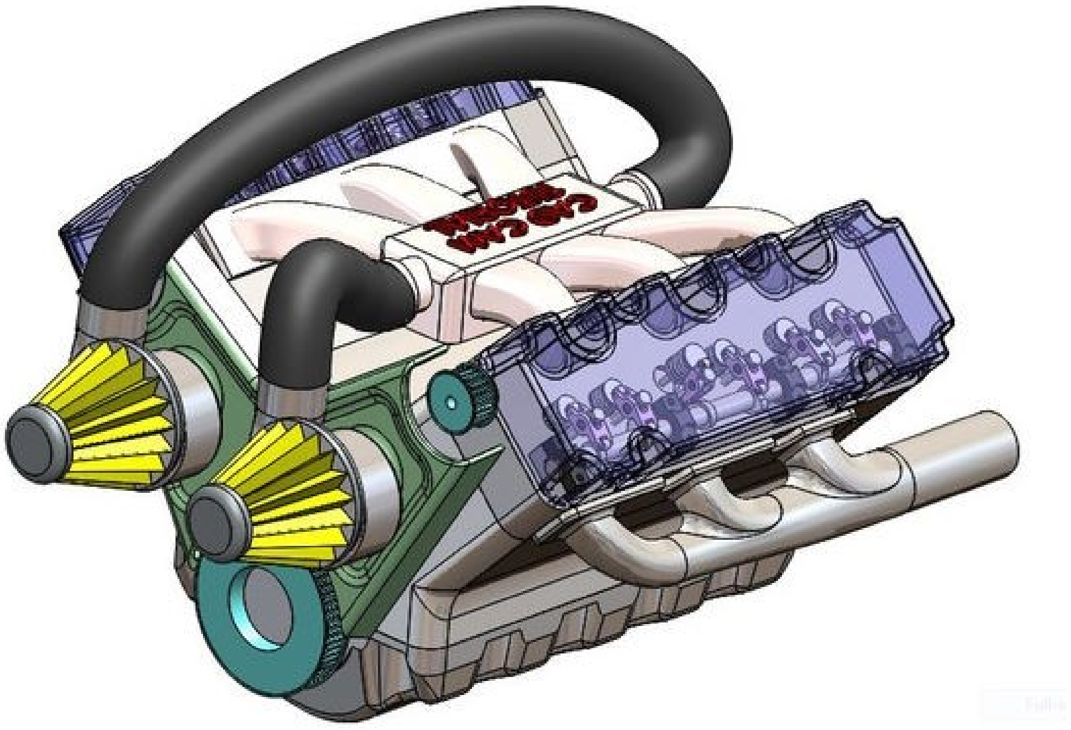


Figure 25 An example of a CAD model. Source: "Car Engine" model by Mahtabalam Khan published on GrabCAD Community website.[21]

However, it is essential to consider what data type is expected to be sampled. For the case of this project and the whole FataMorgana platform, the most common 3D models on input are **CAD models**. They usually have **uniform colors** and very few places to cause aliasing since **textures are rarely used** (see Figure 19 for an example). Therefore, it was decided that improvements in the color sampling process have **low priority** and are outside this project's scope.

6.6 Replacement of Sampling stage

As mentioned in Chapter 5.4.2, the Sampling stage is the **only stage that works with an actual mesh** representation of the scene. All other stages work with G-buffers containing color, normals, and positions for pixels generated in the Sampling stage.

This means the Sampling stage can be easily replaced with anything else that can provide these attributes. This could be, for example, a **3D scanner**, **photogrammetry system**, **ray-tracer**, or even some novel neural-network-based

[21] https://grabcad.com/library/car-engine-8

techniques such as NeRF (Mildenhall, et al., 2020) or Gaussian splatting (Kerbl, et al., 2023).

Note that the **camera's position** with a **depth map** (more prevalent for some of these techniques) instead of the actual positions of pixels is also sufficient since world positions can be calculated from them.

6.6.1 FMBrain

One specific system that could replace the Sampling stage is **FMBrain**. This project was created in Pocket Virtuality as part of the FataMorgana platform.

The purpose of FMBrain is a production of **high-quality meshes** created by the **combination** of data from various sources:

- **low-quality meshes** provided by HoloLens sensors,
- **photos** periodically captured from HoloLens main RGB camera,
- **point clouds** from 3D scanners (such as Leica BLK 360),
- and potentially others.

It contains a **photogrammetry** pipeline for processing those photos and works with colors, normals, and positions for pixels. The exact outputs of the Sampling stage. Therefore, this would be the first **candidate for replacement** of this stage if needed.

6.7 Nullable warnings

Novel C# versions bring features for **mitigating errors caused by null references**. This is mainly in the form of static analysis done by the compiler and can be turned on per file (`#nullable enable` preprocessor directive) or for a whole project (`<nullable>Enable</nullable>`).

It is recommended to write **new code** with this feature **enabled**. However, as this project shown, it is not always possible or beneficial enough.

Since the majority of the work of this program is done on the **GPU side**, occasional null checks in CPU code will not cause a noticeable performance hit.

The vast majority of Silk.NET Vulkan API consists of structs that are **not nullable**. They can be made nullable using `StructType?` notation (shorthand for `Nullable<StructType>`). But most of these Silk.NET structs are just wrappers around its single field `public ulong Handle`. This works like an opaque "pointer" for Vulkan API. If the `Handle = 0`, it is considered an **equivalent of a null pointer**, and use of it in API calls will, in most cases, throw an exception or return an error result.

The compiler's static analysis, of course, does not check for the value of `Handle`; therefore, it would still **miss a lot of possible runtime errors** caused by a null reference.

6.7.1 MemberNotNull attribute

Attributes `MemberNotNull` and `MemberNotNullWhen` could help, but unfortunately, they do not play nicely with **inheritance**. More specifically, using these attributes in such cases results in *"Warning CS8776: Member '<member from base class>' cannot be used in this attribute"*. This problem is still being worked on, as seen in GitHub issue #56531[22] for the Roslyn compiler.

Another problem with the `MemberNotNull` attribute is that it does not seem to work nicely with method calls. Imagine this code:

```
DescriptorsManager decriptorsManager;

void Initialize()
{
    CreateDescriptorsManager(); // creates decriptorsManager
     CreatePipeline(); // the inside of this method does not know that
descriptorsManager is not null
}
```

Then we create the `IsInitialized` property and `AssertIsInitialized` method like this:

```
[MemberNotNullWhen(true, nameof(descriptorsManager))]
bool IsInitialized { get; set; } = false;

void AssertIsInitialized()
{
    if (!IsInitialized)
    {
        <handle error>
    }
}
```

[22] https://github.com/dotnet/roslyn/issues/56531

After all this, it might seem like the following code for the `CreatePipeline` method should be able to detect whether `descriptorsManager` is null or not:

```
void CreatePipeline()
{
    AssertIsInitialized();

    pipelineLayout = VulkanHelperFunctions.CreatePipelineLayout(
      descriptorsManager.DescriptorSetLayout,…);
}
```

If static analysis for nullability is turned on, a warning saying that *"descriptorsManager may be null here"* in a call to `CreatePipelineLayout` appears. This warning can be removed only by copying the **whole body** of the `AsserIsInitialized` method directly inside of the body of `CreatePipeline`. The call to `AssertIsInitialized` (as in the code example above) is **insufficient**. This would, naturally, result in a lot of copied code, which is a bad practice.

Various **other static analyzers** for nullability included in Visual Studio require much work before being fully usable.

6.7.2 API call and out parameter

Furthermore, **changing these Vulkan structs to nullable** would break compatibility with almost all API calls. Let's say we have this code:

```
private Queue queue;

void SetQueue()
{
    Vk.GetDeviceQueue(…, out renderingGraphicsQueue);
}
```

Changing `Queue` to `Queue?` is not compatible with `out` parameter of `Vk.GetDeviceQueue` method. And majority of these structs are created exactly this way – as `out` parameter instead of a return value (which is reserved for `Result`). This problem can be fixed by introducing a new **method-local variable** and assigning it to a class variable `queue`. Still, it would add **boilerplate** code for each of these API calls, making the code less readable.

6.8 Rendering of surfels

`SurfelRenderer` is used only for **debug visualization** of generated surfels. It is still a good demo of how easy it is to draw these surfels. The process it uses is explained in detail in Chapter 5.3.2.

This chapter talks about rendering **primitives** that can be used for rendering surfels and their **blending** with each other and polygonal meshes.

6.8.1 Shape of surfels

Surfels can be rendered as various shapes using several techniques.

Circles

Using a **geometry shader** to create a triangle, which is then **"cut" into a circle** in a fragment shader (as `SurfelRenderer` does), is not the only possible technique to render surfels. And also, it is not always the best. For example, the hardware architecture on Microsoft HoloLens 2 causes the geometry shader to be **very slow**.

Squares

One of the alternative approaches would be to render a **simple square**. This does not require a non-standard approach with geometry shader. However, without a geometry shader, there would have to be **4 vertices** for every surfel in the vertex buffer instead of just a single vertex. Which translates into more data stored and more work for the vertex shader.

Point sprites

Older graphics APIs such as DirectX 9 and OpenGL support so-called point sprites. They are a **generalization of points** as primitives. The primary use for them was particle systems. However, since they support the setting of size and color, they could be an alternative to the aforementioned approaches.

Modern graphics APIs **do not offer** this functionality anymore. Therefore, it was not investigated further. It is possible that this technique was never intrinsic to GPU hardware and was just **emulated** using a geometry shader and subsequent discarding of fragments similar to the rendering of surfels as circles mentioned above.

Squarkle

Azure Remote Rendering supports native rendering of point clouds. Each point is rendered as a squarkle – a **combination of square and circle**. This technique results in **space coverage** almost as good as a square, but it keeps the **precision** of a circle.

Unfortunately, **no more information** was provided or found. It might be a good idea to keep that in mind and later check papers describing this technique in more detail.

Using squarkles for rendering was mentioned in the Azure Remote Rendering presentation for Mixed Reality Dev Days 2022[23].

6.8.2 Transition between surfels and original mesh

One of the more advanced techniques required for high-quality rendering in the final product would be a transition between surfels and the original representation of mesh when the model gets close enough to the camera.

To prevent ugly **"popping"** during the transition, both mesh and surfels of the same models could be rendered simultaneously, and alpha blending would be used to transition between them smoothly. This is, however, strongly dependent on the external system – primarily its rendering pipeline and representation of the original mesh.

6.8.3 Blending between surfels

Alpha blending could also be used for nicer blending between surfels themselves. They could be rendered opaque in the center and progressively more transparent toward the edges – similar to Gaussian splats (mentioned in Chapter 2.3.4).

This technique was used in several older approaches to rendering surfels. However, surfels in those cases were usually significantly larger than those in this algorithm. Therefore, it is possible that this approach would bring little to **no visual improvement** in most cases. What it would definitely bring is, however, the **rendering complexity**. Whenever objects with transparency are rendered, they need

[23] https://youtu.be/R6SoCL25nCY?feature=shared&t=1980 (timestamp 33:00)

to be rendered in an ordered manner (back-to-front). Using alpha values also means **no early depth tests**, resulting in another impact on performance.

6.9 FMsurfelsDebugTools

Part of the `FMsurfels` solution is the project called `FMsurfelsDebugTools`. It is a tiny project consisting of a single `Program.cs` file with all code being just in the `Main` method.

Its only purpose is to provide a simple command line tool to **decompose a surfel sample** packed in `ulong` (64-bit unsigned integer) into its parts – **texture coordinates** (2 `uint` values), **layer index** (`uint`), and **radius** (`float`).

Even though it is relatively simple, it is still robust regarding input parsing, and it correctly displays errors instead of just throwing an exception and shutting down.

6.10 Shared texture in Voronoi stage

The last 2 lines of the fragment shader for the Voronoi stage look like this:

```
outDistance = distanceToSite / bounds.maxDistance;
gl_FragDepth = distanceToSite / bounds.maxDistance;
```

and the question might be – why not share 1 texture between the depth buffer and G-buffer of distances (`outDistance`) since both are assigned the same value?

The answer is that the G-buffer of distances is read in the shader in the next stage. For that, its `Image` needs to be in the `ShaderReadOnlyOptimal` layout. Depth buffer, on the other hand, requires the use of `DepthStencilAttachmentOptimal` layout. The **transition** between these 2 `ImageLayouts` back and forth in each algorithm iteration might be too big of an **overhead** (depending on architecture and drivers). Added complexity to code is also to be considered – `ImageLayout` transition must be done through `CommandBuffer`.

The gain from sharing one texture would be just a **little saved memory**. And almost **no processing time** (initialization and disposal of 1 of the textures) would be saved because of all the new wasted time on `ImageLayout` transitions.

6.11 Disposal of Vulkan objects

All classes creating Vulkan objects also **dispose** of them to **prevent memory leaks**. This is done through `Vk.Destroy…` methods. To make this more manageable, `System.IDisposable` interface is fully implemented on all relevant classes.

While `IDisposable` requires only the implementation of a single `void Dispose()` method, the reality is a bit more **complicated** (primarily due to classes with **inheritance**).

In the end, the whole implementation of C# **Dispose Pattern** (for needs of this project) for a single `ExampleClass` class looks like this:

```
protected bool disposed = false;

public void Dispose() // required by System.IDisposable interface
{
    Dispose(true);

    GC.SuppressFinalize(this);
}

protected virtual unsafe void Dispose(bool disposing)
{
    if (disposed)
    {
        return;
    }

    Vk.Destroy…(…, Allocator);

    <other IDisposable classes>?.Dispose();

    disposed = true;
}

~ExampleClass() // destructor
{
    Dispose(false);
}
```

6.12 Multiple GPUs

One of the considered performance improvements is the use of multiple GPUs. This could be done in several ways.

It is important to note that this algorithm is **not real-time**, unlike games. Therefore, these approaches to utilize multiple GPUs are **not strictly related** to technologies such as **SLI** or **CrossFire** used for synchronized cooperation of GPUs. Also, both SLI and CrossFire have seen a significant **loss of interest** in the last years in the consumer market. The only remaining useful technology for this is **NVLink**, which is focused more on enterprise solutions.

6.12.1 Per-model basis

Utilizing multiple GPUs on a per-model basis would mean that **each sampled model is processed on a different GPU**. Since each processing means a new and independent run of this program, there would be no issues related to memory conflicts and synchronization. It also means there is no need for tight cooperation of GPUs using aforementioned technologies such as SLI or CrossFire. However, this approach would have to be fully **implemented in external application**.

The only change in this program would be an added way to **select a desired GPU** from the outside. That is relatively easy to do.

6.12.2 Inside of algorithm

The second approach to utilizing multiple GPU would be to modify this program by **splitting the load**. This would require extensive changes to the algorithm as well as handling of **synchronization of memory and states**. Doing this correctly for an arbitrary number of GPUs is a very complex task and significantly out of this project's scope.

Since **each run** of the algorithm (for a single model) is expected to be relatively **fast**, it is doubtful whether this approach would bring any significant advantage.

6.12.3 Conclusion

Based on the aforementioned analysis of 2 approaches to utilize multiple GPUs, it is pretty easy to see that **“per model basis”** is easier to implement in general, less prone to issues with synchronization, less dependent on other technologies, and last but not least it would probably bring more significant performance advantage.

This is still just an **initial analysis** – at best, it is a starting point for one of the future improvements/additions.

6.13 Layered rendering

Layered rendering refers to a technique when one render call results in writes to **more than 1 layer** of framebuffer/render target. This technique was initially used for cube-based shadow mapping and cube environment maps – instead of rendering scene 6 times per cube map, it can be rendered in a single pass. A more novel use of this technique is rendering for **stereo displays**, such as AR/VR headsets, stereo projectors/monitors, CAVE systems, etc.

Layered rendering is heavily utilized in this project (in G-buffer stages) to extract as much performance as possible.

There are 2 main approaches to layered rendering:

- **Geometry shader instancing** – This technique uses geometry shader to create multiple copies (instances) of each primitive it processes. The aforementioned example with cube maps can render the whole scene **3-4 times faster** than independent draw calls (Stenning, 2014, page 319). Its primary disadvantage is that the number of instances has to be set in **compile time** right in the shader as an attribute. It is possible to **re-compile** the whole shader with a new constant at runtime, but that significantly increases the complexity of the build process and is not very flexible.
- **Multiview rendering** – A more flexible approach is using multiview rendering. In Vulkan, this is provided by the `VK_KHR_multiview` extension. It allows to set a number of instances right at the **time of calling the draw call**. It also does **not require a geometry shader**, which would be an unnecessary bottleneck in cases where a geometry shader is not used for anything else.

Due to the disadvantages and worse flexibility of geometry shader instancing, **multiview rendering was selected** as a better candidate for layered rendering in this project.

6.14 Alternative for atomic operations

As mentioned in Chapter 4.1.4, DirectX 11 lacks an essential operation for this project – **atomic max on 64-bit floats**. There were attempts to work around this issue using **mutex**.

For evaluation purposes, 2 shaders were created to do essentially the same max operation on 1D texture containing `uint` values:

- One shader used **atomic operation** (`InterlockedMax`), and its code looked like this:

```
RWTexture1D<uint> SimpleSurfelSamples : register(u0);

[numthreads(THREADSX, THREADSY, THREADSZ)]
void ExtractFarthestPointCS(
    uint groupIndex : SV_GroupIndex,
    uint3 groupId : SV_GroupID,
    uint3 groupThreadId : SV_GroupThreadID,
    uint3 dispatchThreadId : SV_DispatchThreadID)
{
     uint coord = (groupThreadId.x * dispatchThreadId.y * groupId.z *
groupIndex) % 1000;

    InterlockedMax(SimpleSurfelSamples[coord], groupIndex);
}
```

- The second one, significantly more complicated, uses mutex and active waiting. Compared to the aforementioned shader, this one adds mutex in the form of 1D texture:

```
RWTexture1D<uint> SimpleSurfelSamples : register(u0);
```

and changes the body of `ExtractFarthestPointCS` function accordingly:

```
bool keepWaiting = true;

while (keepWaiting)
{
    uint originalValue;
    // try to set the mutex to 1
    InterlockedCompareExchange(Mutex[coord], 0, 1, originalValue);

    if (originalValue == 0)
    {   // nothing was locked (previous entry was 0)

        // do actual work
        if (groupIndex > SimpleSurfelSamples[coord])
        {
            SimpleSurfelSamples[coord] = groupIndex;
        }

        // unlock mutex
        InterlockedExchange(Mutex[coord], 0, originalValue);

        // exit loop
        keepWaiting = false;
    }
}
```

The **result of the performance evaluation** of these 2 shaders was undoubtedly in **favor of atomic operation**. While that shader took around **0.16ms** (average of 10 runs), the version of the shader with mutex took **1551ms** on average. That is several **orders of magnitude slower**. The standard deviation in both cases was small and did not play a role in the results.

This shows that using mutex instead of atomic operations in shaders would result in a **considerable performance hit**. Therefore, the use of DirectX 11 was not feasible.

7 Results

This chapter talks about the concrete results of the project as a whole – both the research and the implementation part. It consists mainly of **discussions of contributions** in various ways and to various parties, as well as the project's current state.

7.1 Research

The first part of this thesis consisted of **research** for adequate LoD technique. Chapter 2 went through various rendering optimization approaches, compared state-of-the-art LoD algorithms, and justified the selection of Blue Surfels for the second part of this thesis.

This research should not be considered a meta-analysis of LoD algorithms since it was constrained by the requirements for this project. However, it is still **extensive**, **detailed**, and **up-to-date** enough to help others get an overall idea of where research in the field of LoD is and even how it got there all the way from the first papers.

7.2 Improvements compared to existing implementation

The second part of this project, the implementation of the selected LoD technique, is **far more** than just a copy of the existing implementation of Blue Surfels.

First of all, this project **replaced deprecated OpenGL** with modern **Vulkan**. Besides the utilization of modern technologies, Vulkan also provides the most possibilities for future improvements and the best performance gain. All that while keeping multiplatform support similar to that provided by OpenGL.

The result of this project is a completely **independent executable unit** with no 1st party and only a minimum of (carefully selected) 3rd party dependencies. This is unlike the existing implementation, which relies heavily on university renderer PADrend and is not prepared for compilation on other platforms other than Linux. Even the compilation on Linux is not without issues.

The important part is also significantly more **readable**, **organized**, and **documented** code. This is a notable disadvantage of the existing Blue Surfel implementation. It looks like a typical research paper implementation – as long as it works and is implemented quickly, it is fine. There were no expectations of programmers using it other than authors themselves.

7.3 State of the project

The algorithm implemented for the purpose of this project is a type of algorithm that can be **extended** and **improved for years**. Naturally, it was impossible to include all of them in this project at the time of submission for the purpose of the thesis. However, some of these potential improvements are **mentioned** all around this thesis text.

The code's architecture is designed to be **easily extensible** in the future. **Future work** on this project was always expected since the code was developed partially for the company Pocket Virtuality.

Nonetheless, the submitted code for this thesis is fully functional from the beginning to the end. More specifically, the program takes an **.obj file on the input** and **produces a surfel representation** of it.

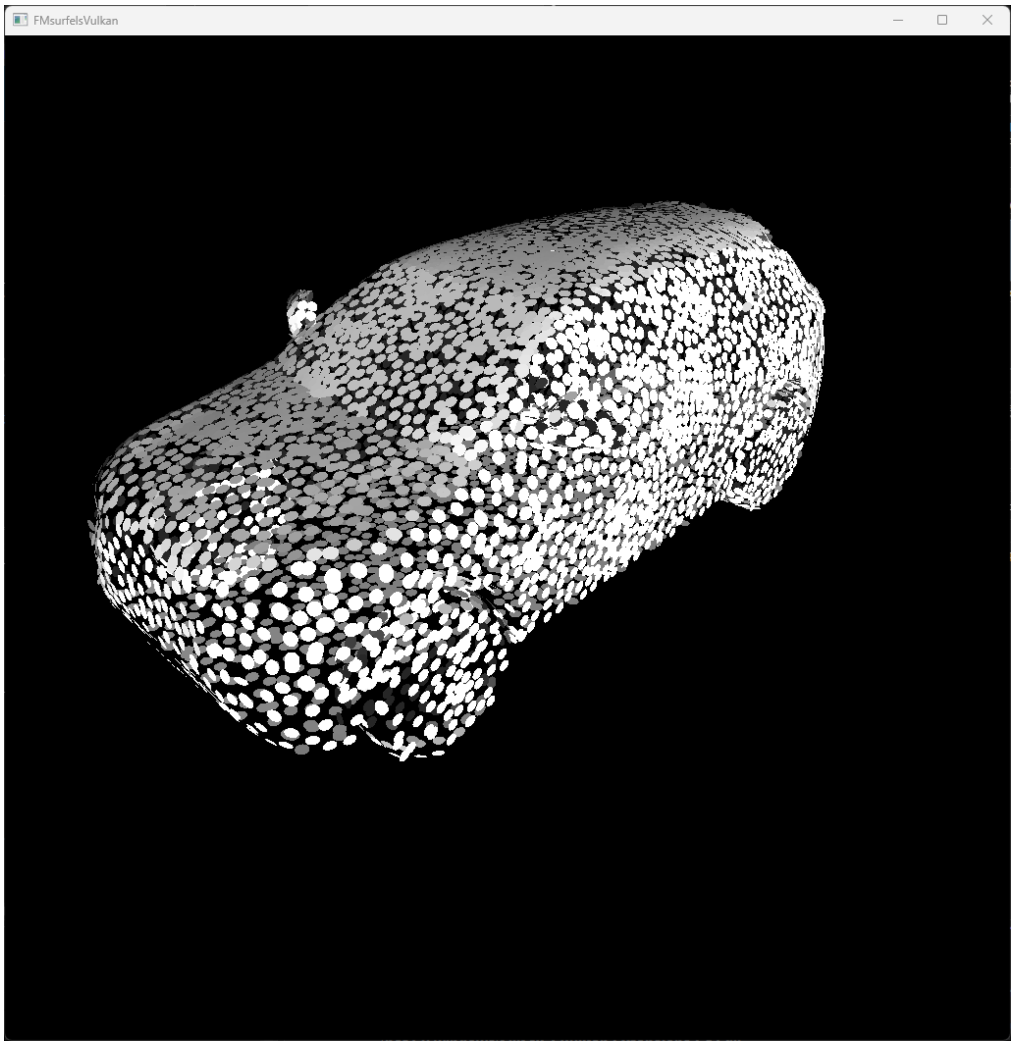


Figure 26 A debug renderer window displaying a sampled car model in the form of surfels (10k in total).

The submitted code also includes an interactive **debug renderer** (see Figure 24) for easy visualization of the result.

Command line arguments provide a way to adjust the inner workings of the surfel generator as needed.

The main part of future work would be an **external system** around this program. Such a system would take care of the **calls to the program** (setting its inputs and parameters) and then handle **produced outputs**. However, it is expected that this external system would be extremely dependent on the platform into which it should be integrated. Therefore, it was outside of this project's scope.

7.3.1 Conclusion

The submitted version of this project's program:

- supports all the **essential parts**,
- is fully **functional**,
- **improved the original implementation** of Blue Surfels in several aspects,
- and is architecturally prepared for **future extensions** and improvements.

This all seemed like a reasonable scope for the needs of this Master's thesis.

7.4 Presentation of results

This subchapter presents 2 selected 3D models – **sampled** and **rendered as surfels** using a debug renderer.

A lot of various 3D models were tested during the development. Two of them were selected for presentation of results. These models are sufficiently different in size, proportions, and overall shape to represent the most common use cases well.

7.4.1 Car

The car model was selected because it has a **uniform cuboid-like shape** with mostly **curved surfaces** (both convex and concave). Its mesh is pretty low-poly (~3.6k faces) for such an object.

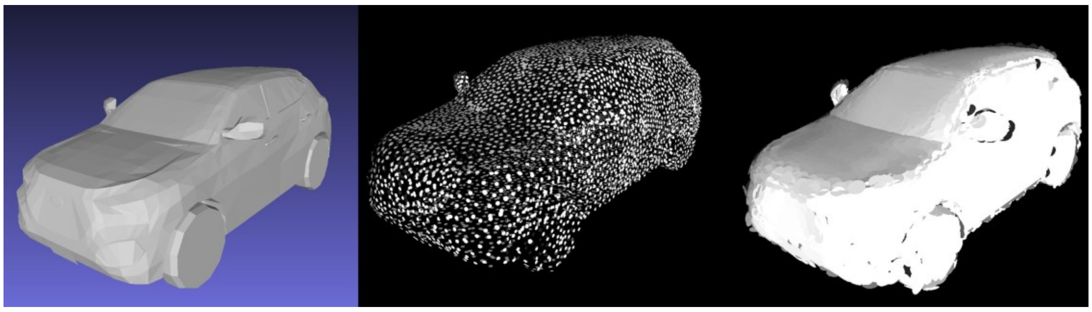


Figure 27 A 3D model of a car viewed as a triangular mesh rendered in MeshLab[24] on the left and the debug renderer visualizing the same model as sampled surfels on 2 right pictures. There are 10k surfels in both right pictures, just with varying sizes for easier visualization.

Firstly, Figure 27 shows a **comparison of triangular mesh and surfels**. Both surfel representations contain the same number of surfels, only the surfel size is different. Note that the middle picture from this figure is **not representative** of the

[24] https://www.meshlab.net/

intended final usage. The size of the surfels should always be selected so there are **no gaps** compared to the original mesh.

Also, rendering 10k surfels instead of the original mesh, which has only 3.6k faces, would make very little sense. This is only for **demonstrative purposes**. The **sampling of an ultra-high-poly** version of the same car model would produce a very **similar set of surfels** for rendering, making surfel representation more useful.

On the other hand, Figure 28 shows that even a **few hundred surfels** are enough to **approximate** the car model relatively well.

Again, such low numbers of surfels would be used only in case the model is really far from the camera. Usually, the size of the surfel would be selected so that its projection in the worst-case scenario is a **few pixels** on the screen. In these pictures, a single surfel is notably projected to thousands of pixels, if not more. The more realistic visualization is in Figure 29.

Nonetheless, this demonstrates the effect of **varying numbers of surfels** (even in extremes) on **visual quality** and **recognizability** compared to the original model.

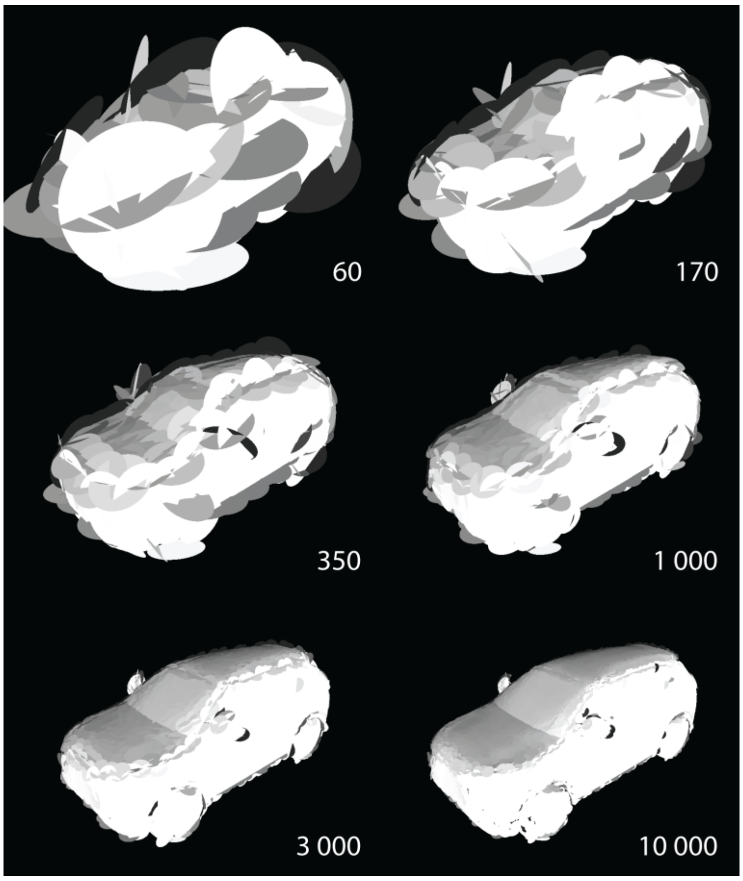


Figure 28 The same model of car rendered using a varying number of sampled surfels (number in the bottom right corner of each picture). From unusably low 60 surfels, through decently usable (at high distance from camera) hundreds of surfels, all the way up to 10 000 surfels.

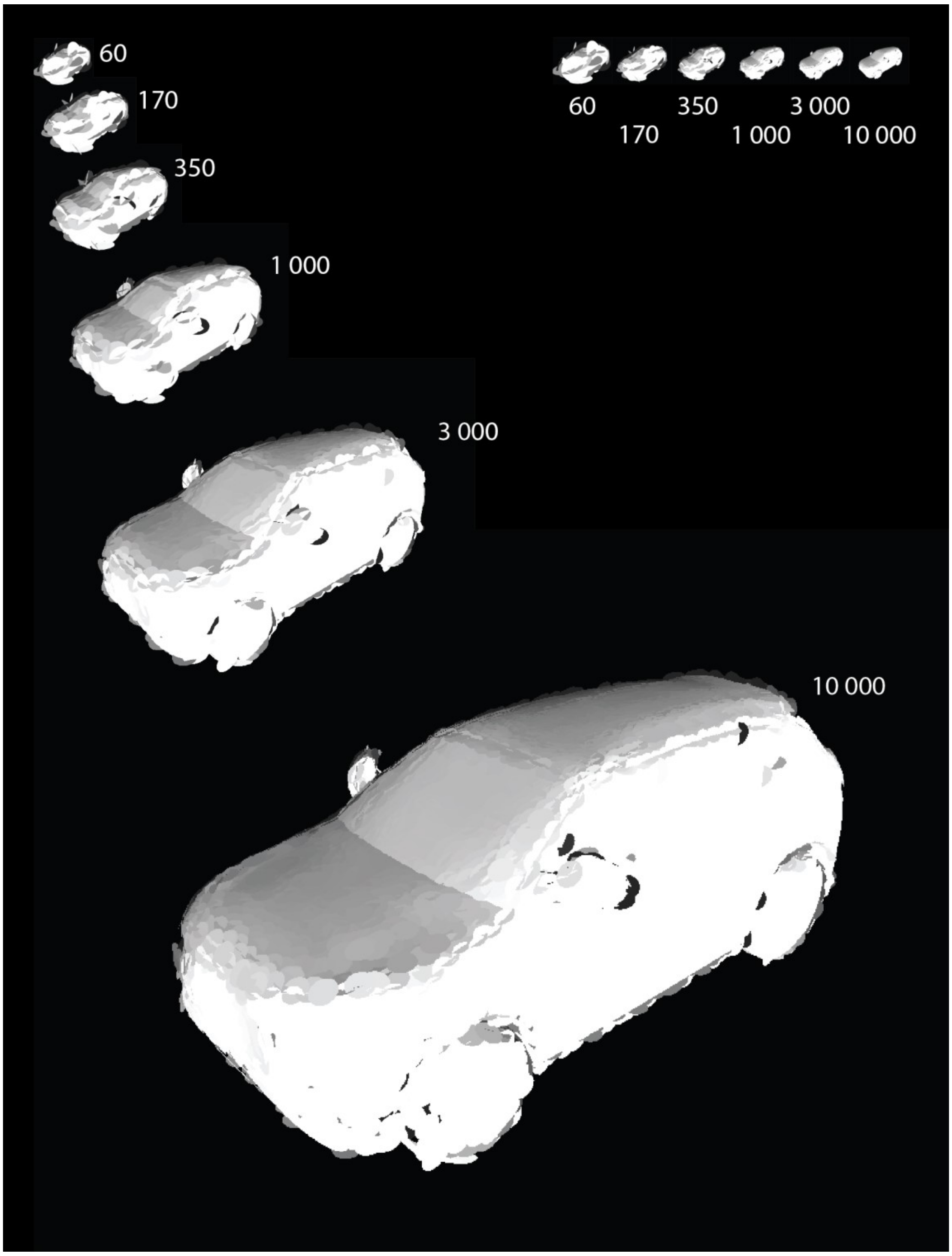


Figure 29 Comparison of the same models (same surfel counts) as in Figure 28. Displayed horizontally (upper right corner), there are all models next to each other scaled down to the same size (to simulate large distance from the observer). Vertically, each model is scaled to match the same surfel size (based on the most upper left model).

7.4.2 Crane

The second model selected for the presentation of results is a crane. Compared to the car model, the crane:

- is significantly **less uniform**,
- has marginally **more polygons** – about 130k, compared to less than 4k polygons used for car model,
- consists of mostly **sharp edges** (and very few curved surfaces),
- contains challenging parts such as **long thin rods** and **cables** (pendants)

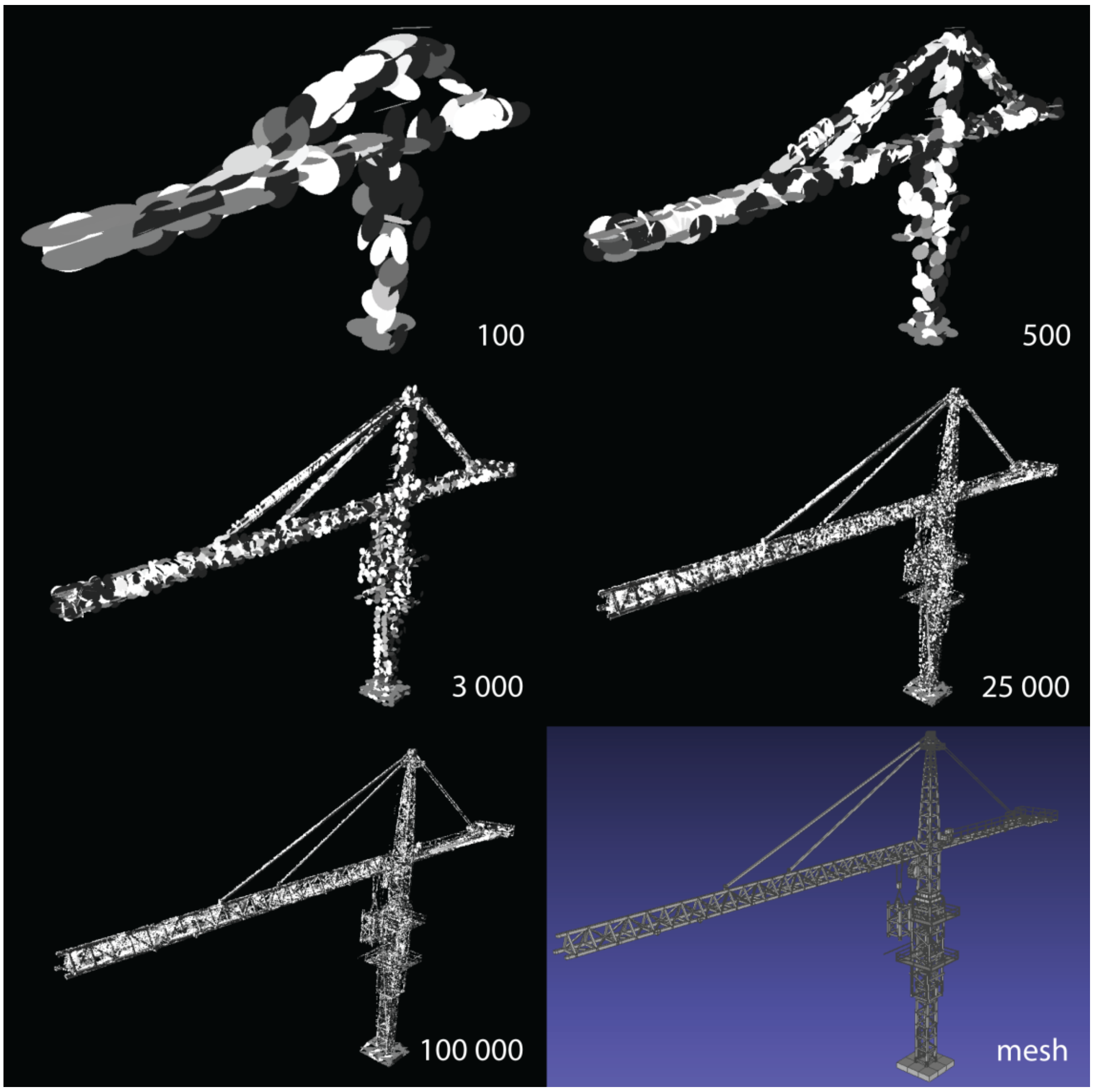


Figure 30 Model of crane composed of 100 all the way up to 100k surfels (number in the corner of each picture). The bottom right picture shows an original triangular mesh (~130k faces) rendered in MeshLab.

This model poses a great **challenge** with its very low uniformity and long, thin parts. **Pendants**, diagonal cables connecting the middle tower peak with the horizontal jib and counterjib, are a **dominant part** of any crane and are very important in its **recognition**. But they are also **thin** and, therefore, generally challenging for most LoD techniques.

As Figure 30 shows, even versions with an **extremely low** number of **surfels** (a few hundred) **preserve** these pendants very well, making the object easily **recognizable** as a crane from a distance.

It is important to note that these surfel representations show **artifacts** in the form of **high contrast** between neighboring surfels. This is partially due to the very primitive **lighting model** (Lambertian shading) used in the debug renderer but also because surfels appear way **too big** in low-surfel-count versions. This is an unrealistic scenario used only for demonstration purposes. Techniques such as EWA filtering (see Chapter 2.3.4) would greatly help this problem.

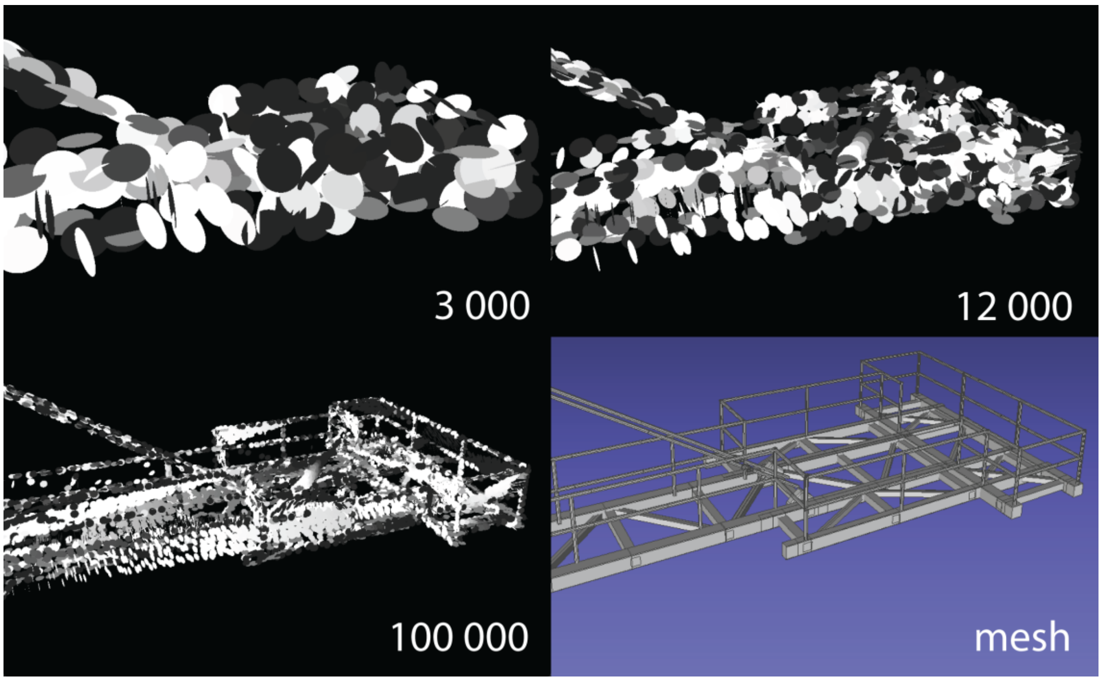


Figure 31 A close-up of the back platform on the crane from Figure 30. It contains pictures of both the original mesh rendered in MeshLab and the debug renderer using 3k, 12k, and all 100k surfels. The number of surfels refers to the total number on a whole crane, not just a visible part.

Figure 31 shows a close-up of a platform on a crane's counterjib. This spot was chosen to show **challenging parts** of the model, such as various rods and railings, in more detail. And as this figure shows, this technique can reconstruct even meshes like this one pretty well.

Low surfel counts (thousands of surfels) are not very usable in this scenario, but they would not be used in such a close-up anyway. In some cases, it might be a good idea to **split** such a big model as this crane into **multiple parts** and let each part **handle the LoD level independently**. This would make sense if an external program expected a camera to be located, for example, at the bottom platform of the crane (where original mesh or at least high surfel count should be used) and distant parts (top of the crane) could be rendered using just a few surfels.

7.4.3 Render times

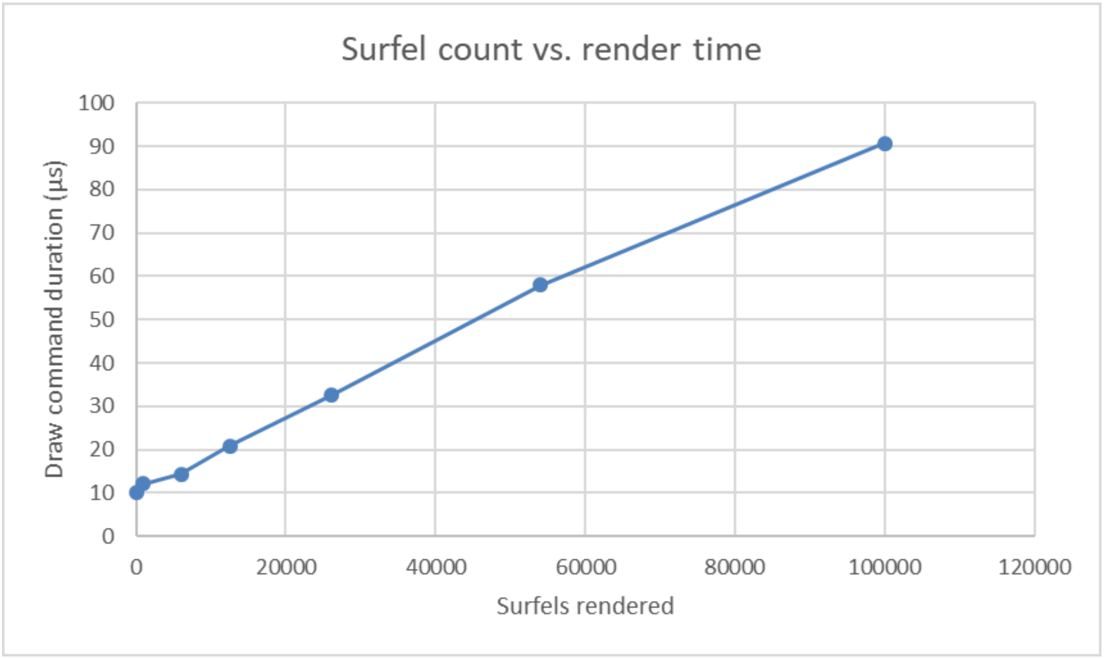


Figure 32 Dependency of number of rendered surfels and time for a single draw call (`vkCmdDraw`) in microseconds. Tested in debug renderer on a model of the crane. Times are calculated as an average of 5 independent frames, captured and analyzed using RenderDoc (Nvidia Nsight Graphics reported similar times). Measured standard deviations were insignificantly small.

Time to render the model is **linearly dependent** on number of surfels it is made of, as can be seen on Figure 32. This was tested in the debug renderer with **no optimizations** such as frustum or occlusion culling and with standard depth test enabled.

As can be seen, rendering **very small amounts** of surfels (the lowest tested number was ~150) is **not effective** since there is an **overhead** of about 10µs for a render call itself. For a comparison, rendering 1000 surfels took only 2µs longer, and rendering 6000 surfels took, on average, an additional 2µs. From that point onward, the change is more or less proportional.

7.4.4 Conclusion

The previous subchapters and their figures showed that this technique of mesh sampling has good results on both concave and convex **curved** surfaces, as well as surfaces with **sharp edges**. Moreover, generally **challenging meshes** such as long, thin rods/ropes are handled correctly as well.

Different **homogeneity** of the input model does not pose a problem for this sampling technique either. However, depending on the scenario, **big models** such as an example crane might need to be **split** into multiple parts, and each part sampled independently.

Limitations such as graphical artifacts were acknowledged and explained, and a solution for them was proposed.

Figure 28 together with Figure 30 show that as low as **few hundreds** of surfels can create a representation of a model which is **good enough** in case it is viewed from a relatively long distance. This, with the results presented in Figure 32, shows that these sampled models are **adequate for LoD needs**.

For example, rendering a **single model** consisting of 10k surfels (because it is close to the camera) is **equivalent** to rendering **10 instances** of the same model, each consisting of 1k surfels (since they are further away). Naturally, this example does not consider other optimization techniques, rendering approaches, overheads, etc.

7.5 Other uses of the project

The result of this project is not only an improved implementation of Blue Surfels. The critical part is also an extensive **foundation for Vulkan-based applications/renderers**.

As can be seen throughout the whole of Chapter 5, there was a strong focus on **scalable** and **extensible** architecture. Evidence of that is deep hierarchies of classes, from the most abstract one, through generic implementation (great as a base for other projects), all the way to concrete classes used for this project.

The minimal amount of **3rd party dependencies** also helps with the use as a base for other projects. Besides that, these 3rd party libraries were carefully selected with regard to **licenses** (so there are no issues with commercial use) and probability for long-term **support**.

7.6 Limitations

The following subchapters discuss some of the largest **limitations** of this project in its **current state**, with **justification** for the lack of these features.

7.6.1 Input format

This project, in its current state, has very strict limitations related to the format of input.

First of all, **only .obj files** are supported. This format for chosen for its (relative) **simplicity** of its implementation using 3rd party libraries and widespread **access to free data**. However, as mentioned, this is not very useful for the final implementation. Any external program using this project would have to implement its own import of models.

It is **not possible** to implement **fully universal system** for all cases. Not only it can be any **format** (even proprietary one) but it can be **imported in various ways** (as a pointer to CPU memory, file on drive, handlers to vertex and index buffers on GPU, etc.).

Not to mention that input might not be in the form of 3D model at all. Skipping sampling stage and providing **sampled data from different system** (such as 3D scanner) is a valid strategy. And again, creating universal system for this case would be extremely time consuming and irrelevant to the goals set for this project.

Second of all, even though .obj format is relatively simple, **supporting it fully is not easy**. There is far more than just positions and colors. Full support for .obj files would have to handle **various colors** (ambient, diffuse, specular), illumination models (reflections, …), **bump** and **displacement maps**, and much more.

With all that, a surprisingly high number of .obj files available online are **not fully correct**. There are often invalid values and missing MTL files with materials. Handling all of this is outside of the scope of this project.

The complexity of the full support is also one of the reasons the results presented in previous subchapters are all **greyscale**. Finding adequate and interesting .obj files with the correct colors and materials is not an easy task. However, the general **support for handling of color** in the rest of the algorithm was tested and worked without any issues.

7.6.2 Sampling of model interior

Currently, the sampling stage **samples the input model only on its surface**. In many cases, this is good enough. However, as mentioned in (Brandt, et al., 2019), some models would benefit from being sampled from inside as well.

This can be achieved using techniques such as **depth peeling**. In the context of this project, depth peeling would work on the principle of sampling **topmost** layer from each direction several times, while **rejecting** samples **already sampled** in the previous iteration. This would result in sampling **deeper and deeper layer** each time.

But all of this comes with a cost of **higher complexity**. Not only **computational** but also a complexity of **decision making** on which models should be sampled in depth and, especially, how deep should the sampling go.

More about depth peeling approaches in general can be found in, for example, (Bavoil, et al., 2008) or (Liu, et al., 2009).

7.7 Contributions to 3rd party software

This project works with several unusual techniques, such as the non-standard use of rendering pipeline and rare formats. This caused several issues with 3rd party software. These **issues** were adequately **reported**, which led to their **fix**.

7.7.1 RenderDoc

RenderDoc having an active community and creator came in handy when several issues were discovered in this graphics debugger.

More specifically, there was a lack of support for using both **geometry shaders** and **multi-view** rendering simultaneously. This caused buggy behavior of GUI (such as missing output in panels) and lack of any form of notification[25].

The second case was improper handling of **packing of vector** consisting of two 32-bit numbers into 64-bit and vice versa (`unpackUint2x32` and related methods) in **shader debugger**.

[25] GitHub issue: https://github.com/baldurk/renderdoc/issues/2595

The last encountered issue was also related to 64-bit numbers, which are rare in standard rasterization pipelines. The usage of **64-bit numbers in the vertex buffer** (when read in the vertex shader) caused the crash of RenderDoc.

All these issues led to proper bug reports and were quickly fixed by the creator.

7.7.2 Visual Studio

As part of the development of this project, a bug in Visual Studio was discovered and reported, which led to its successful fix[26]. This was a minor bug in **IEnumerable Visualizer** (debug window for `IEnumerable` classes). It involved an incorrect row index being displayed for selected rows. Before it was discovered, this bug caused several inconveniences since this feature was crucial in early development and debugging.

[26] https://developercommunity.visualstudio.com/t/IEnumerable-Visualizer-row-index-does-no/10130331

8 Discussion

This chapter contains a few **miscellaneous topics** that would not fit anywhere else. These topics are usually just **extensions** and **thoughts** on top of the whole project.

8.1 Selection of models for conversion to surfels

Not all models are suitable for conversion using this algorithm. This chapter discusses it, mainly from the point of the FataMorgana system.

CAD models are great candidates for conversion. They usually contain a **considerable number of triangles**, have **uniform colors** (and rarely any textures – see Figure 19), and are mostly **static**. Even if there are animations, it is usually just a simple translation/rotation of parts of the mesh and no morphing of the mesh itself. Therefore, this technique could be used as long as separate static sub-models are **converted separately**.

On the other hand, a **scanned environment** from HoloLens is not the best candidate for this. It is usually **low-poly**; therefore, the performance advantage of surfels would be relatively small. And, more importantly, these meshes are constantly **changing** (as the physical scene changes) and **improving** (with finer and finer detail).

The more complicated is the decision of whether to convert the environment scanned using **3D scanners** such as Leica BLK360. Output from such scanners can **vary in size** a lot depending on the device and settings. Also, it depends on the usage of the scanner itself. If the environment is scanned once and remains **static** after that, conversion to surfels could be used. But if the 3D scanner is still used to **gradually improve** and **extend** existing scans (similar to the case with HoloLens), it might be wise to wait with conversion.

8.2 Small triangle draw efficiency

Rasterizing many triangles smaller than **2x2 pixels** is extremely **ineffective**. This applies to the vast majority of standard modern GPU architectures. The authors of Nanite realized this[27] when working on their rendering optimization algorithm, which is based on rendering a huge number of small triangles. Their research concluded that primitive and mesh shaders could help, but **software rasterization** was still several times faster than standard GPU rasterization. Besides Nanite, similar work was presented in (Kenzel, et al., 2018).

The rendering of surfels outputted from the algorithm presented in this thesis could pose the **same challenges**. Software rasterization is not easy to implement effectively with all the required features, such as **depth tests**.

Moreover, **hardware rasterization** is still more effective for large triangles. This means that the selection process between software and hardware rasterization needs to be implemented. Additional care must be taken to prevent **pixel cracks** between these 2 rasterizers.

[27] https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf (starts at page 80)

9 Conclusion

Rendering for AR/VR devices is especially **demanding**. There are many rendering optimization techniques, each with its advantages and disadvantages.

Therefore, the first goal of this project was to **research** and find an adequate LoD algorithm for the needs of the FataMorgana platform. The second goal was to **implement** this algorithm with potential improvements and additions.

9.1 Research

The first part of this project, a research, consisted of an extensive **review** of existing literature in the field of rendering-optimization techniques, mostly based on the LoD approach. The focus was on:

- **novelty** – which usually means good performance and modern hardware utilization compared to older techniques,
- **scalability** – since the algorithm should help with the rendering of huge scenes in a vast range of devices, from powerful desktop computers to low-power standalone AR/VR headsets,
- and the **support for input format** – which is expected to be not only in the form of a triangular mesh but also a point cloud from a 3D scanner and other non-standard representations.

Based on the stated criteria, the best candidate for the LoD-based algorithm was selected to be a technique called **Blue Surfels**, as presented in (Brandt, et al., 2019).

9.2 Implementation

The use of an aging OpenGL graphics API, a lack of documentation, and a very strong coupling to its base rendering platform prevented the **existing implementation** of Blue Surfels from being used directly for this project's needs.

Therefore, it was decided that it would be a better idea to **completely rewrite** the existing program from scratch. This enabled the use of modern, multiplatform **Vulkan** API instead of OpenGL. Its main benefits are the ability for low-level optimization and full utilization of modern GPUs.

Besides that, while rewriting the whole algorithm, a great deal of focus was spent on creating an **extensible** and **scalable architecture**. The result is a mini **framework** consisting of a complex but flexible hierarchy of classes and many helper methods. This framework could be easily used as a base for (almost) any Vulkan-based application.

Although several aspects of this project and implementation decisions were influenced by its being primarily developed for Pocket Virtuality, it does **not directly depend** on any proprietary technology or code developed by this company. Also, only a minimum number of carefully selected **3rd party libraries** were used.

9.3 Results

An extensive **comparison** of related **literature** was conducted. As a result, the Blue Surfels algorithm was selected as best suited for the case. This algorithm was completely **rewritten** and **improved** in several ways, such as the utilization of modern technologies and improved readability and organization of code.

The resulting code, created as a part of this thesis, supports **all essential parts** of Blue Surfels algorithm. The structure of the code enables easy implementation of **extensions** for the future work. The important part is that the code is **fully functioning** from import of the model, through setting up parameters, all the way to producing expected output.

Besides the sampling algorithm itself, a **debug renderer** was created as a part of this project as well.

9.3.1 Limitations

An implementation part of this thesis is a type of project that can be extended for years. Therefore, it was important to set **reasonable limitations** and treat several potential improvements as being out of the project's scope and a future work.

This includes, for example, an **input format** being restricted to **.obj files**. In the case of the final implementation into an external system, inputs will heavily depend on the said system. Implementing this universally is very difficult and time consuming.

The other limitation is **graphical artifacts** in the debug renderer. This is due to its simplistic nature and intended purpose (a debugging of surfel placement). Its rendering is not representative of an expected rendering in the external system. Various techniques can be implemented to improve this.

In its current form, sampling of the model is done only on its **surface**. Blue Surfels project also implemented a **depth peeling** algorithm for sampling inside of the model. This does benefit some specific models. However, due to its implementation **complexity** and relatively **small output enhancement** for models expected in the FataMorgana platform, this was deemed to be outside the project's scope.

The external system using this project is also responsible for **splitting** input models as necessary, setting up desired **parameters**, and managing the whole **LoD selection process**.

9.3.2 Bug fixes in 3rd party software

A lot of **non-standard techniques** were used in this project. This led to the discovery of several **software bugs** in a graphics debugger called RenderDoc. These bugs were further investigated (thanks to the open-source nature of this software) and properly reported, which led to their fix. The same goes for a GUI bug in Visual Studio.

10 References

Bavoil Louis and Myers Kevin Order Independent Transparency with Dual Depth Peeling. - [s.l.] : NVIDIA Corporation, February 2008.

Brandt Sascha [et al.] Rendering of Complex Heterogenous Scenes using Progressive Blue Surfels [Journal]. - April 2019.

Brandt Sascha [et al.] Visibility-Aware Progressive Farthest Point Sampling on the GPU [Conference] // Computer Graphics Forum. - [s.l.] : The Eurographics Association and John Wiley & Sons Ltd., 2019. - Vol. 38. - pp. 413-424. - ISSN: 1467-8659.

Carpenter Loren The A -buffer, an antialiased hidden surface method [Conference] // SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques. - 1984. - pp. 103-108.

Clark James Henry Hierarchical Geometric Models for Visible Surface Algorithms [Journal] // Communications of the ACM. - New York, NY, USA : Association for Computing Machinery, October 1, 1976. - 10 : Vol. 19. - pp. 547-554. - ISSN: 0001-0782.

Coconu Liviu and Hege Hans-Christian Hardware-Oriented Point-Based Rendering of Complex Scenes [Conference] // EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering / ed. Debevec P. and Gibson S.. - [s.l.] : The Eurographics Association, 2002. - pp. 43-52. - ISBN: 1-58113-534-3.

Cohen Jonathan D., Aliaga Daniel G. and Zhang Weiqiang Hybrid simplification: Combining multi-resolution polygon and point rendering [Conference] // Proceedings Visualization, 2001. VIS '01.. - San Diego, CA, USA : IEEE, 2001. - pp. 37-539.

Derzapf Evgenij and Guthe Michael Dependency-Free Parallel Progressive Meshes [Conference] // Computer Graphics Forum. - [s.l.] : The Eurographics Association and Blackwell Publishing Ltd., 2012. - Vol. 31. - pp. 2288-2302.

Holst Mathias and Schumann Heidrun Surfel-Based Billboard Hierarchies for Fast Rendering of 3D-Objects [Conference] // 4th Symposium on Point Based Graphics, PBG@Eurographics 2007, Prague, Czech Republic, September 2-3, 2007 / ed. Botsch Mario [et al.]. - Prague, Czech Republic : Eurographics Association, 2007. - pp. 109-118.

Hoppe Hugues Progressive meshes [Conference] // Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. - New York, New York, USA : ACM Press, 1996. - pp. 99-108.

Hou Xueshi, Lu Yao and Dey Sujit Wireless VR/AR with Edge/Cloud Computing [Conference] // 2017 26th International Conference on Computer Communication and Networks (ICCCN). - Vancouver, BC, Canada : IEEE, 2017. - pp. 1-8.

Hu Liang, Sander Pedro Vieira and Hoppe Hugues Parallel view-dependent refinement of progressive meshes [Conference] // I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games / ed. Haines Eric [et al.]. - Boston, Massachusetts, USA : ACM, 2009. - pp. 169-176.

Chang C.-F., Bishop G. and Lastra A. LDI tree: a hierarchical representation for image-based rendering [Conference] // Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '99. - 1999. - pp. 291–298.

Ip Cheuk Yiu [et al.] PixelPie: maximal Poisson-disk sampling with rasterization [Conference] // Proceedings of the 5th High-Performance Graphics Conference. - New York, NY, USA : Association for Computing Machinery, 2013. - Vol. 13. - pp. 17-26. - ISBN: 9781450321358.

Jähn Claudius Progressive Blue Surfels [Journal]. - 2013.

Kenzel Michael [et al.] A high-performance software graphics pipeline architecture for the GPU [Article] // ACM Transactions on Graphics. - New York, NY, USA : Association for Computing Machinery, July 30, 2018. - 4 : Vol. 37. - pp. 1-15. - ISSN: 0730-0301.

Kerbl Bernhard [et al.] 3D Gaussian Splatting for Real-Time Radiance Field Rendering [Journal] // ACM Transactions on Graphics. - [s.l.] : ACM, July 2023. - 4 : Vol. 42.

Lischinski Dani and Rappoport Ari Image-Based Rendering for Non-Diffuse Synthetic Scenes [Conference] // Rendering Techniques '98, Proceedings of the Eurographics Workshop / ed. Drettakis George and Max Nelson L.. - Vienna, Austria : Springer, 1998. - pp. 301–314.

Liu Baoquan, Wei Li-Yi and Xu Ying-Qing Multi-Layer Depth Peeling via Fragment Sort [Conference] // 11th IEEE International Conference on Computer-

Aided Design and Computer Graphics. - Huangshan, China : IEEE, 2009. - ISBN: 978-1-4244-3699-6.

Maggiordomo Andrea, Moreton Henry and Tarini Marco Micro-Mesh Construction [Conference] // ACM Transactions on Graphics. - New York, New York, USA : ACM, 2023. - Vol. 42. - pp. 1-18.

Mildenhall Ben [et al.] NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis [Journal] // Communications of the ACM. - [s.l.] : ACM, 2020. - 1 : Vol. 65. - pp. 99-106.

Munguba Gabriel [et al.] The complex build algorithm to set up starting structures of lanthanoid complexes with stereochemical control for molecular modeling [Journal] // Scientific Reports. - November 2021. - Vol. 11. - Article number: 21493.

Pajarola Renato Efficient level of details for point-based rendering [Conference] // Proceedings of the Sixth IASTED International Conference on Computer Graphics and Imaging. - Honolulu, Hawaii, USA : IASTED/ACTA Press, 2003. - pp. 141-146.

Pauly Mark, Gross Markus and Kobbelt Leif P. Efficient Simplification of Point-Sampled Surfaces [Conference] // IEEE Visualization, 2002. VIS 2002. - Boston, MA, USA : IEEE, 2002. - ISBN: 0-7803-7498-3.

Pfister Hanspeter [et al.] Surfels: Surface Elements as Rendering Primitives [Conference] // SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques. - [s.l.] : ACM Press/Addison-Wesley Publishing Co., 2000. - pp. 335-342. - ISBN: 1581132085.

Ren Liu, Pfister Hanspeter and Zwicker Matthias Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering [Conference] // Computer Graphics Forum / Eurographics. - Saarbrücken, Germany : [s.n.], 2002. - Vol. 21. - pp. 461-470.

Rusinkiewicz Szymon and Levoy Marc QSplat: A Multiresolution Point Rendering System [Conference] // SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques. - [s.l.] : ACM, 2000. - pp. 343-352.

Satish Nadathur, Harris Mark and Garland Michael Designing efficient sorting algorithms for manycore GPUs [Conference] // 2009 IEEE International

Symposium on Parallel & Distributed Processing. - Rome, Italy : IEEE, 2009. - pp. 1-10.

Shade Jonathan [et al.] Layered depth images [Conference] // SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques. - New York, New York, USA : ACM, 1998. - pp. 231–242.

Shi Bao-Quan, Liang Jin and Liu Qing Adaptive simplification of point cloud using point cloud using k-means clustering [Journal] // Computer-Aided Design. - [s.l.] : Elsevier, August 2011. - 8 : Vol. 43. - pp. 910-922.

Stenning Justin Direct3D Rendering Cookbook [Book]. - [s.l.] : Packt, 2014. - ISBN: 9781849697101.

Ulichney Robert A. Dithering with blue noise [Conference] // Proceedings of the IEEE. - [s.l.] : IEEE, 1988. - Vol. 76. - pp. 56-79.

Wilson Michael Lee The Effect of Varying Latency in a Head-Mounted Display on Task Performance and Motion Sickness [Journal] // All Dissertations. - [s.l.] : TigerPrints, 2016.

11 List of figures

12 List of abbreviations

- ***n*D (1D, 2D, 3D)** – *n*-dimensional
- **AABB** – Axis-Aligned Bounding Box
- **API** – Application Programming Interface
- **AR** – Augmented Reality
- **CAD** – Computer-Aided Design
- **CAVE** – Cave Automatic Virtual Environment *(recursive acronym)*
- **CLR** – Common Language Runtime
- **CPU** – Central Processing Unit
- **EXT** – Extension
- **FM** – FataMorgana
- **FPS** – Frames Per Second
- **GLFW** – Graphics Library Framework
- **GLSL** – OpenGL Shading Language
- **GPU** – Graphics Processing Unit
- **GPGPU** – General Purpose GPU
- **GUI** – Graphical User Interface
- **HLSL** – High-Level Shading Language
- **IDE** – Integrated Development Environment
- **KHR** – Khronos
- **LDI** – Layered Depth Image
- **LoD** – Level of Detail
- **MIT** – Massachusetts Institute of Technology
- **MSAA** – Multisample Anti-Aliasing
- **NeRF** – Neural Radiance Field
- **OS** – Operating System
- **PADrend** – Platform for Algorithm Development and Rendering
- **PLY** – Polygon File Format
- **SDL** – Simple DirectMedia Layer
- **SLI** – Scalable Link Interface
- **SPIR-V** – Standard Portable Intermediate Representation - Vulkan

- **TBD** – To Be Done *(if you are seeing this anywhere else in the final document, something went wrong)*
- **UBO** – Uniform Buffer Object
- **VK** – Vulkan
- **VR** – Virtual Reality
- **WDDM** – Windows Display Driver Model

13 Attachments

13.1 Attachment 1 - Source code

This thesis's first and only attachment is an archive file containing source code. This includes all **3 projects**: FMsurfelsVulkan, FMsurfelsDebugTools, and abandoned FMsurfelsDirectX. The first 2 are accessible through the **FMsurfels.sln** solution file. For more details about the organization of files, see Chapter 5.1.

As a bonus, there are **ClassDiagram<1,2>.cd** files under the FMsurfelsVulkan folder. A component called "Class Designer" for Visual Studio must be installed to open them. (This is an official component installed through Visual Studio Installer, not an extension.) These files contain a hand-picked and carefully arranged set of the classes/structs with their most important properties and methods.

For the **testing purposes**, see OBJ files in `FMsurfelsVulkan/Models` folder and `ObjFileName` command line argument (`-i` or `--inputFileName`).