Informatics

In-Core Level-Of-Detail Generation For Point Clouds On GPUs Using CUDA

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Philip Klaus, BSc
Matrikelnummer 01407087

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Projektass. Dipl.-Ing. Dr.techn. Markus Schütz, B.Sc.

Wien, 30. Oktober 2023

Philip Klaus   Michael Wimmer

Informatics

In-Core Level-Of-Detail Generation For Point Clouds On GPUs Using CUDA

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Philip Klaus, BSc
Registration Number 01407087

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. Dipl.-Ing. Dr.techn. Markus Schütz, B.Sc.

Vienna, 30th October, 2023

Philip Klaus Michael Wimmer

Erklärung zur Verfassung der Arbeit

Philip Klaus, BSc

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

Wien, 30. Oktober 2023

Philip Klaus

Danksagung

Ich möchte mich hiermit bei all jenen Personen bedanken, die mir diese Arbeit ermöglicht und mich dabei auch laufend unterstützt haben. Es freut mich sehr, dass ich meine Masterarbeit an der TU Wien in Kooperation mit meiner Arbeitsstelle, dem AIT Austrian Institute of Technology GmbH durchführen konnte. Auf Seiten der TU Wien möchte ich mich bei meinem Betreuer Michael Wimmer bedanken, der mir diese Masterarbeit erst ermöglichte. Außerdem möchte ich ihm für sein schnelles Korrekturlesen meiner schriftlichen Arbeit danken. Weiters gilt mein Dank meinem zweiten Betreuer Markus Schütz – dem Entwickler von Potree – insbesondere für das Zurverfügungstellen von 3D Daten und die anregenden inhaltlichen Diskussionen. Auf Seiten des AITs möchte ich mich bei meinem Vorgesetzten Markus Clabian bedanken, der es mir ermöglichte meine Masterarbeit kooperativ mit dem AIT durchzuführen. Außerdem möchte ich mich bei Nicole Brosch für die Betreuung auf Seiten des AITs und ebenfalls für ihr Korrekturlesen bedanken. Abschließend möchte ich mich bei meiner Familie und insbesondere bei meiner Lebensgefährtin für die mentale Unterstützung während meines Studiums und meiner Masterarbeit bedanken.

Acknowledgements

I would like to thank all those people who made this work possible and also supported me continuously. I am glad that I could do my master thesis at the TU Wien in cooperation with my workplace, the AIT Austrian Institute of Technology GmbH. On the part of the TU Wien I would like to thank my supervisor Michael Wimmer, who made this master thesis possible for me in the first place. I would also like to thank him for his quick proofreading of my written work. Furthermore, I would like to thank my second supervisor Markus Schütz – the developer of Potree – especially for providing 3D data and stimulating discussions about the content. On the part of AIT, I would like to thank my supervisor Markus Clabian, who made it possible for me to conduct my master's thesis cooperatively with AIT. Furthermore, I would like to thank Nicole Brosch for her supervision on the AIT side and also for her proofreading. Finally, I would like to thank my family and especially my partner for the mental support during my studies and my master thesis.

Kurzfassung

In dieser Masterarbeit stelle ich *PotreeConverterGpu* vor, eine GPU-basierte Software zur Erzeugung von Octree-basierten *Level-Of-Detail* Strukturen (LOD) aus Punktwolken. Die resultierenden LODs sind vollständig kompatibel mit *Potree* [Sch16], einem Punktwolken-Renderer zur Darstellung von großen Punktwolken im Webbrowser.
In der Vergangenheit wurden bereits verschiedene Algorithmen zur Erzeugung von Punktwolken-LODs vorgeschlagen. Obwohl diese Lösungen unterschiedliche Schwerpunkte, Vor- und Nachteile hatten, zielten sie alle auf einen hohen Punktedurchsatz, sowie eine hohe visuelle Qualität der erzeugten Detailabstraktionen ab. Die implementierten LOD-Generierungsalgorithmen in dieser Masterarbeit basieren hauptsächlich auf *PotreeConverter* [SOW20], wurden aber vor allem in Bezug auf den Punktedurchsatz und der visuellen Qualität der generierten LODs verbessert. PotreeConverterGpu erreicht einen Punktedurchsatz, der - abhängig von der gewünschten visuellen Qualität der generierten LODs - 13-mal (höchste Qualität) bis 85-mal (niedrigste Qualität) höher ist als bei PotreeConverter 2.x. Diese Leistungssteigerung ist das Ergebnis einer hochoptimierten GPGPU-Implementierung auf der Basis von CUDA. Während LODs die mit der aktuellen PotreeConverter-Implementierung erzeugt werden unter Aliasing-Artefakten leiden, löst PotreeConverterGpu dieses Problem, indem die Software eine abstandsbasierte Farbfilterung für Punktwolken bereitstellt. Dies ermöglicht hochwertige Punktwolken-Renderings mit deutlich verminderten Aliasing-Artefakten, bei gleichzeitiger Beibehaltung von Texturdetails.
Aufgrund seiner wohldefinierten Software-API und Architektur ermöglicht PotreeConverterGpu außerdem eine einfache Integration der LOD-Generierung für Punktwolken in industriellen Anwendungen. Ein Beispiel für eine solche Applikation ist Inline Computational Imaging [GBT22] (ICI), eine Hochgeschwindigkeits- und Hochpräzisions-3D-Sensortechnologie des AIT Austrian Institute of Technology GmbH. Um eine ICI-kompatible LOD-Generierung zu ermöglichen, wurde diese Masterarbeit in Kooperation mit dem AIT durchgeführt.

Abstract

In this master's thesis, I present *PotreeConverterGpu*, a GPU-based software for generating octree-based *Level-Of-Detail* Structures (LOD) structures from point clouds. The resulting LODs are fully-compatible with *Potree* [Sch16] a point cloud renderer for viewing large point clouds in the web browser.
In the past, several different point cloud LOD generation algorithms have been proposed. Although these solutions had different focuses, advantages, and disadvantages, they all aimed at providing a high point throughput as well as a high visual quality of the generated detail abstractions. In this thesis, the implemented LOD generation algorithms rely mainly on *PotreeConverter* [SOW20] but have been improved especially in terms of point throughput and the visual quality of the generated LODs. PotreeConverterGpu achieves a point throughput that is – depending on the desired visual quality of the generated LODs – 13 times (highest quality) up to 85 times (lowest quality) higher than within PotreeConverter 2.x. This performance improvement is the result of a highly-optimized GPGPU implementation based on CUDA. While LODs generated with the current PotreeConverter implementation suffer from aliasing artifacts, PotreeConverterGpu solves this problem by providing distance-based color filtering for point clouds. This enables high-quality point cloud renderings with significantly reduced aliasing artifacts while preserving textural details.
Due to its well-defined software API and architecture, PotreeConverterGpu also enables easy integration of LOD generation for point clouds into industrial applications. One example for such an application is *Inline Computational Imaging* [GBT22] (ICI), a high-speed and high-precision 3D sensing technology of the AIT Austrian Institute of Technology GmbH. To provide an ICI-compatible LOD generation, this master thesis was conducted in cooperation with the AIT.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Over the past two decades, numerous 3D point-cloud rendering solutions have been proposed. Due to limitations in hard- and software, the real-time rendering of point clouds with millions of points was previously only possible using standalone desktop applications. However, the development of 3D capable mobile devices and technologies like *WebGL* [Grob] or *WebGPU* [Con] now enable developers to stream and render large amount of points to virtually any device with a web browser. The advantage of rendering point clouds directly in the browser is that it is device and platform independent and requires no installation. Thus, web-based viewing of point clouds is used in several applications, such as product presentations, geological surveys (e.g., *Entwine* [Sur] or *Sketchfab* [Ske]), or for viewing 3D reconstructions in an industrial setting (e.g., *ICI 3D data* [GBT22]). To this end, several web-based point-cloud viewers with real-time capabilities have been proposed, e.g., *Potree* [Sch16], *Plasio* [Ver] and *Leica Tru View* [Geo]. They impress with incredible viewing speeds and an immediate graphical response, but they typically also take trade-offs considering the quality of the viewed data (e.g., aliasing artifacts, decimated or lower LOD models, etc.). To be able to stream and render huge point clouds in real time, it is necessary to pre-process the original clouds into an LOD structure beforehand. This conversion is performed with time-consuming processing steps and hence, can be considered as a bottleneck in a user's workflow.

The fundamental goal of this master thesis is to increase the accessibility of real-time web-based viewing of detailed 3D point clouds in two aspects. First, we propose a CUDA-based LOD generation process that reduces preprocessing times and simultaneously increases visual quality by applying color filtering in lower levels of detail. Second, the usage in an industrial context shall be facilitated by providing a flexible and easy-to-integrate software library that is capable of converting 3D reconstruction data such as *Inline Computational Imaging* (ICI) 3D data [GBT22].

1.2 Problem Statement

This thesis addresses the problem of efficiently converting unstructured point clouds to LOD structures that can be viewed and explored in web browsers. The basis for the approaches in this master thesis lies in the *PotreeConverter* [SOW20], which prepares point clouds for the *Potree* [Sch16] point cloud renderer.

Potree is a high-performance point-cloud renderer that is capable of displaying datasets with up to hundreds of billions points of in real time in a web browser. Potree makes this possible by not displaying all points at once, but only those that are currently needed. Based on the viewport and distance between the point cloud and the camera, only those points are loaded and rendered that are required to provide the user with a satisfactory representation of the entire point cloud. According to this, if the camera-to-point-cloud distance is de-/ or increased, more points (and thus more details) or fewer points are displayed. In order to make it possible to show and hide certain details, it is first necessary to convert point clouds into an *LOD* (*Level-of-Detail*) structure. Such a structure consists – as the name suggests – of different hierarchical levels, where each higher level contains a more detailed point-subset of the original point cloud, showing more or less detail. Another advantage of transforming point clouds into an LOD structure is that Potree can download and render only sub-parts of the cloud with a desired resolution independently. Thus, Potree can provide a quick initial graphical response to the user with point-cloud resolution increasing over time as higher LODs are streamed in and added to lower LOD data. [Sch16] [SOW20]

However, the generation of these LOD structures is a time- and memory-consuming, computationally intensive process. Depending on the area of application, point clouds may consist of up to billions of points, resulting in large amounts of data and requiring a scalable, memory-efficient and fast method for LOD generation. Although PotreeConverter is highly optimised, there is still potential to speed up the entire conversion process since it runs on the CPU. Even though certain parts of the code are implemented in parallel, a large part of the software still works sequentially, which leads to a bottleneck. Another common problem is the quality of the converted point cloud data, in particular in relation to aliasing artifacts caused by the lack of color filtering. In contrast to textured meshes, where color filtering is well studied, color filtering for point clouds turns out to be more difficult due to the lack of a two-dimensional neighbourhood relation between points, which makes approaches such as mip mapping infeasible (commonly applied anti-aliasing strategies for point clouds are described in Section 2.3.3). The aliasing artifacts are caused by a sampling rate that is too low, i.e., when creating lower LODs, primarily too little color information is encoded in the lower levels. Increasing the sampling rate would counteract this problem, but would also increase the amount of points in lower LODs, which would introduce a longer response time during rendering. Although Potree applies high-quality splatting [BSHZK05] – a method for reducing aliasing artifacts during rendering – the artifacts are still perceptible.

The usage of web-based 3D point-cloud viewers in an industrial context is often hindered by the fact that it is not possible to integrate the necessary point-cloud LOD generation tools directly into proprietary industrial applications. One example for such an application is Inline Computational Imaging (ICI), which is a novel single-sensor technology for 2D / 3D inline inspection (see Section 2.2.1), developed by AIT Austrian Institute of Technology GmbH. In the case of ICI, the rendering of reconstructed 3D point clouds should be shifted from traditional desktop-based software to a web-based solution, more precisely to Potree. The problem is that PotreeConverter does not expose an API and thus, cannot be simply integrated into the ICI processing pipeline. Moreover, ICI's processing pipeline is implemented in CUDA and is executed entirely on the GPU. In order to be able to generate Potree-compatible LOD data within ICI efficiently while avoiding costly GPU-to-CPU memory transfers, the LOD generation would also have to be carried out on the GPU. Thus, this master thesis is conducted in cooperation with AIT to solve these problems and to provide a software library which can be easily integrated into ICI and which converts 3D point clouds to LOD data entirely on the GPU.

1.3 Contribution

We propose *PotreeConverterGpu*, an in-core GPGPU-based implementation of PotreeConverter that improves the previous LOD generation algorithm for Potree in terms of runtime and visual quality of the resulting multi-resolution point clouds. Further, PotreeConverterGpu was designed to be easily integrated into third-party software for automatic LOD generation. Basically, PotreeConverterGpu proposes the following contributions:

- A highly-optimized LOD generation for point clouds based on a replacement scheme using GPGPU-programming (CUDA). For this, the LOD generation algorithms based on [SOW20] have been re-designed to process point clouds on the GPU in a fully-parallelized manner. This enables processing of point clouds residing either in CPU (host) or GPU (device) memory with an increased point throughput in LOD generation ranging from 64M points (highest quality) to 418M points (lowest quality) per second. Compared to PotreeConverter 2.1, PotreeConverterGpu has a throughput that is 13 times or 85 times higher.
- Color filtering for point clouds. Based on suggestions from Rusinkiewicz and Levoy [RL01] and Wand et al. [WBB$^+$08], PotreeConverterGpu significantly improves the visual quality of the generated multi-resolution point clouds. In detail, a baked-in anti-aliasing strategy based on color filtering (similar to mip-mapping) is applied during point subsampling. We distinct between inter-cell color filtering, considering more and intra-cell color filtering, considering less neigbouring points. Both approaches significantly minimize high-frequent color noise and reduce the

occurrence of aliasing artifacts while preserving texture details. Especially in combination with the complementary high-quality splatting approach [BSHZK05], this effect becomes visible when the converted point clouds are rendered in Potree. Additionally, inter-cell color filtering allows to calculate distance-weighted color averages, further improving the preservation of textural details.

1.4 Thesis Outline

The thesis is organised in 5 chapters. This chapter discussed the fundamental motivation, the problem statement and the contributions of this thesis. The remaining chapters are briefly described below:

- **Fundamentals and State-of-the Art** (Chapter 2)
  This section introduces the topics of 3D reconstruction and point cloud rendering. In particular, Inline Computational Imaging, GPGPU programming and the current state of Potree and PotreeConverter are addressed. In this context, relevant algorithms and data structures are also covered.

- **Software Architecture** (Chapter 3)
  The goal of this chapter is to provide a basic overview of the proposed and improved LOD generation approach from a software engineer's perspective. This helps to get the big picture of the goals and concepts in this master thesis before getting into detailed information about the applied algorithms.

- **Generating an LOD structure on the GPU** (Chapter 4)
  This is the main part of the master thesis. It comprises detailed descriptions of all relevant aspects and steps regarding the LOD generation. This includes implemented algorithms as well as data structures, necessary for giving a clear and comprehensible explanation of the LOD generation.

- **Results and Evaluation** (Chapter 5)
  Beside different measurements (runtime, memory consumption, etc.), this chapter also contains a discussion explaining different interrelationships within the LOD generation. Special attention is paid to different subsampling configurations and their effects on runtime, memory consumption, result size, visual LOD quality etc.

- **Conclusion and Future Work** (Chapter 6)
  This last chapter provides suggestions for improvement of the LOD generation in terms of *quality* and *quantity*. *Quality* targets the actual quality of the generated multi-resolution point clouds with respect to anti-aliasing and point distribution, whereas *quantity* targets different LOD generation metrics, such as runtime.

CHAPTER 2

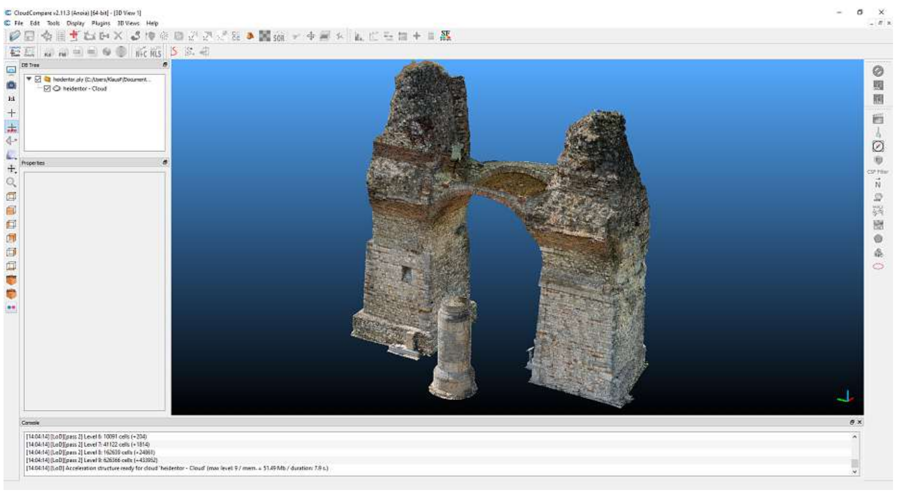
Fundamentals and State-of-the Art of LOD generation

This chapter gives an overview of existing point cloud rendering algorithms and viewers and, if applicable, their LOD generation methods. Special emphasis is put on a specific web-based point cloud viewer and converter, i.e., *Potree* and *PotreeConverter* [Sch16], which are the basis of this thesis.

2.1 State-of-the-Art of Point Cloud Rendering

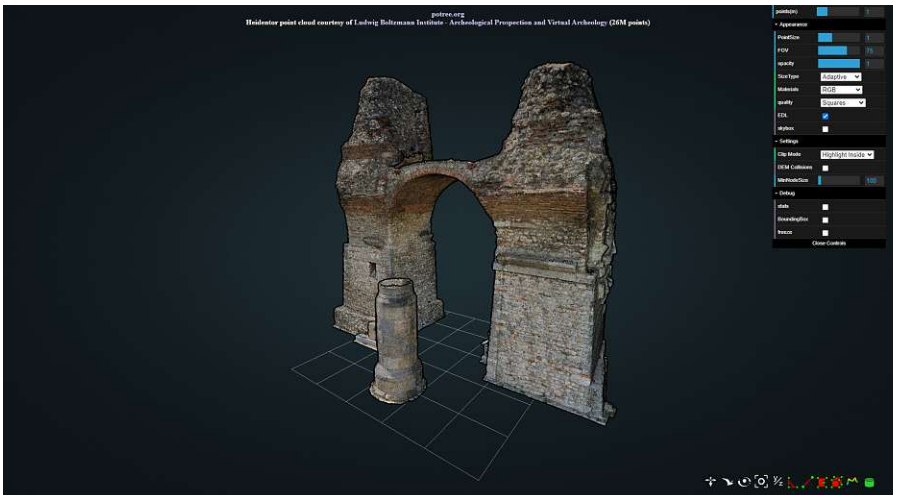
In general, there are several possibilities to view and explore 3D point clouds. An obvious choice is *standalone desktop software* such as *CloudCompare* [GM] which is available as open source. As point cloud visualisation becomes more and more popular, 3D editing programs such as *Blender* [Fou] or *3D Studio Max* [Aut] nowadays also offer the possibility to render point clouds in addition to conventional mesh data. Moreover, point clouds can also be used in established game engines such as *Unity* [Tec] or *Unreal Engine* [Gam]. As those programs are installed directly on a PC or workstation, they can efficiently utilise all of the computer's resources and can therefore provide high load and render performance. The disadvantages, on the other hand are, that stand-alone solutions are often not available for multiple platforms and that they require an installation.

Web-based point cloud viewing is a light-weight alternative to conventional standalone software. Since web-based renderers do not require any installation and run purely in the web browser, they are completely platform-independent and enable an unprecedented possibility of distributing interactive visualizations to the widest possible audience. Nowadays, there exist several web-based point cloud rendering solutions, many of them are proprietary software like *Faro Scene Web Share* [FAR] or *Leica Tru View* [Geo], whereas only a few of them are provided under an open-source license, e.g. *Potree* [Sch16], *Plasio* [Ver] or *Cesium* [CG].



```
[14:04:14] [LoD][pass 2] Level 6: 10091 cells (+204)
[14:04:14] [LoD][pass 2] Level 7: 41122 cells (+1814)
[14:04:14] [LoD][pass 2] Level 8: 162639 cells (+24861)
[14:04:14] [LoD][pass 2] Level 9: 626366 cells (+433952)
[14:04:14] [LoD] Acceleration structure ready for cloud 'heidentor - Cloud' (max level: 9 / mem. = 51.49 Mb / duration: 7.9 s.)
```

(a) The Heidentor data set rendered in CloudCompare [GM].



potree.org
Heidentor point cloud courtesy of Ludwig Boltzmann Institute - Archeological Prospection and Virtual Archeology (26M points)

```
points(m)        1
▾ Appearance
PointSize        1
FOV              75
opacity          1
SizeType         Adaptive
Materials        RGB
quality          Squares
EDL              ☑
skybox           ☐
▾ Settings
Clip Mode        Highlight Inside
DEM Collisions   ☐
MinNodeSize      100
▾ Debug
stats            ☐
BoundingBox      ☐
freeze           ☐
Close Controls
```

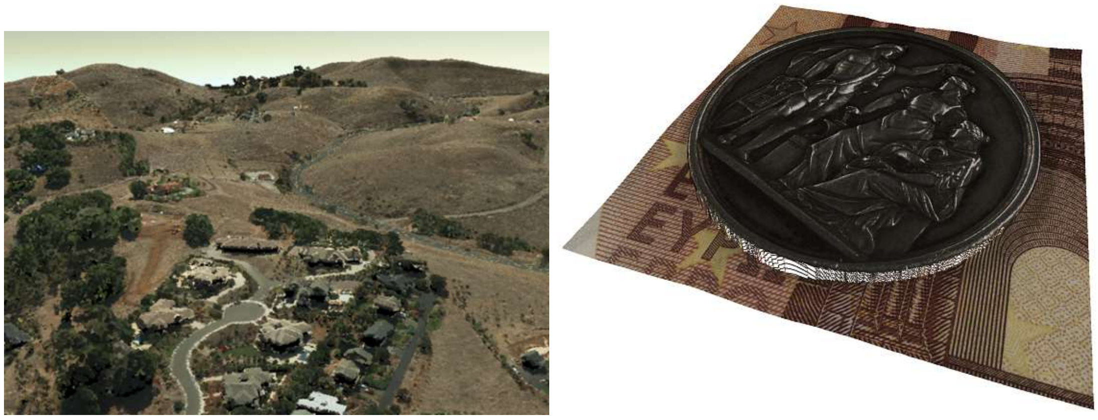
(b) The Heidentor data set rendered in Potree [Sch16].

Figure 2.1: Point cloud visualisation: a dataset from the Heidentor in Carnuntum (Austria), provided by the Ludwig Boltzmann Institute (LBI ArchPro) [Insb], visualised in CloudCompare [GM] (standalone application) and in Potree [Sch16] (web-based renderer)

2.2 Applications of Point Cloud Rendering

Point cloud rendering plays an important role in various fields. The discrete mathematical representation of three-dimensional objects using a finite number of points, enables a wide range of applications, ranging from research over industrial to geological ones. Due to the fact that several 3D scanning technologies produce very large point clouds (up

[1]PotreeDesktop [Sch] is a portable desktop version of the Potree [Sch16] web-based point cloud renderer. At the time of writing this thesis, the latest PotreeDesktop version is 1.8.0.



(a) ArcGIS scene. Image taken from [Insa]. **(b)** ICI 3D coin data set

Figure 2.2: Point cloud rendering applications:
a) An ARCGis [Insa] point cloud scene layer.
b) The ICI [GBT22] 3D coin data with 5.3 million points rendered in PotreeDesktop [Sch] [1]. The reconstruction was performed based on an ICI acquisition with a sampling of approximately 20μm/pixel. The coin data set is available on the project GitHub repository [Kla].

to billions of points), there is a need for point cloud rendering applications which can handle and display these amounts of points efficiently while preserving high quality rendering results. One of these technologies for acquiring large point clouds, mostly used in a geological context is *Light Detection And Ranging* (LIDAR). The resulting point clouds can consist of millions or even billions of points and thus pose a great challenge in terms of data storage, streaming and rendering. LIDAR scans are used, for example, in archaeological research or for the visualization of *Geographic Information System* (GIS) data. One example for such a GIS application is ArcGIS [Insa] which can be seen in Figure 2.2a. On the other hand there exist also several image based 3D reconstruction methods which fall into the category of photogrammetry. The goal of photogrammetric methods is to obtain the 3-dimensional surface structure of an object via images from multiple viewpoints. For instance, classical stereo-based 3D reconstrucion approaches extract depth information about an object by correlating two or more rectified images of this object [Sze11]. To obtain better 3D results, several image-based reconstruction approaches can also be combined. For example, for the ICI [GBT22] setup and software, we use light field technology and photometric stereo to perform depth estimation of surfaces in an industrial environment (see Figure 2.2b). This setup is explained in the next section 2.2.1.

2.2.1 ICI - Inline Computational Imaging

This master thesis is conducted in corporation with the AIT Austrian Institute of Technology GmbH. We at AIT have developed *Inline Computational Imaging* (ICI), a high-

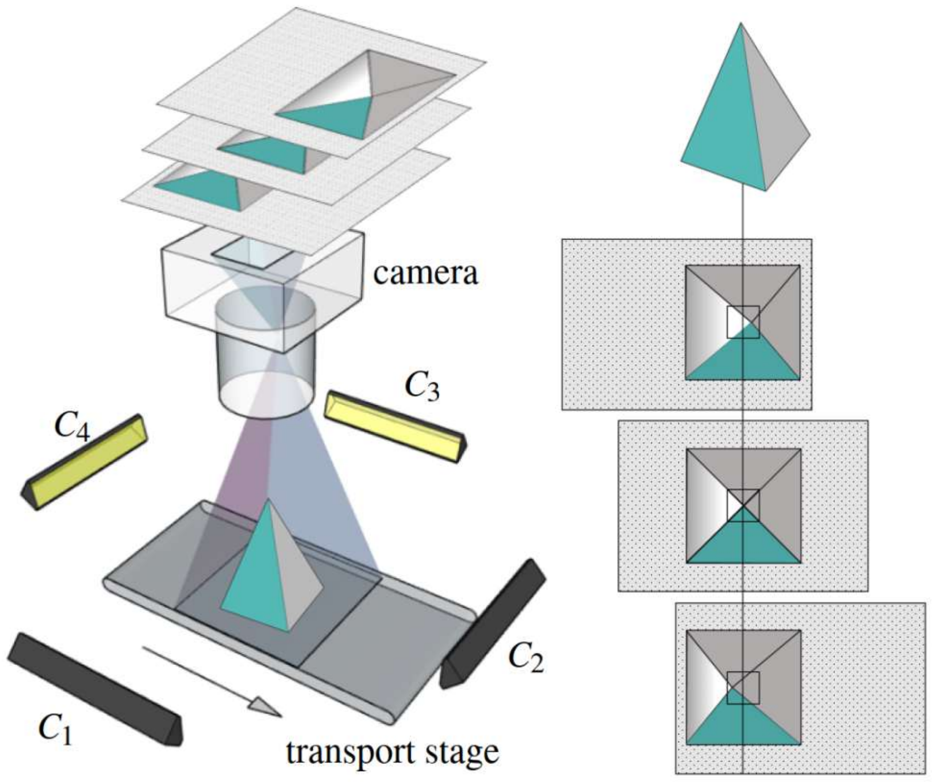


Figure 2.3: The ICI (Inline Computational Imaging) principle: a camera continuously acquires images from an object which is moved on a conveyor belt. Additionally, the object is alternately illuminated from multiple light sources. The result is an image stack consisting of acquisitions from the object from different viewing angles and under different illuminations. These images are processed with the ICI 3D pipeline and a 3D point cloud is created from the final depth model at the end. Figure is taken from [GBT22].

speed and high-precision 3D-reconstruction technology performing surface reconstruction on the GPU [GBT22, TGBB21]. In future, we want to use Potree for visualizing large ICI 3D point cloud data sets and we want to integrate the LOD generation directly into the ICI 3D-reconstruction pipeline. Thus, this master thesis is aligned to ICI and targets to provide high-performant LOD generation on the GPU in form of a software library. In addition to the performance advantages, this also allows to directly process ICI data residing on the GPU without additional $device - to - host$ copies. This section is dedicated to ICI and explains the basic concepts behind this technology.

Where former 3D sensing technologies suffer from inaccuracies on fine surface details or lack global consistencies, we at AIT propose ICI, which performs solid in macro (millimeter) as well as in microscopic scales (one-digit micrometer resolutions). The key point to achieve this depth range and accuracy is, that ICI exploits the sensitivity of light fields [LH96] on (large) surface structures and the sensing ability of photometric stereo [Woo92] for fine surface deviations without the need for visible structures.

The ICI technology consists of the ICI hardware setup, including an acquisition software and the ICI 3D processing pipeline. We distinguish between *MultilineICI* [ASV$^+$17] and *AreaICI* [GBT22]. Both technologies have several commonalities but also differ in some crucial aspects (e.g. hardware setup and image acquisition). Depending on the ICI variation a working ICI hardware setup consists of a multi-line (for MultilineICI) or area camera (for AreaICI), a defined number of light sources and a mechanism like a conveyor belt for a guided movement (see Figure 2.3). The number of mounted light sources has developed over time and depends on the acquisition mode. Typically two, four or six lights are used within the setup. During the acquisition, an object is constantly moved under the camera in one direction. This object is illuminated by multiple light sources, and a range of images is taken from the object from different viewing angles. These images form a light field structure, spanning over 3 dimensions (two spatial and one directional) and can be represented in an image stack in which each image shows the object of interst from a different viewing angle and under a different illumination. The images are passed through the ICI processing pipeline which performs the 3D-reconstruction. The result of the reconstruction is a final depth model that is converted to a 3D point cloud. The entire 3D reconstruction pipeline is implemented in CUDA and performs processing on the GPU in four steps. Here we focus on the ICIArea processing pipeline which is described in particular in [GBT22]:

1. Multi-view stereo matching
2. Disparity map fusing
3. 3D model generation (see [BSA18])
4. Applying *Total Generalized Variation* (TGV) for regularizing the 3D model (see [ASP18])

One of the goals of this master thesis is to provide a modular LOD conversion library which can be integrated into any third-party application. In our case, we use the resulting library to convert the point clouds that ICI generates with CUDA into a LOD structure directly on the GPU, without the need to move it to system memory or hard disk first.

2.3 Fundamentals and Related Work of LOD Conversion

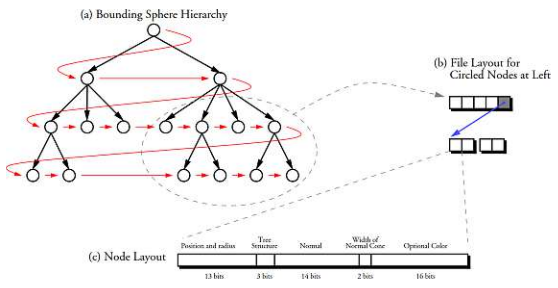
Many web-based point cloud viewers use an LOD structure to efficiently stream and render large point cloud data. The lowest LOD contains only a very rough representation of the overall point cloud, whereas higher LODs contain more detailed versions at a higher resolution. These multi-resolution data structures enable rendering engines such as Potree to quickly stream-in a coarse representation of the actual point cloud and to increase the amount of details over time by loading higher LODs. The goal of this thesis is to improve the state-of-the art LOD generation from Potree and PotreeConverter [Sch16] in terms of runtime and visual quality of the generated multi-resolution point clouds. For that purpose, previously proposed data structures and algorithms are used. These fundamentals are discussed below.

2.3.1 LOD Structures for Point Clouds

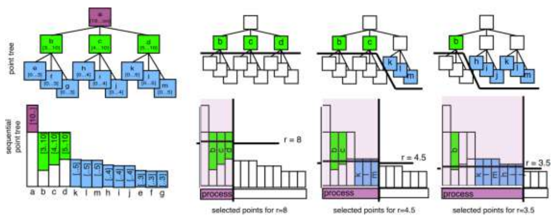
Converting point clouds into LOD structures and rendering them continuously is a wide-studied topic in the area of computer graphics. While several approaches have been proposed in the last years, this section focuses on approaches which form the basis for Potree, its multi-resolution point cloud conversion and thus also for PotreeConverterGpu. The first system which rendered point clouds based on an LOD structure was *QSplat* [RL01] (see Figure 2.4a) which was proposed in 2001. Qsplat converts arbitrary point data (point clouds, mesh vertices, etc.) to a binary tree and rearrange the actual points into the leaf nodes of the tree. Each further inner node contains a representative bounding sphere as well as averaged point properties (such as colors or normals) of the underlying subtree. During rendering, Qsplats traverses the binary tree and draws a single representative splat for each bounding sphere. Splatting is further discussed in Section 2.3.3. Although, Qsplat performed tree traversion and rendering on CPU side and was therefore limited in performance, it lays the foundation for several future point cloud LOD conversion and rendering systems.

One of these successors was an LOD rendering system called *Sequential point trees* (see Figure 2.4b) which aimed to completely shift workload from CPU to GPU. The concept of Sequential point trees was proposed by Dachsbacher et al. [DVS03] and targets to avoid traversing the point tree hierarchy on CPU side during rendering the point cloud on GPU. For this purpose the tree is flattened into a non-hierarchical list of nodes. Additionally, every node gets an r_min and r_max value assigned which indicate a distance range for which the actual node should be rendered. The entire node list is afterwards sorted for r_max. During rendering, based on the camera distance r it is decided for each point if it is in the boundary of $[r\_min, r\_max]$. This decision is performed directly in the vertex shader. Points that do not meet this requirement are culled, otherwise they are displayed as splats. To further optimize this process, not the entire node list is passed to the GPU. Instead, the CPU performs pre-culling for r_max and just passes the resulting segment to the GPU.

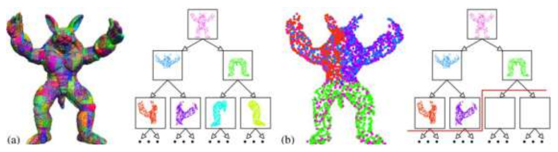
Another popular concept is that of *Layered Point Clouds* (see Figure 2.4c) which was proposed by Gross et al. in [GM04] in 2004. They propose to generate a point cloud LOD structure by building a binary tree in a top-down fashion. The final tree has exactly the same amount of points as the input point cloud model and each node stores M points where $M < N$ (N is the point amount in the cloud). The root node contains a coarse representation of the entire point cloud, whereas each tree branch divides the point cloud spatially. To get the model at a spesific resolution, all nodes above a predefined cut are merged. The idea behind Layered point clouds is to provide a system to load and render large point clouds either from a local hard disk or to stream them over the network. For



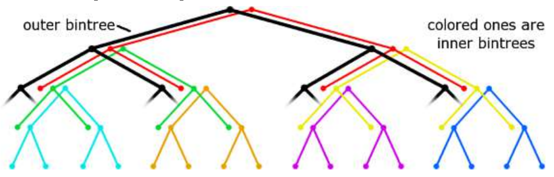
(a) QSplat: Image taken from [RL01]



(b) Sequential point tree: Image taken from [DVS03]



(c) Layered point clouds: Image taken from [GM04]



(d) Nested octree: Image taken from [WS06]

Figure 2.4: Related work: trend-setting point cloud LOD structures in the order of their publication: starting top left with QSplat and ending down right with the Nested Octree.

this purpose the system builds an index tree (containing the traversal data) and a point cloud repository which contains the actual octree nodes. During rendering, a specific model resolution is loaded from the point cloud repository by the use of the metadata stored in the index tree.

Two variations of the Layered Point Clouds were proposed by Wand et al. in [WBB+08] and also by Scheiblauer in [Sch14] which propose concepts to convert point clouds into editable multi-resolution point clouds. Although, both works are based on the idea of converting a given point cloud into an LOD structure using an octree, they differ in crucial aspects such as how points are inserted into the tree or how the tree is updated after point insertion or deletion. This thesis adopts a color filtering concept for point clouds which was first proposed by Rusinkiewicz and Levoy [RL01] and later by Wand in [WBB+08] (see Section 2.3.3). In addition, PotreeConverterGpu is based on Potree's octree structure from Schütz [Sch16], which is a slightly adapted version of the *Modifiable Nested Octree (MNO)* from Scheiblauer [Sch14]. Originally, the MNO was proposed to compensate for the weaknesses of the Nested Octree [WS06] (see Figure 2.4d) in terms of subsequent point editing (adding, deleting). The basic principle behind an MNO is, that it stores different resolutions of the same point cloud, whereas the root node contains the lowest resolution. Points are inserted into the MNO sequentially. For each point, the target index in a regular grid is evaluated which is placed over the point cloud. If the evaluated cell is empty, then the point is stored there, if not, a child node index is calculated and the same validation is performed. This process is continued until an empty cell is found.

	PotreeConverter 1.x	**PotreeConverter 2.x**
Building strategy	top-down	bottom-up
Subsampling	Poisson-disk (per node)	Poisson-disk (node and ancestors) (improved blue-noise)
Binary output	One file per node (up to millions)	One file for whole octree
Performance	Up to 1M Points / sec	Up to 10M Points / sec

Table 2.1: Comparison between PotreeConverter 1.x and PotreeConverter 2.x

2.3.2 PotreeConverter

In order to generate point cloud LODs for rendering with Potree, Schütz proposed *PotreeConverter* in [SOW20], which loads point clouds (e.g. from LAZ files) and converts them to a Potree-compatible LOD structure. In this master thesis the state-of-the-art LOD generation process from PotreeConverter is improved in terms of runtime and in terms of the visual quality of the resulting LOD data (reduction of aliasing artifacts). Furthermore, the resulting LOD generation software, called PotreeConverterGpu shall be easily integrateable into third-party applications. This section explains the theoretic fundamentals of PotreeConverterGpu and its related work.

2.3.2.1 PotreeConverter 1.X vs PotreeConverter 2.x

The current version of PotreeConverter is 2.1 and was proposed in [SOW20]. Table 2.1 lists several key differences between version 2.x and its predecessor. Although both versions are based on the MNO, the way it is built up is very different. PotreeConverter 1.x propagates points from the root node (lowest level of detail) to higher levels of detail - so the octree is built in a top-down fashion. The goal thereby is to limit the number of points in a node to a certain number, while minimizing the overall node amount. PotreeConverter 2.x on the other builds the octree in a bottom-up way. For this it first distributes points into leaf nodes and then propagates points upwards into lower levels of detail.

In fact, the main goal of PotreeConverter 2.x was to improve point subsampling and data export of the multi-resolution point cloud. Whereas PotreeConverter 2.x still relies on Poisson-disk sampling, it enforces point sets with a blue noise characteristics [YGW$^+$15] which implies (i) a minimum distance between subsampled points, (ii) avoidance of gaps between subsampled points and (iii) avoidance of regular sampling patterns in the subsamplings. In previous PotreeConverter 1.x versions these properties have only been applied within the boundaries of individual nodes. PotreeConverter 2.x improves the overall quality of the point subsamples by enforcing blue noise also between internal and child nodes. Moreover, the data export has also been improved by the fact, that PotreeConverter 2.x exports the multi-resolution point cloud into a single binary file

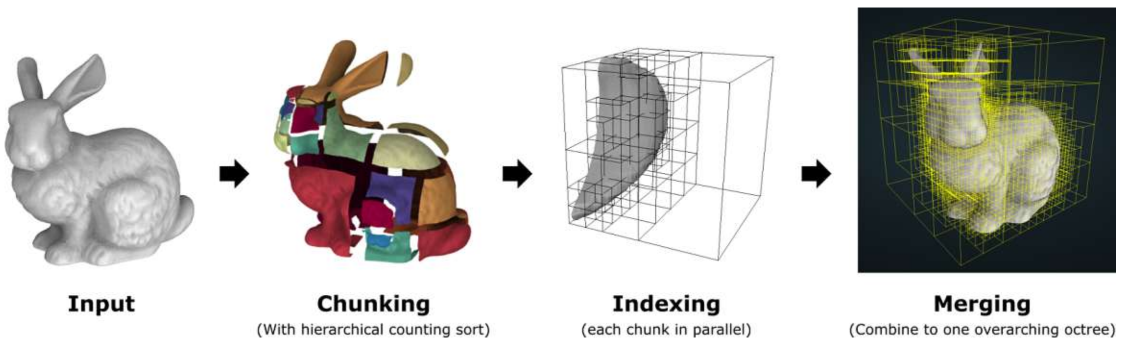

Input
Chunking
(With hierarchical counting sort)
Indexing
(each chunk in parallel)
Merging
(Combine to one overarching octree)

Figure 2.5: PotreeConverter 2.x overview. Figure taken from [SOW20].

rather than storing each octree node in a separate file. The disadvantage of creating a separate file for each node is, that the number of exported files can quickly lead to multiple millions, which reduces the data export speed and introduces problems during file system operations (copy, delete, transfer, etc.). This has been overcome by PotreeConverter 2.x by storing all binary octree nodes in one file and by storing descriptive node metadata in another separate file. More on the Potree data export format in Section 4.5. Another important aspect is, that PotreeConverter 2.x features a significant better runtime performance. Where PotreeConverter 1.x was able to process up to 1 million points per second, PotreeConverter 2.x is able to process up to 10 million points per second.

2.3.2.2 PotreeConverter 2.x in Detail

In addition to the improved blue-noise sampling strategy, PotreeConverter 2.x aims to optimise the export of the multi-resolution point cloud and thus to minimize the amount of generated binary output files. Furthermore, PotreeConverter 2.x generates an octree in a bottom-up way using an out-of-core approach. This process can be described in 3 steps:

1. **Chunking**
   Initially, the point cloud is partitioned into cubic chunks by applying a hierarchical variation of *counting sort* (further described Section 2.5). This means to place a three-dimensional, regular grid (e.g., 512^3 cells) on the point cloud and to project the points to grid cells. Afterwards, sparsely populated cells are merged hierarchically in a process that is similar to the creation of *image pyramids* [AAB+83]. For this purpose, cells are compared in groups of $2 * 2 * 2$. If the accumulated point count in those groups is lower than a specified *threshold*, their cells are merged into one bigger cell. The last step within the *Chunking* pass is to actually distribute the points from the source point cloud to the chunk files. For this, a *chunk lookup table* is created which maps the original cells of the full resolution grid to the merged

cells. Afterwards, the points are distributed and exported to the chunk files. Those files contain cubic point chunks, which align to a node in the octree.

2. **Indexing**
   After all points have been distributed, the resulting chunk files are loaded and local octrees are built in parallel (one thread per chunk/octree) – thus, multiple resoultions for a single chunk are created. For this, the chunk is processed in nearly the same way as during the *Chunking* pass, but this time in-core and a smaller counting grid of 32^3 cells is used, which results in an octree with 5 levels. If leaf nodes in the resulting local octree contain too many points (more than 10k), they are recursively processed into 5 more octree levels.

   When the local octree structure is created, *subsampling* can be performed. This means to extract coarser representations from finer octree levels and propagate them upwards in a bottom-up fashion. Starting at the bottom, all nodes are visited in a post-order depth-first traversal and if a node is an internal one, subsampling is executed for all direct child nodes. Extracted points are deleted from child nodes, which are then considered as finished and exported to the single output file (octree.bin).

3. **Merging**
   After all local octrees have been generated from chunks, the global octree can be created and exported. At the beginning, the global octree's leaf nodes are assembled of the root nodes from the local octrees. Creating the global octree requires to subsample and export its nodes in the same way as for the local octrees. After this step the octree is completely generated and exported.

2.3.3 Color Filtering for Point Clouds

Aliasing artifacts are a common problem in computer graphics, especially in point cloud renderers. A well known solution which targets this problem is *Surface Splatting*. This approach minimizes aliasing artifacts by blending overlapping points [ZPBG01]. The idea behind this algorithm is to weight the projected pixels of a point using a Gaussian filter. The filtered output of all contributing points is accumulated and afterwards normalised by the sum of all weights. The problem is that surface splatting relies on oriented splats, which requires the presence of normals or radii in the point cloud data. As these properties are often not present in common point cloud data sets, Potree blends screen-aligned rather than oriented splats, as originally suggested by Scheiblauer and Pregesbauer [SP11]. Despite this, aliasing artifacts are still perceptible, especially when rendering points from coarser LODs because the small number of subsampled points in lower LODs is insufficient to build a meaningful average color value that is representative of all the hidden points that are stored in higher levels of detail.

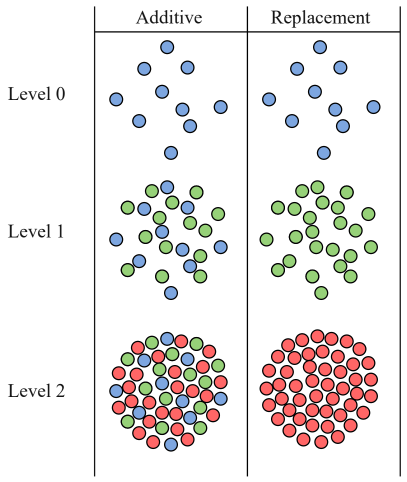
Similar to QSplat [RL01], which stores average color values in lower LODs of its bounding sphere hierarchy, and following a suggestion of Wand et al. [WBB$^+$08] who suggest to do the same for their LPC-like structure, we implement color filtering in our octree generator in order to improve the visial quality of lower levels of detail. The main modifications to the approach of PotreeConverter are:

1. Switching from an additive LOD scheme to a replacing LOD scheme.
2. Performing color filtering during subampling (similar to mip maps)

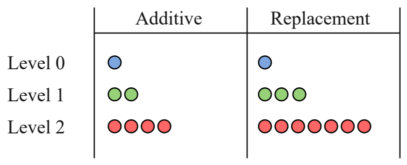
2.3.3.1 Replacement LOD Scheme

This section explains the difference between the additive LOD scheme used by LPC, MNO, and Potree, and the replacement scheme, which has been advised by [WBB$^+$08] to provide color filtering for point clouds. Consider an octree containing a multi-resolution point cloud. Additive refers to schemes where points in higher levels of detail are added to points from lower levels of detail during rendering (see Figure 2.6a). The advantage of this structure is that it is sufficient to reorder the original points into a hierarchical structure, and does not require additional or duplicate data. The idea of replacement schemes, on the other hand, is that higher levels of detail replace lower levels of detail during rendering. These schemes require additional memory (see Figure 2.6b) for each of the lower levels of detail, but they have the advantage that points in lower levels of detail can be representative averages rather than selected samples. The difference in quality between additive and replacement LOD schemes for point clouds is akin to the difference between nearest-neighbor sampling and a linear minification filter for textures.

PotreeConverter 2.x [SOW20] uses an additive approach - points are subsampled from finer to coarser octree levels without any averaging or quantization, i.e. one representative point is chosen from the underlying child nodes per subsampling grid cell. What [WBB$^+$08] suggests is not to just take over subsampled points as they are, rather they suggest to accumulate point attributes (e.g. color values) from all points within a grid cell and to assign them to the subsampled point. Furthermore, the amount of points per cell is also stored. This enables to normalize subsampled point values during quantization. In fact this approach suggests to apply an unweighted average filter on subsampled points. To my best knowledge this method has not yet been applied in modern, GPU-friendly, point cloud rendering structures. One reason might be the relatively high effort to calculate the averages per subsample for hundreds of millions of points..



(a) Additive vs. replacement LOD scheme during rendering



(b) Additive vs. replacement LOD scheme in data storage

Figure 2.6: a): Using the additive scheme, points from lower LODs are added to the points from higher LODs during rendering. When applying the replacement scheme, points from higher LODs replace points from lower LODs.
b): To be able to replace points from lower LODs during rendering, points are stored redundantly using the replacement octree scheme.

2.4 GPGPU Programming: The CUDA Framework

General Purpose Computation on the Graphic Processor Unit (GPGPU) programming extends the usage of GPU devices from graphics processing tasks to general high-performance applications with a focus on parallelization. Especially the rise of different GPU programming paradigms such as *Compute Unified Device Architecture* (CUDA) [NVF] and *Open Computing Language* (OpenCL) [Groa] has enabled a range of completely new applications in different areas, including image processing and machine learning. The next Section provides a general comparison of CPU and GPU devices and how latter ones are utilized by CUDA for high-performance GPGPU programming.

2.4.1 An Introduction to CUDA

CUDA is a computing platform and programming model which was introduced by Nvidia in 2006. Its purpose is to exploit the massively parallel nature of GPUs to solve complex, parallel and computationally intensive tasks. This section gives a rough overview about the CUDA framework and how it used, where the focus lies especially on those aspects that are relevant to this master thesis. The explanations in this section are mainly based on the *CUDA Programming Guide* [NVIa].

When comparing CPUs with GPUs, one has to keep in mind that both devices were designed for different areas of applications. Although both device types share common

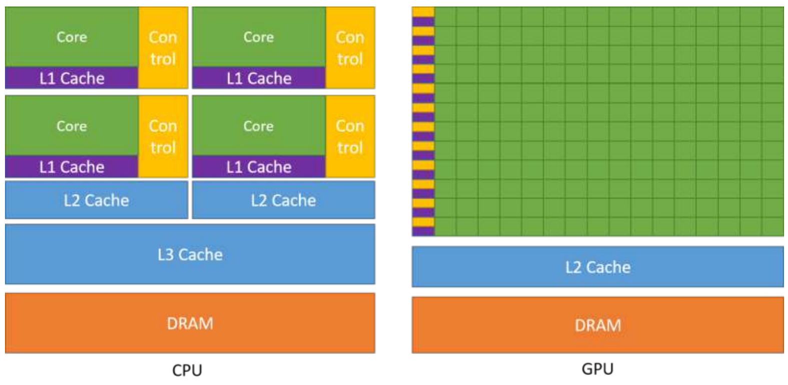


Figure 2.7: Comparing CPUs and GPUs: CPUs and GPUs use a different number of transistors for data processing. GPUs utilize more transistors for cores than CPUs. Figure is taken from [NVIa].

features and are based on the same technologies, the available amount of transistors are dedicated for different purposes. As the main focus of CPUs lies on high-performance and task-parallel workload, a high amount of transistors is used for complex control logic and for a large set of data caches to minimize memory latencies (see Figure 2.7). GPUs on the other hand, feature a different hardware architecture where the goal is to maximize the computational capability in terms of data-parallelism while minimizing the amount of hardware resources for caching and flow control. Thus, the majority of transistors in GPUs are dedicated for arithmetic logics components [HCZ16]. For example, modern high-end consumer CPUs like the *Intel Core i9 Extreme Edition Processor* [Int] feature an amount of 18 cores, where actual GPUs like the *RTX 4090* [NVIb] have a core amount around 16k. Further, GPUs feature a *Single Instruction Multiple Threads* (SIMT) architecture which allows to execute workloads embarrassingly parallel. In particular, this allows to execute the same function (even with multiple branches) onto millions of data sets simultaneously. This has the advantage that GPUs highly outperform CPUs in terms of parallel data processing and computation performance [HCZ16, NVIa]. CUDA utilizes the GPU's parallel compute engine and leverages its functionalities using application programming interfaces for various languages, such as: C, C++, Fortran, etc. As describing CUDA's architecture would exceed the boundaries of this master thesis the next section directly explains the practical usage of CUDA in a C/C++ environment. For more information about the CUDA architecture and its programming model refer to the *CUDA Programming Guide* [NVIa].

2.4.2 CUDA Programming by Example

Basically, the CUDA workflow is to define algorithms in CUDA kernels first and to let CUDA execute those kernels on one or more GPU devices with a given launch configuration, afterwards. These kernels can be implemented using *CUDA C++*, which is a C++ extension that has to be compiled using the *NVIDIA CUDA Compiler* compiler

```
 1 // A cuda kernel, which is executed on the GPU
 2 __global__ void ScalarMultiply(float *A, float Scalar)
 3 {
 4     int idx = threadIdx.x;
 5     A[idx] = A[idx] * Scalar;
 6 }
 7
 8 // Host-function that invokes the CUDA kernel N-times
 9 int main()
10 {
11     // ... Create and copy an array 'A' to GPU (device) memory
12
13     ScalarMultiply<<<1, N>>>(A, Scalar);
14
15     // ... Copy the array back to CPU (host) memory
16 }
```

Listing 2.1: A minimalistic CUDA kernel example code

(nvcc) in order to be executable. It is also possible to define kernels using the CUDA instruction set architecture (PTX) but NVIDIA highly recommends the use of CUDA C++ for this purpose.

2.4.2.1 A Simple CUDA Kernel

Listing 2.1 contains a minimalistic example that shows how a CUDA kernel is embedded into a regular C++ application. The example multiplies the elements of a floating point array by a single scalar value in parallel. The CUDA kernel definition starts at Line 2 with the `__global__` declaration, denoting that this kernel function can be called from host side. The opposite would be `__device__`, meaning that the function cannot be called from host code. After the declaration of the return type (which is always `void`) and the function name, the parameters passed from the host to the device are listed. Restrictions on what types of parameters can be passed to CUDA kernels can be found in *CUDA Programming Guide* [NVIa]. The given example dispatches the CUDA kernel with N threads, and each invocation then uses the thread index to identify the array element that it should process. This can be seen on Line 4 in the example. Based on the index, the executing thread accesses different data or executes different operation. Referring to Figure 2.8, CUDA threads are organized in blocks of threads, where these blocks are again organized in a grid of blocks. Threads inside a block can be identified in a multi-dimensional way, up to three dimensions and thread blocks can also be arranged in a one-dimensional, two-dimensional or three-dimensional grid. This allows to obtain natural thread indices suitable for several data structures such as vectors or matrices. The amount and layout of the executed threads are specified within the angle brackets

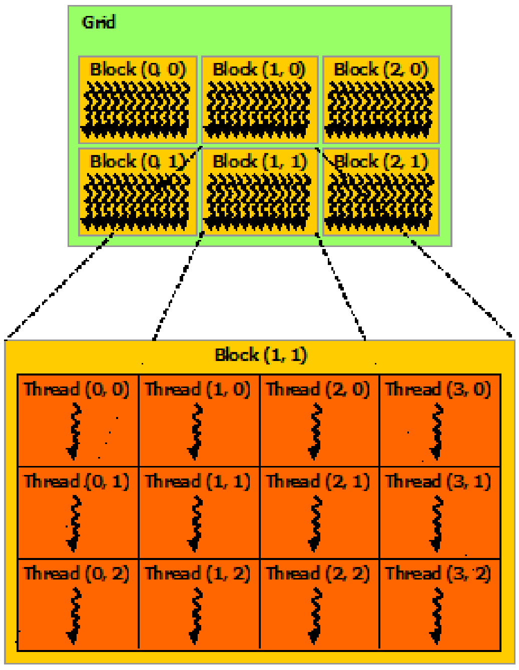


Figure 2.8: Thread blocks arranged in a grid. Figure is taken from [NVIa].

during kernel invocation (see Line 13). The maximum amount of threads per block is always bound to `1024`, so in order to process larger arrays, CUDA-kernels need to be launched with an appropriate number of blocks, containing multiple threads.

2.4.2.2 GPU Memory Management

This section focuses on the memory management in CUDA as an integral component within the CUDA framework. CUDA considers two separate memory spaces: *host* and *device* memory. This results from the fact that CUDA uses heterogenous programming and assumes CUDA kernels to be executed on a separate physical *device*. The *host* on the other hand executes the actual C++ program and refers to the *device* as a coprocessor. Memory management such as allocation, deallocations and copies are triggered by the host through the CUDA runtime. Each CUDA thread has access to a range of different device memory types which can be seen in Figure 2.9:

- Private local memory: restricted access - Only accessible to a single thread
- Shared memory: restricted access - shared by threads in a block
- Global memory: can be accessed by every thread in any block

Parameters which are passed to CUDA kernels (see Listing 2.1, Line 1) have to reside in global device memory to be accessible for CUDA threads. Thus, device memory has to be allocated and initialized beforehand. Listing 2.2 provides an example for typical

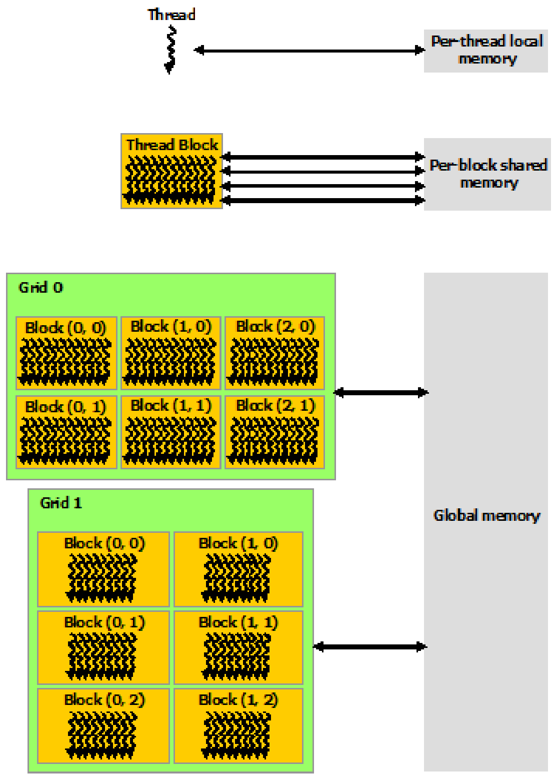


Figure 2.9: The CUDA memory hierarchy. The figure shows the dedicated context of each device memory type. Figure is taken from [NVIa].

CUDA memory management. To be able to process host data on the GPU, the host data (`data_host`) has to be copied to the device. For this purpose it is necessary to allocate GPU memory (`data_device`) with `BYTESIZE` bytes using `cudaMalloc` function (see Line 6). Afterwards, the host data can be copied to the GPU using `cudaMemcpy` function with the argument `cudaMemcpyHostToDevice`. After data processing on the GPU has finished, it might be necessary to copy data back to host for further processing. This can be seen in Line 11, where an additional `cudaMemcp` is performed, but this time using the `cudaMemcpyDeviceToHost` argument. Finally, reserved GPU memory has to be released, which is performed on Line 15 using the `free` function. As one can see in Listing 2.2, CUDA memory management introduces a significant amount of boilerplate code. Thus, allocating and deallocating of GPU memory has been abstracted inside a separate class in this master thesis.

2.4.3 Atomic Functions

Sometimes, parallel running CUDA threads may need to be synchronised across block boundaries. A suitable solution for this is provided by CUDA's atomic functions. According to the NVIDIAs Programming guide [NVIa], concurrently executed atomic operations

```
int main()
{
    // ... Generate host data ...

    int *data_device;
    cudaMalloc ((void**)&data_device, BYTESIZE));
    cudaMemcpy(data_device, data_host, BYTESIZE,
       cudaMemcpyHostToDevice);

    // ... CUDA kernel invocation ...

    cudaMemcpy(data_host, data_device, BYTESIZE,
       cudaMemcpyDeviceToHost);

    // ... Further host processing ...

    cudaFree(data_device);
}
```

Listing 2.2: A typical CUDA memory management example

consisting of (read / modify / write) are serialised and executed in a non-deterministic order one after the other. In PotreeConverterGpu, the most used atomic functions are `atomicAdd` and `atomicSub`. Those functions increment or decrement a variable in an atomic way and return the previous stored value. This mechanism is therefore highly suitable for generating globally coherent indices. Those indices can be either used for e.g. accessing array data or to execute special commands based on the index value. For example, a thread may need to execute a special function if it is the first to atomically access a specific array or grid element (i.e., `atomicAdd` returned 0).

2.5 Counting Sort

In PotreeConverter 2.x, [SOW20] suggests a cloud chunking algorithm that is based on a hierarchical variant of counting sort [CLRS09]. With counting sort n integer values can be sorted in a value range from *0* to k. Unlike comparison sort algorithms, the runtime complexity of this integer-sort algorithm is *O(n)*. The common way to implement the algorithm is to provide an input array *input* which is sorted using a counting array *counters*, where the sorted values are finally stored in an output array *output*. Thus, counting sort is only applicable, if it is possible to allocate a counting array with an entry amount that match the amount of possible keys. As shown in Algorithm 1, counting sort is performed in several steps. First, the counting array *counters [0, k]* is created for the input array *input[1, n]* and its elements are set to zero (Lines 1-3). Then each element in *input* is iterated over and is taken as an index for the counting array *counters*. The indexed entry in *counters* is incremented afterwards. After the last iteration, *counters* contains the number of occurrences for each element in *input* (Lines 7-9). The next step ranges from Line 7 to 9. Here, the goal is to determine for each element in *input*, how many elements are equal or less. For this, *counters* is iterated over and a running sum is calculated which is stored within each *counters*[*i*]. The last step is to distribute the sorted values from *input* to *output*, which is shown on Lines 10-13. The counting array *counters* can be directly used for indexing the output array *output*. However, the corresponding entry in *counters* has to be decremented in each iterations. The proposed hierarchical counting sort algorithm in PotreeConverter 2.x [SOW20] is an adapted version of the original counting sort algorithm, in the sense that sorting is performed cell-wise rather than point-wise.

Algorithm 2.1: The Counting Sort algorithm [CLRS09]

```
   /* Reset counting grid                                   */
 1 for $i \leftarrow 0$ to $k$ by 1 do
 2     counters[i] = 0;
 3 end
   /* 1. Phase: Counting                                    */
 4 for $i \leftarrow 0$ to $n$ by 1 do
 5     counters[input[i]] = counters[input[i]] + 1;
 6 end
   /* 2. Phase: Calculate bucket offsets                    */
 7 for $i \leftarrow 1$ to $k$ by 1 do
 8     counters[i] = counters[i] + counters[i-1];
 9 end
   /* 3. Phase: Value distribution                          */
10 for $j \leftarrow (n-1)$ to 0 by $-1$ do
11     output[counters[input[j]]-1] = input[j];
12     – counters[input[j]];
13 end
```

CHAPTER 3

Software Architecture

PotreeConverterGpu[1] is a re-implementation of PotreeConverter that generates point cloud LOD entirely on the GPU using CUDA. Whereas the concepts implemented in PotreeConverterGpu are mainly based on [Sch16] and [SOW20], its algorithms and internal processes have been re-designed to be executed efficiently on the GPU. To better understand the core concepts in Chapter 4, the current chapter provides a high-level introduction to PotreeConverterGpu, focusing on the software requirements, architecture and basic workflows/algorithms from a software engineering perspective.

3.1 Requirements

The flow chart in Figure 3.1 lists the processing steps within PotreeConverterGpu sequentially. These processing steps are used to deduce the *functional* and *non-functional* requirements for PotreeConverterGpu, which are listed in Table 3.1. Requirements describe the expected basic functionality of a software, i.e., what the software is supposed to do.

The main *functional* requirement for PotreeConverterGpu is that it should be possible to convert point clouds residing on either host or device memory into LOD data of the Potree 2.x format. Further, the conversion should be done entirely on the GPU. Finally, the software shall export the generated LOD data in Potree format 2.0 and should additionally generate statistics about the octree.

The *non-functional requirements* describe general attributes such as quality attributes, i.e., how a system should be designed. PotreeConverterGpu shall be fully integratable into (proprietary) third-party software – thus, a properly designed software interface is necessary. Nevertheless, it shall be possible to use PotreeConverterGpu as a stand-alone

[1]PotreeConverterGpu is freely available on Github under an open source license [Kla]

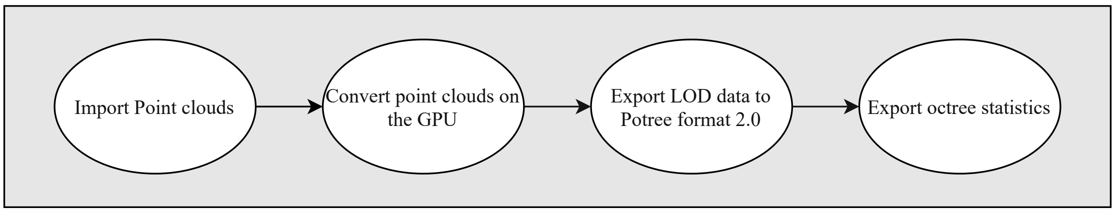

Import Point clouds
Convert point clouds on the GPU
Export LOD data to Potree format 2.0
Export octree statistics

Figure 3.1: PotreeConverterGpu flow chart

Functional requirements	Non-functional requirements
Shall import point clouds from host or device memory	Shall be integratable into third-party-software
Shall convert point clouds to LOD data entirely on the GPU	Shall be used as a stand-alone software
Shall export LOD data in Potree format 2.0	Shall be compatible with ICI 3D point clouds
Shall export octree statistics such as point distribution and GPU memory consumption	Shall perform the LOD conversion *in core*
	Shall perform the LOD conversion of 100 Mio points in an amount of time ≤ 1 second
	Shall provide color filtering for point clouds

Table 3.1: PotreeConverterGpu: functional and non-functional requirements

application. In order to be able to integrate PotreeConverterGpu into the ICI pipeline, the software shall be able to process ICI 3D data sets. To increase the LOD conversion performance, processing is entirely done *in core* on GPU using CUDA. In core means that all relevant data structures are kept in memory during processing, and that no data is buffered on hard disk. This requirement limits the size of the point clouds to be processed, as the point clouds and all data structures required for processing must be kept in GPU memory at the same time. On the other hand, this simplifies and accelerates point-cloud processing. In comparison to the former PotreeConverter 2.x, which is able to process ~10M points per second (see [SOW20]), we expect a minimum performance improvement of factor 10 from PotreeConverterGpu. Thus, a non-functional requirement is to be able to process 100M points per second. The last non-functional requirement targets the quality of the generated LOD data. As mentioned in Section 1.2, PotreeConverterGpu shall increase the visual quality of the produced LOD data by providing color filtering.

3.2 Software Architecture

This section introduces PotreeConverterGpu's basic software architecture based on the requirements described above. It is summarised in the UML component diagram in Figure 3.2.

According to Table 3.1, PotreeConverterGpu may be used as a stand-alone application or as library for third-party applications. The key consideration to achieve this is *modularity*. This architectural principle shall provide flexibility by separating concerns, while minimising *tight-coupling* between components. Following this principle, PotreeConverterGpu is separated into three independent components (Figure 3.2), each fulfilling its own purpose in a separated context (*Separation of Concerns* design principle). The communication with other components is performed over a well-defined interface. The following listing explains the purpose of each of the three software components.

- **OctreeLibrary**
  OctreeLibrary encapsulates the core functionality and data structures within PotreeConverterGpu. Thus, it contains all relevant CUDA kernels for generating the actual LOD structure for a given source point cloud. It is designed as a static library which gets linked to OctreeAPI and exposes a single C++ class API. Chapter 4, which explains the core algorithms in PotreeConverterGpu, mainly refers to this software component.

- **OctreeAPI**
  It is considered as an intermediate layer for exposing functionality from OctreeLibrary within a classical C API. To save state between consecutive API calls, a *session* object is is generated at the beginning and afterwards passed to each API call. This type of library interface in combination with the fact that OctreeAPI is compiled into a shared library, enables an easy integration into other software. Wrapping a C API additionally enables the integration into other programming languages such as Python or MATLAB.

- **PotreeConverterGpu Executable** This component represents the actual executable and consumes the OctreeAPI to convert point clouds into LOD structures. It provides flexible parametrisation of the LOD generation by accepting a variety of command line arguments.

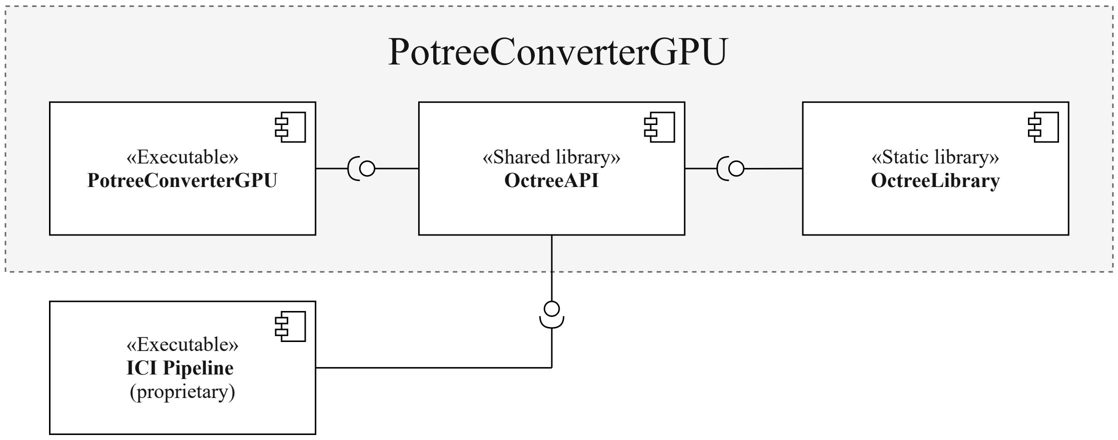

PotreeConverterGPU
«Executable» PotreeConverterGPU
«Shared library» OctreeAPI
«Static library» OctreeLibrary
«Executable» ICI Pipeline (proprietary)

Figure 3.2: PotreeConverterGpu UML component diagram: PotreeConverterGpu is split into independent software components. By default, PotreeConverterGPU is built as an executable that can be used from the command line. Alternatively, external applications may include OctreeAPI to integrate the functionality directly into their own executable.

3.3 Algorithmic Overview

This section briefly describes the core functionality and the processing pipeline of PotreeConverterGpu. As shown in Figure 3.3, PotreeConverterGpu performs several processing steps sequentially either on the CPU or the GPU. Tasks which are performed on the GPU are implemented inside one or more CUDA kernels. Although GPU programming requires a certain amount of CPU interaction in practice, e.g., starting of CUDA kernels, this is not shown in the activity diagram for simplicity. The core algorithms and data structures for each processing step are described in particular in Chapter 4. The following listing summarises each of those steps and refers to the individual sections.

1. **Point cloud import** (Section 4.1)
   A point cloud is imported for further processing either from host or device memory.

2. **Preparation phase** (Section 4.2)
   After data import, all relevant data structures are created. The idea is to perform as many memory allocations on the GPU side as possible during this phase. The reason for this is to be able to reuse once allocated GPU memory and to separate memory allocations from processing.

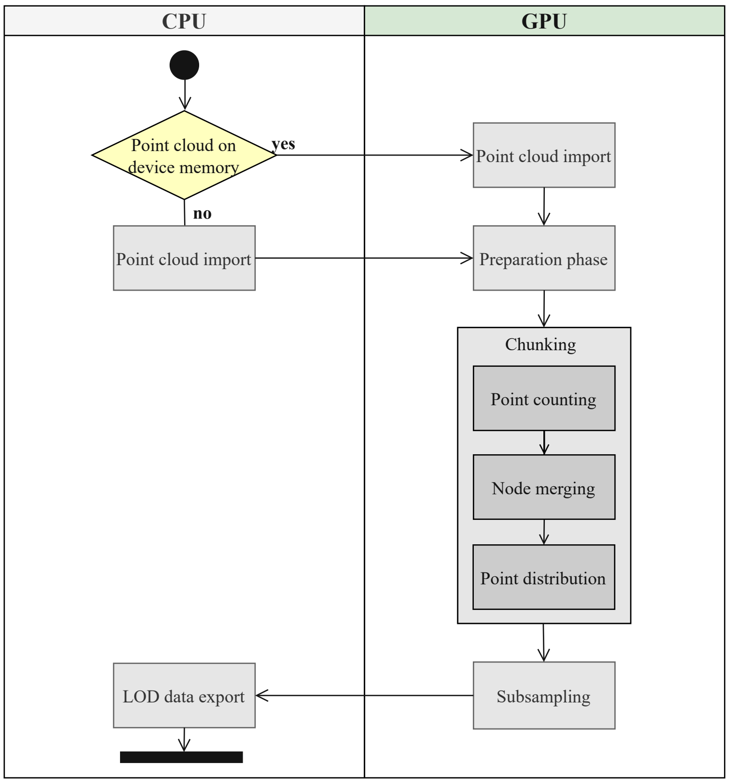


Figure 3.3: PotreeConverterGpu UML activity diagram: PotreeConverterGpu performs the point cloud to LOD conversion in multiple steps. These processing steps are either performed on the CPU or on the GPU side. Processing on GPU is performed in CUDA kernels which are executed from the host side.

3. **Chunking** (Section 4.3)
   Its main purpose is to split the source point cloud into cubic chunks and to generate the basic octree structure. During this process, the following tasks are performed:
   a) Point counting (Section 4.3.1)
   b) Node merging (Section 4.3.2)
   c) Point distribution (Section 4.3.3)
4. **Subsampling** (Section 4.4)
   Subsampling is the process of extracting points from finer to coarser octree levels. Here, multiple resolutions of the original point cloud are created.
5. **LOD data export** (Section 4.5)
   PotreeConverterGpu exports multi-resolution point clouds together with necessary metadata files in Potree 2.x format.

CHAPTER 4

Generating an LOD Structure on the GPU

This chapter describes *PotreeConverterGpu*, a GPGPU-based approach for converting point clouds into Potree compatible LOD data. PotreeConverterGpu significantly speeds up the octree generation process (13 times up to 85 times depending on subsampling quality) that is proposed in [SOW20] (PotreeConverter 2.x) through targeted performance optimisations and massive parallelization on the GPU. In addition, PotreeConverterGpu proposes anti-aliasing for point clouds, an approach which is based on [WBB$^+$08] and performs color filtering during subsample generation, similar to mip-maps. Although the core concepts are mainly based on [Sch16] and [SOW20], this requires a complete re-design and re-implementation of the point cloud conversion process and algorithms to be executed efficiently on the GPU.

The focus in this chapter lies on describing PotreeConverterGpu in detail, while putting particular attention to the mentioned adjustments of algorithms and data structures that are required for its highly optimised GPU implementation. The explanations follow the processing pipeline mentioned in Figure 3.3, whereas each processing step is described in a separate section and with a combination of figures and pseudo code. Here, a particular focus is put on the runtime improvement within PotreeConverterGpu in comparison to PotreeConverter 2.x, and how it could be achieved. As PotreeConverterGpu provides baked-in anti-aliasing for point clouds, which significantly improves the visual quality of the generated LOD data, this topic will also be discussed.

According to Figure 3.3, we first describe how supported point cloud formats are loaded/imported (Section 4.1), followed by the preparation phase that describes essential pre-processing steps such as GPU memory allocations (Section 4.2). The actual generation of the octree structure is covered in the *Chunking* Section 4.3 and explains how the initial point cloud is partitioned into leaf nodes. Generating the final LOD structure requires to subsample previously generated octree nodes. The corresponding explanations can be found in the *Subsampling* Section 4.4, which also explains the proposed color filtering strategy for point clouds in detail. The last processing step is the *LOD Data Export*, which is covered in Section 4.5 and explains how the actual LOD data is exported to a Potree-compatible data format.

4.1 Point Cloud Import

Point cloud import is the first step in the processing pipeline (Figure 3.3). Contrary to PotreeConverter 2.x, which only processes data from disk, PotreeConverterGpu also supports processing data that already resides in GPU memory. This allows PotreeConverterGpu to be integrated into 3D reconstruction software that generates point cloud data directly on the GPU, and thereby avoids additional round-trips to disk storage and back into memory.

In its simplest form, point clouds consists of an array of 3D points, each of them describing the position of one point in the 3D space (x,y,z). Depending on the acquisition or generation technique, each point may have additional properties, such as color or scalar values. As a minimum requirement in our implementation, each point has to consist of an (x,y,z) coordinate value and an (r,g,b) color value. Currently, PotreeConverterGpu supports coordinate values of type `float` or `double`, and (color) attribute values of type `uint8` per channel. Additional attributes in the input data set are currently ignored and not added to the resulting LOD structure.

To be able to process point clouds with an arbitrary amount of input attributes, the knowledge of the data *stride* is essential to access/ignore the correct attributes (see Figure 4.1). This value is equal to the sum of bytes per point, here referred to as $attributeSize$:

$$stride = \sum_{i=1}^{k} attributeSize_i \tag{4.1}$$

Using the *stride*, we can retrieve the coordinate and color values of each point n in the cloud by following byte offsets (where $coordByteSize$ is the byte size of one coordinate component):

$$\begin{aligned} \text{byteOffset}_{coord} &= stride * n \\ \text{byteOffset}_{color} &= stride * n + 3 * coordByteSize \end{aligned} \tag{4.2}$$

Supporting arbitrary attributes and strides is essential because commonly used formats differ in their layout. For example, the smallest stride for points with colors in Potree, LAS [fPS], and ICI are 16, 26 and 15, respectively. The difference between Potree and ICI stems from the fact that Potree pads the rgb attribute to 4 bytes, while ICI does not.

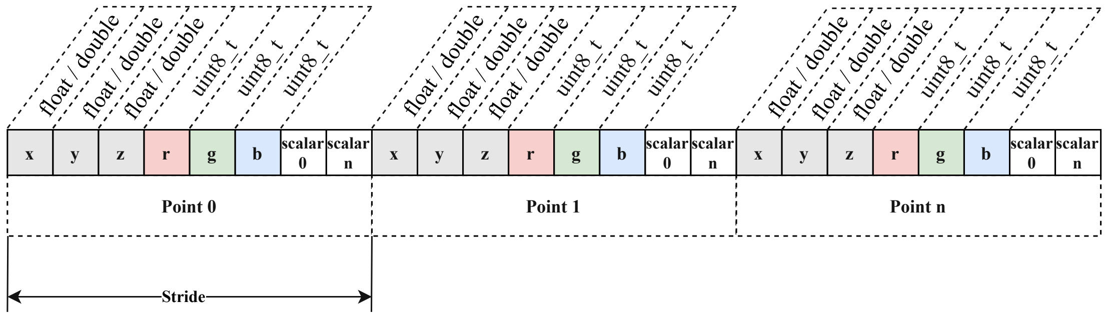


Figure 4.1: PotreeConverterGpu supported point cloud data layout: a single block of memory containing points in a linear, consecutive order. Each point has to contain coordinate (x, y, z) and rgb color values.

Points in the LAS format are larger because they contain more than just coordinate and color attributes, and because colors are stored with two bytes per channel.

PotreeConverterGpu is able to process point clouds residing on host as well as on device memory. In practice this means that OctreeAPI has to be provided with a data pointer, pointing to a continuous block of memory (no matter whether on CPU or GPU) which contains the point cloud data. If the point cloud resides on host memory, it is copied to device memory for further processing during the *Preparation phase* 4.2. Otherwise, the point cloud can be imported directly from GPU memory without any time-costly copy operation, which speeds up the overall processing time. To be able to provide OctreeAPI with a pointer to a host point cloud, the PotreeConverterGpu executable reads the cloud from a binary file with the proposed data layout. The buffer on the GPU that contains the imported point cloud is referred to as *cloudBuffer* (see Table 4.1). To save GPU memory, CUDA kernels within PotreeConverterGpu avoid additional copies of the original point cloud data. In fact, the data is only copied once, i.e., during exporting (discussed in Section 4.5).

4.2 Preparation Phase

According to the activity diagram in Figure 3.3, the *Preparation phase* is the second step in the processing pipeline. Its purpose is to reserve and initialize the required amount of GPU memory using CUDA's blocking `cudaMalloc` function. The reason is two-fold: First, initially reserved GPU memory can be reused during processing. The second reason to reserve as much GPU memory as needed during *Preparation phase* is that this enables a separate time measurement of memory allocation time and pure processing time. This section comprises the two most important data structures (listed in Table 4.1) which are allocated on the GPU during the Preparation phase.

Variable	Datatype	Size
cloudBuffer	uint8_t[ ]	pointAmount * inputDataStride
outputBuffer	uint8_t[ ]	pointAmount * outputDataStride * outputFactor

Table 4.1: Relevant data structures pre-allocated during preparation phase

4.2.1 Allocating the Cloud Buffer

If the input point cloud resides on host (CPU) memory it is necessary to copy it to the device (GPU) memory beforehand. If it already resides in GPU memory, we will simply take a pointer to the GPU memory location and avoid additional allocations and copy operations. The destination buffer for the input point cloud on the GPU is referred to as *cloudBuffer* and is listed in Table 4.1. The size of *cloudBuffer* depends on the amount of points (*pointAmount*) in the input cloud and its *inputDataStride* (see Section 4.1). Thus, the required amount of GPU memory for *cloudBuffer* can be calculated as follows:

$$cloudBuffer = pointAmount * inputDataStride \tag{4.3}$$

4.2.2 Allocating the Output Buffer

octree.bin is a file which is created within the *Data Export phase* and is described in Section 4.5. After export, it contains the multi-resolution point cloud which is going to be rendered by Potree. Unlike [SOW20], which exports point data directly to octree.bin already during processing, PotreeConverterGpu holds exported point data in an output buffer. This buffer resides on the GPU and is copied from there to the host (CPU) after all points of the finished LOD data structure are stored in it. Only then the contents of the buffer (here referred to as *outputBuffer*) is written to octree.bin on disk.

The *outputBuffer* is allocated on the GPU during the preparation phase. As the exact amount of exported points is unknown at this stage (the replacing LOD approach generates an unkown amount of additional points), the size of the *outputBuffer* is estimated using the *pointAmount* of the source cloud, the *outputDataStride* and an empirically chosen factor – the *outputFactor*. This *outputFactor* heavily depends on certain configuration and processing parameters and is set to 2.2 per default. The following equation shows how the amount of memory for the output buffer is calculated:

$$outputMem = pointAmount * outputDataStride * outputFactor \tag{4.4}$$

The stride for the output data together with its structure is explained in Section 4.5 and can be seen in Figure 4.16. In general, the layout of the *outputBuffer* is very similar to the one of the *inputBuffer*. Specifically, all points are stored one after the other, each of them consisting of three coordinate and three color values (rgb). Besides the data types of the exported point attributes, the output buffer also differs in another aspect: additional point attributes such as scalar values are not present anymore. The reason for not exporting scalar values is that processing in PotreeConverterGpu is done in-core and thus, we have to save as much GPU memory as possible while focusing only on rgb color values. In consequence, the stride of the output data can be assumed fixed to be of a size of 18 Bytes. This stride is the sum of the amount of Bytes per point coordinate (*coordinateBytes*), which is 12 Bytes, and the amount of Bytes per point color (*colorBytes*), which is 6 Bytes (2 Bytes per color channel):

$$stride = 3 * (coordinateBytes + colorBytes) \tag{4.5}$$

For a source point cloud with, e.g., 100 million points and using an $outputFactor$ of 2.2, the approximated amount of memory for the output buffer would be:

$$outputMem = 100{,}000{,}000 * 18byte * 2.2 = 3.96GB \tag{4.6}$$

4.3 Point Chunking

During the *Point Chunking* phase, the hierarchichal structure of the octree is created and all points from the source point cloud are distributed to the octree's leaf nodes. This forms the starting point for the subsampling phase.

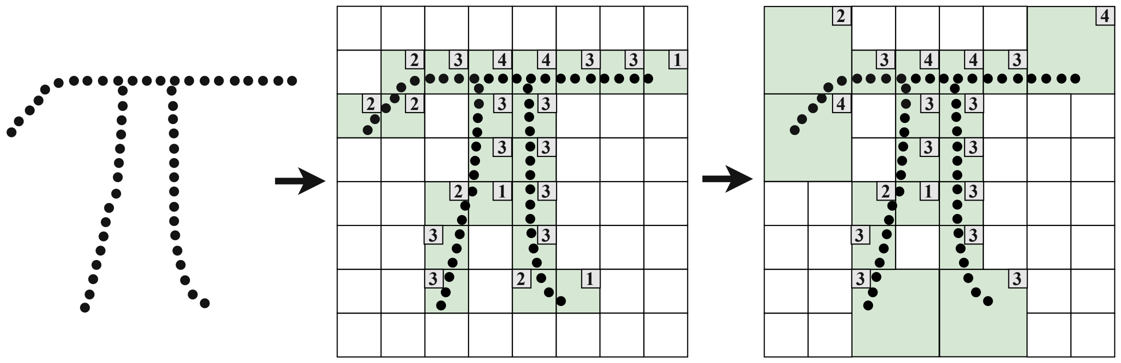


Figure 4.2: Point chunking overview: at first, the input point cloud is spatially divided into cells within a high-resolution grid. These cells correspond to potential leaf nodes. The partition is done by counting the amount of points per cell. To avoid empty octree nodes and nodes with only a few points later on, empty cells are eliminated, whereas any 8 adjacent cells with too few points (lower than a threshold) are merged. This step is done in a hierarchical way until all cells contain enough points. Finally, points are distributed from the point cloud to the resulting leaf nodes of an octree.

The applied chunking algorithm is based on [SOW20] and uses a hierarchical variant of counting sort [CLRS09]. As shown in Algorithm 1 in Section 2.5, the original counting sort algorithm is performed in 3 steps. Our point chunking approach is basically modelled after these 3 steps, but with crucial adjustments. The individual steps are:

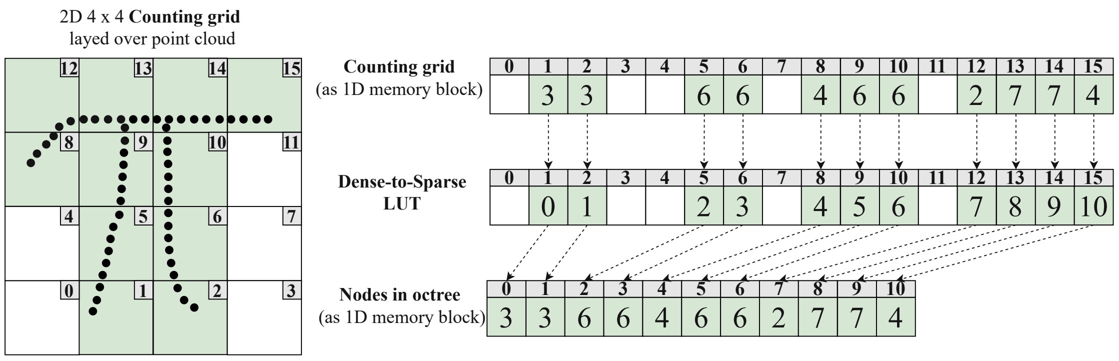


Figure 4.3: Point counting: a schematic representation of a 2-dimensional 4x4 counting grid placed over a point cloud. Occupied cells with a point amount greater than zero are marked green. A dense-to-sparse lookup table (LUT) maps dense counting cells to sparse octree nodes.

1. **Point Counting**
   The Point Counting algorithm can be found in Section 4.3.1 and works basically the same as the *Counting* phase in Algorithm 1. This pass places a 3-dimensional high resolution grid over the point cloud and counts how many points fall into each cell. The resulting counters are then used to reserve the exact amount of required memory for each leaf node.

2. **Node Merging**
   During this phase, the actual sparse octree structure is created. To avoid nodes with only few points in them, neighbouring nodes are merged together into one bigger node if their point sum is lower than a threshold (see Figure 4.2). Node merging is done recursively until no more nodes can be merged.

3. **Point Distribution**
   The third and last phase during chunking is the point distribution phase. It corresponds to the *Value distribution* phase in Algorithm 1 and is described in Section 4.3.3. The purpose of this last step is to distribute the points to the octree's leaf nodes.

4.3.1 Point Counting

The goal of Point Counting is to partition the source point cloud into cubic chunks. For this purpose, a 3-dimensional high-resolution grid with equal side length is placed over the entire point cloud. Afterwards, it is evaluated how many points fall into each of the grid cells. Those point counts are then propagated upwards and are used for creating the actual octree data structure in the next phase – *Node Merging* in Section 4.3.2.

Table 4.2 contains an overview of relevant data structures for this section:

Variable	Datatype	Size
countingGrid	`uint32_t[]`	$1^3 + 2^3 + 4^3 + 8^3 + ... + gridSize^3$
denseToSparseLUT	`int[]`	$1^3 + 2^3 + 4^3 + 8^3 + ... + gridSize^3$
sparseCellCounter	`uint32_t`	1

Table 4.2: Relevant data structures for point counting

- **countingGrid**
  Initially, the point cloud is divided into several cubic chunks (see Figure 4.3). This fragmentation is done by placing a 3-dimensional grid with equal side length over the whole point cloud and by assigning points to the grid cells. For this purpose, the amount of points inside the grid cells have to be evaluated. The side length of this grid is referred to as *chunkingGridSize* (e.g. 4 in Figure 4.3) and allows a conclusion about the number of octree levels. If *chunkingGridSize* is, for example, 512 (that means 512 cells per side), then the following equation is true:

  $$2^{levels^3} = 512^3 \tag{4.7}$$

  Solved for *levels* this results in: $levels = 9$ and an octree that will have 9 levels starting from root node with level 0. Each of these octree levels correspond to one level of detail. The counting grid corresponds to potential octree nodes at the highest level of detail, and the root corresponds to the lowest level of detail. During the node-merging phase, point counts are summed up level-by-level starting from the initial counting grid. Thus, a counting grid has to exist for every octree level. The actual propagation is done in an *image pyramid* like fashion. This means that every grid in a lower pyramid (octree) level has a resolution lowered by a specific factor [AAB+83]. As the goal is to create an octree, this factor is set to 2^3. Thus, if the grid at the highest LOD level (bottom of the octree) has the resolution of 512^3, the grid of the next lower LOD level has a resolution of 256^3. To simplify data handling, all of the grids are stored in one array named *countingGrid* on the GPU during the Preparation phase (see Section 4.2). Thus, the size of *countingGrid* is for example: 512^3+ 256^3 + ... + 1^3 (for an octree with 9 levels).

- **denseToSparseLUT**
  The *denseToSparseLUT* is a lookup table that provides a mapping from the dense voxel coordinate of the grid to the index of the encompassing octree node. This is necessary as PotreeConverterGpu generates a sparse octree for faster processing and for saving GPU memory. In a sparse octree, an internal node may have up to 8 child nodes, but not necessarily. The idea is that the octree should only contain relevant nodes and avoid empty nodes if possible. The *denseToSparseLUT* allows accessing correct nodes in constant time, which requires just a single look-up instead of an octree traversal. To detect if a dense-to-sparse mapping exists for a given cell, *denseToSparseLUT* is initialized with -1 values.

- **sparseCellCounter**
  For the dense to sparse mapping, unique sparse indices have to be generated. The *sparseCellCounter* is a helper variable which is used to generate those indices, as well as to determine the sparse node amount of the octree.

4.3.1.1 Initial Point Counting

Figure 4.3 visualizes the initial subdividing of the point cloud, how point counts are stored and how cells are mapped from dense to sparse. The cells' point amounts are stored in the *countingGrid* at their dense indices. Each occupied cell gets an individual sparse index starting from zero. Afterwards, an entry with the cell's sparse index is stored in *denseToSparseLUT* at the cell's dense index position. The figure also shows how these nodes are stored continuously in the octree memory without any empty nodes in between. To explain the process of point counting in more detail, the point counting CUDA kernel is summed up in Algorithm 4.1. In contrast to PotreeConverter [Sch16], where counting is performed sequentially, counting in PotreeConverterGpu is done fully parallelized. Thus, instead of iterating through all the points sequentially, a CUDA thread is spawned for each point in the point cloud. After the actual point data is fetched from the cloud, it is mapped to 3d voxel grid coordinates, and afterwards to a linear cell index using the *mapPointToGrid()* function (see Algorithm 4.2). The evaluated cell index is then used to increment the counter at the respective cell (see line see lines 1-3). As all threads might access the *countingGrid* simultaneously, the increments have to be performed in an atomic way. Lines (4-6) create new sparse cells if the current point was the first to fall inside the dense grid. The old value of the *sparseNodeCounter* is used as the sparse index for the cell and stored in the denseToSparseLUT.

Algorithm 4.1: CUDA kernel: point counting

```
  /* Increment point count                                  */
1 point = cloud[index];
2 denseIndex = mapPointToGrid(point);
3 previous = atomicAdd(countingGrid[denseIndex], 1);
  /* Create dense to sparse mapping                         */
4 if previous == 0 then
5     sparseIndex = atomicAdd (sparseNodeCounter, 1);
6     denseToSparseLUT[denseIndex] = sparseIndex;
7 end
```

Algorithm 4.2: The mapPointToGrid() device function maps points to a 3D grid.

```
cellSize = boundingBoxSize / gridSize;
uX = (point.x - boundingBoxMin.x) / cellSize;
uY = (point.y - boundingBoxMin.y) / cellSize;
uZ = (point.z - boundingBoxMin.z) / cellSize;
ix = cast< $int64\_t$ > (fmin (uX, gridSize - 1.0));
iy = cast< $int64\_t$ > (fmin (uY, gridSize - 1.0));
iz = cast< $int64\_t$ > (fmin (uZ, gridSize - 1.0));
return cast< $uint32\_t$ > (ix + iy * gridSize + iz * gridSize * gridSize);
```

4.3.1.2 Point Count Propagation

After all point counts have been determined, we propagate them down to the root node. This is necessary to merge nodes with few points, and to determine how many potential sparse nodes the final octree will have. Assuming that the octree has two levels (exclusive root), as shown in Figure 4.4, the initial point counting is performed on the highest level with a $4 * 4 * 4$ grid. After that, the point counts are propagated to the next lower hierarchy level – a counting grid with the dimensions: $2 * 2 * 2$. This process is then continued until level *0* (corresponding to the root node of the octree) is reached. Count propagation is executed in parallel within a CUDA kernel for each cell of the counting grid for a specific level. This means that when propagating the point counts from, e.g,. *Level 2* (with $4 * 4 * 4$ cells) to *Level 1* (with $2 * 2 * 2$ cells), $2^3 = 8$ CUDA threads are spawned simultaneously. As this hierarchical spawning of kernels is a recurring task in PotreeConverterGpu, it is further explained in Algorithm 4.3.

The actual point count propagation is explained in Algorithm 4.4. In the first half (Lines 2-7) the dense indices of the matching eight cells in lower hierarchy level must be determined (Line 2). In a further step, the point counts of those cells are summed up and stored in the current cell of the counting grid. The remaining Lines 8-11 deal with the dense to sparse mapping as already known from Algorithm 4.1. Dense to sparse mapping only takes place if the propagated point amount is greater than zero. This is necessary to avoid empty nodes in the octree and thus to generate a sparse octree.

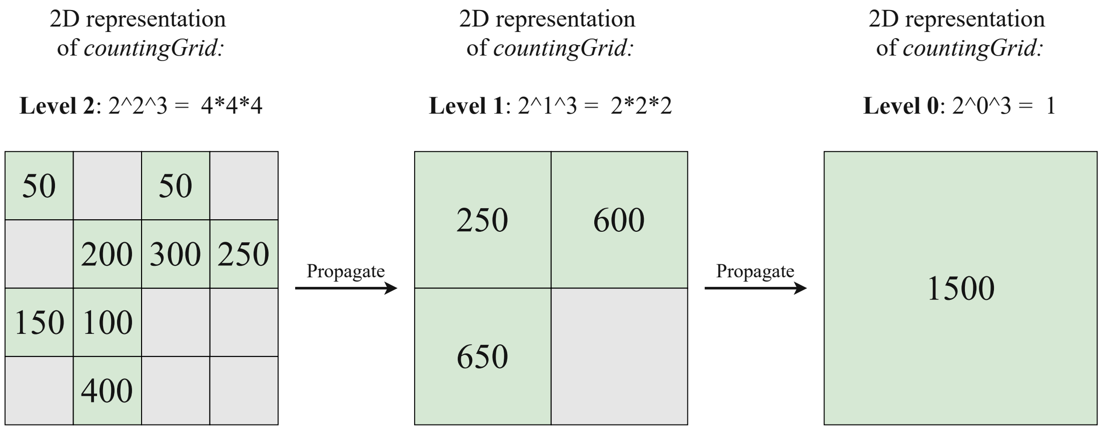


Figure 4.4: Hierarchical point count propagation: the figure shows a 2-dimensional representation of the *countingGrid* at three different levels. The point counts are propagated downwards, from the highest to the lowest level. The point count of the lowest cell is equal to the overall point amount in the point cloud. Occupied cells with a point amount greater than zero are highlighted in green.

Algorithm 4.3: Hierarchical kernel execution: shows how to call a CUDA kernel for each level in the octree, starting from the second highest. The number of spawned threads is equal to the number of possible (dense) nodes of a hierarchy level in the octree.

An example: $chunkingGridSize = 512$
Here kernel is executed 9 times (including the root level) spawning 256^3 threads, then 128^3 and so on.

```
gridSideLength = chunkingGridSize / 2;
while $gridSideLength > 0$ do
    numThreads = pow(gridSideLength, 3);
    cudaKernel<numThreads>();
    gridSideLength / 2;
end
```

Algorithm 4.4: CUDA kernel: point count propagation

```
Input   : level
/* Iterate over all children and sum up their point counts     */
pointSum = 0;
childIndices = getChildIndices();
for $i \leftarrow 0$ to 7 by 1 do
    childIdx = childIndices[i];
    pointSum += countingGrid[level + 1][childIdx];
end
/* Assign the point sum from the children to the actual cell   */
countingGrid[level][denseIndex] = pointSum;
/* Create dense to sparse mapping                              */
if $sum > 0$ then
    sparseIndex = atomicAdd(sparseNodeCounter, 1);
    denseToSparseLUT[denseIndex] = sparseIndex;
end
```

4.3.2 Node Merging

Variable	Datatype	Size
octree	Node[ ]	Amount of sparse cells
globalOffsetCounter	uint32_t	1

Table 4.3: Relevant data structures for node merging.

After all point counts have been determined and propagated, the actual octree structure can be generated. This requires to allocate the octree on the GPU device memory at first, and then to initialize all octree nodes. Initializing the octree is performed in a bottom-up approach and consists of two processing steps:

1. The leaf nodes in the newly allocated octree are initialised
2. Nodes with a point amount smaller than a *mergingThreshold* are merged. Merging of nodes is necessary to prevent many octree nodes with only few points in them, as this would negatively impact the performance during LOD generation as well as during rendering in Potree.

4.3.2.1 Octree Allocation and Initialization

As already explained in Section 4.3.1.1 and 4.3.1.2, a sparse index is generated for each non-empty cell during Point Counting. To generate these indices, the variable *sparseCellCounter* (see Table 4.2) stores the respective current number of sparse cells. For saving GPU memory, PotreeConverterGpu allocates a sparse octree instead of a dense octree. This octree is internally realized as an array of the *Node* structure which can be seen in Listing 4.1. The length of this array corresponds to the amount of sparse cells stored in *sparseCellCounter* and each entry corresponds to an octree node. The octree datastructure is referred to as *octree* and can be seen in Table 4.3.

```
struct Node
{
    uint32_t pointCount;
    uint32_t parentNode;
    bool isFinished;
    uint64_t dataIdx;
    int childNodes[8];
    bool isInternal;
};
```

Listing 4.1: The C++ Node struct

Each node stores a set of data properties which are described in Table 4.4. These properties store a reference to the actual cloud points and make it possible to traverse the octree, regardless if the octree data reside on host or device memory.

Variable name	Description
pointCount	The amount of points in the node
parentNode	Sparse index of the parent node in the octree
isFinished	Marks if the node is mergeable (false) or not mergeable / finished (true)
dataIdx	Determines the start position for the node's point data in the *outputBuffer*
childNodes	The sparse indices of all connected child nodes in the octree
isInternal	Marks if the node is an internal node or not

Table 4.4: The C++ Node struct: this table shows the layout of the C++ Node struct and explains its public accessible members.

Referring to Table 4.3, the following enumerations describes all relevant data structures necessary to explain the node merging phase:

- **octree**
  The actual sparse octree data structure. In contrast to other octree implementations such as [SOW20], the individual nodes are not linked directly via pointers, but store the position to connected nodes with the help of indices. This is a necessity

to ensure the proper function of node connections, even if the octree is copied from the device memory to the host memory or vice versa.

- **globalOffsetCounter**
  During Distribution phase, which is explained in Section 4.3.3, points are assigned to nodes. All of the distributed points are stored directly to the output buffer. To be able to read or write a node's point data from or to the buffer, each node gets a continuous amount of memory space within the buffer. This space is specified by an index (offset) which is calculated by the help of *globalOffsetCounter*.

4.3.2.2 Leaf Node Initialisation

In order to merge nodes starting from leaf nodes (Figure 4.5, nodes on highest hierarchy level), it is necessary to initialise those leaf nodes first. This initialisation process is executed in parallel in a CUDA kernel. Since the number of sparse leaf nodes is unknown, one CUDA thread is created per dense leaf node. For example: if *chunkingGridSize* is 512, then the theoretical amount of dense nodes is 512^3.

Algorithm 4.5 gives a short overview about the initialisation process for one node. First, the node's sparse indexed is looked up. If this is -1, then no sparse index exists and therefore no corresponding sparse node. The thread will be terminated immediately. Otherwise, the sparse node is fetched from *octree* and its *pointCount* is set to the value stored in the corresponding *countingGrid* cell (Lines 5 and 6).

Algorithm 4.5: CUDA kernel: initialization of leaf node for sparse octree

```
1 sparseIndex = denseToSparseLUT[denseIndex];
2 if sparseIndex == -1 then
3     return;
4 end
5 node = octree[sparseIndex];
6 node.pointCount = countingGrid[denseIndex];
```

4.3.2.3 Hierarchical Node Merging

As already mentioned, node merging is for combining nodes with too few points into one bigger node. Another important task performed in this phase is initialising all internal nodes and connecting them to their child nodes. Furthermore, the memory location for each node's point data is also determined.

The cell merging algorithm is summarised in Algorithm 4.6. For better understanding the algorithm is explained together with Figure 4.5. The complete merging is performed

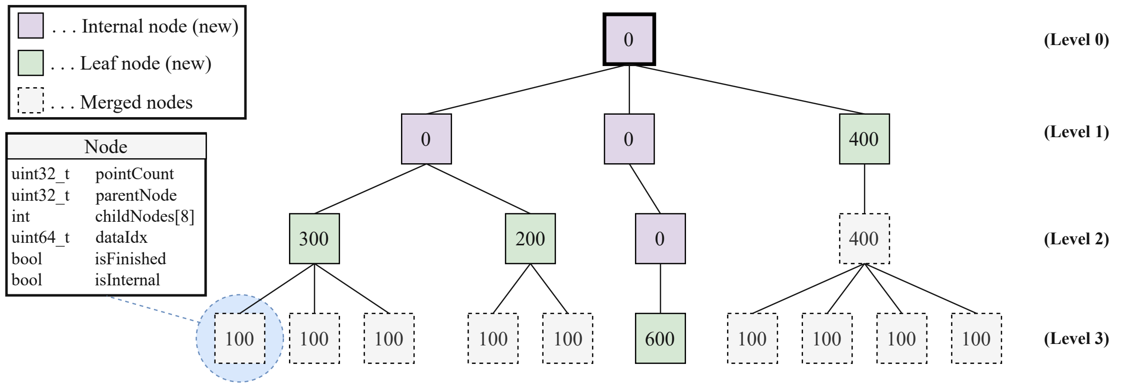


Figure 4.5: Hierarchical node merging: the *mergingThreshold* is set to 500 in this example. Octree nodes containing too few points (< 500) are merged, whereas nodes with enough points are marked as finished. Those finished nodes act as new leaf nodes. Later, during Point distribution phase, the points from the point cloud are assigned to those leaf nodes. Point counters in internal nodes are set to zero and will be populated later on during the subsampling phase.

in a single CUDA kernel. This kernel is executed hierarchically for each octree level, starting from one level above the bottom (level 2 in the Figure). Starting the kernel is the same as in Section 4.3.2.2, i.e., in each level, a CUDA thread is started for each dense node. The complete algorithm can be divided into 5 steps:

1. **Lines 1-3**:
   The first step is again to look up the sparse index for the current node and to check if the entry exists. If this is not the case, the thread terminates immediately.

2. **Lines 5-8**:
   Next, all relevant data is fetched. This means that both, the propagated number of points (from *countingGrid*), as well as all child nodes are determined. In order for Potree to be able to spatially index the child nodes during rendering, it is necessary to store them in a certain order in the parent node. Therefore, *getChildNodes()* follows a strict mapping from the logical 3D space to the physical 1D space in memory, as shown in Figure 4.6.

3. **Lines 9-10**:
   After data fetching, the actual node can be initialised. The first step is to determine whether the current node is already finished. This is the case if its corresponding entry in *countingGrid* is greater than a certain *mergingThreshold*. Depending on this, *initializeNodes()* sets the node attributes *isInternal* and *isFinished* to true or false. The child nodes' indices are also stored in the node's *childNodes* attribute. If a child does not exist, -1 is stored. Depending on whether the node is finished, its *pointCount* is set to 0, otherwise to the current value in *countingGrid*. Nodes that are marked as finished are not mergeable any more.

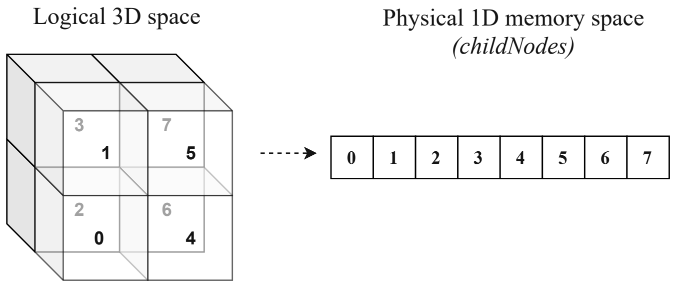


Figure 4.6: Child node mapping: from logical 3D space to physical memory space. Child nodes are stored in their parent's node in this order.

4. **Lines 11-14**:
   The next step is to connect the child nodes to their parent node. This is done within the *connectChild()* function by setting every child node's *parentNode* attribute to the parent sparse index. Further, the *isFinished* attribute is set in every child node. To be able to determine the data position in the next step, it is necessary to accumulate all child node point counts in *pointSum*.

5. **Lines 16-20**
   In a final step, the global position in the *outputBuffer* is determined for each child node (*dataIdx*) if the actual node is considered as finished. As one can see in Figure 4.7, an internal node's child nodes reside continously in the *outputBuffer*, starting at *globalOffsetCounter*. To evaluate the current value of *globalOffsetCounter* and to add an additional offset (*pointSum* of all child nodes), *globalOffsetCounter* is incremented using CUDA's `atomicAdd` function. Using this approach it is possible to generate a global position for each node, even if multiple threads try to access *globalOffsetCounter* in parallel. The current value of *globalOffsetCounter* is buffered in *globalDataIndex*. This variable is used to assign a *dataIdx* for each child node. After each assignment, *globalDataIndex* is incremented by the child's *pointCount*.

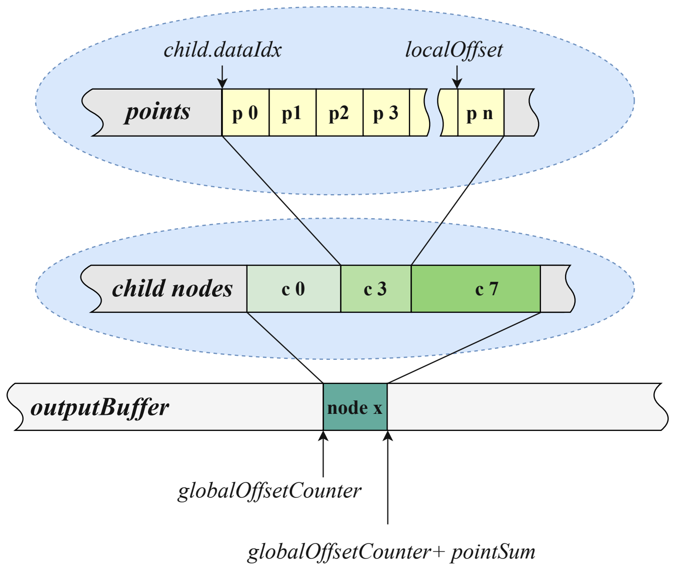


Figure 4.7: Child nodes in output buffer: an internal node's existing child nodes are stored continuously in a block of memory in the *outputBuffer*. This memory block starts at *globalOffsetCounter and ends on globalOffsetCounter + pointSum*. Each child consists of a number of points which are linearly stored starting at the child's *dataIdx*. Thus a point's position can be described as *child.dataIdx + localOffset*.

Algorithm 4.6: CUDA kernel: node merging

```
/* 1. Check if sparse node exists                                  */
sparseIndex = denseToSparseLUT[denseIndex];
if $sparseIndex == -1$ then
    return;
end
/* 2. Fetching data                                                */
pointCount = countingGrid[denseIndex];
node = octree[sparseIndex];
childDenseIndices = getChildIndices();
childNodes[ ] = getChildNodes(childDenseIndices);
/* 3. Initializing node                                            */
isFinished = (pointCount >= threshold);
initializeNode(node, childDenseIndices, isFinished);
/* 4. Initializing child nodes                                     */
pointSum = 0;
foreach $child \in childNodes$ do
    connectChild(child, node);
    pointSum += child.pointCount;
end
/* 5. Assign data index                                            */
if $isFinished$ && $pointSum > 0$ then
    globalDataIndex = atomicAdd (globalOffsetCounter, pointSum);
    foreach $child \in childNodes$ do
        child.dataIdx = globalDataIndex;
        globalDataIndex += child.pointCount;
    end
end
```

4.3.3 Point Distribution

After the merging phase, the octree is built and the position for writing point data to the *outputBuffer* per node is evaluated, but no points are actually assigned to the nodes. During distribution, we will transfer all points to the memory regions that were reserved for each of the leaf nodes. After this pass, all leaf nodes' points are exported to the *outputBuffer* (see Section 4.5). To be able to access the original point data within the source cloud, the point indices are also assigned to each node. This is necessary to be able to access the original points rather than the quantized and exported points during subsampling to preserve floating-point precision.

As processing is done in-core, it is absolutely necessary to minimize the memory consumption. Thus, storing original point data for each octree node should require a minimum of GPU memory and should avoid physical memory copies. Rather than to store and copy point data entries for each node, the indices of the points in the original point cloud are stored in a separate lookup table, here referred to as $pointLUT$. Applying this approach requires an additional register lookup for each point access afterwards but reduces the memory consumption by up to 85 %. Assuming a point cloud with 3 *double* coordinates and 3 *uint8_t* color values, the necessary memory amount to store such a single point is:

$$memory = 3 * (sizeof(double) + sizeof(uint8\_t)) = \textbf{27 byte} \tag{4.8}$$

Referring to Section 3.1, one of the requirements for this master thesis is to process up to 100 millions of points. Thus, the lookup table necessary to store the points' indices, can be of type *uint32_t*, as it would be theoretically possible to index $2^{32} = 4{,}294{,}967{,}296$ points, which is more than enough. The necessary amount of memory to store a point's index is therefore only 4 bytes, resulting in a memory saving of 85 %. Referring to Table 4.5, allocating the $pointLUT$ requires to use the empirical *outputFactor*. This factor has already been used and explained in Section 4.2 during *outputBuffer* allocation. To be able to assign each point within a node to a memory space in parallel, $localOffset$ keeps track of these local memory offsets.

Variable	Datatype	Size
pointLUT	uint32_t[ ]	pointAmount * *outputFactor*
localOffset	uint32_t[ ]	Amount of sparse cells

Table 4.5: Relevant data structures for point distribution

Algorithm 4.7 shows the single steps which are performed during Point distribution. In contrast to [SOW20], where points are distributed sequentially, PotreeConverterGpu distributes all points in parallel. For this, a single CUDA thread is started for each point in the source point cloud, executing the following operations:

1. **Lines 1-3:**
   At first, the point with a specific index is fetched from the cloud and mapped to a dense position within the 3D grid of the lowest octree level. Additionally the sparse index is also looked up.

2. **Lines 4-7:**
   All points should be stored in a leaf node. Since nodes can be merged, it is not possible to simply store the point in the originally mapped node. Instead, it must be evaluated whether the respective node represents a valid leaf node in the octree (see Figure 4.5). If this is not the case, the leaf node must be searched for iteratively. For this, first the mapped node is assumed to be the *targetNode*. Then it is checked whether *isFinished* is true in the *targetNode*. If this is true, then *targetNode* is a valid leaf node and the search can be finished. However, if *isFinished* is false, the parent node of the examined node is determined, assumed as *targetNode* and the same check is performed. This process is repeated until a valid *targetNode* is determined.

3. **Line 9:**
   After the target node has been determined, the output position ($outputPosition$) for each point can be calculated as follows:

   $$outputPosition = globalOffsetCounter + localOffset \tag{4.9}$$

   In this equation *globalOffsetCounter* is the node's assigned *dataIdx* and the localOffset is the position of the point within the node. $localOffset$ is calculated by atomically incrementing the $localOffset$ for the actual node. Thus, its possible to calculate the $outputPosition$ for every point in parallel.

4. **Lines 10-11:**
   The last step is the actual distribution of points using the previously calculated *outputPosition*. To be able to access original point values later on, the points' indices are stored in a lookup table called $pointLUT$. Furthermore, the points are also exported to the *outputBuffer*. For this purpose the coordinate and color values are quantized first and then written to the buffer at the *outputPosition*. The quantization of exported point data is described in particular in Section 4.5.2.

Algorithm 4.7: CUDA kernel: point distribution

```
    Input   : index
    /* Initialising                                             */
 1  point = cloud[index];
 2  denseIndex = mapPointToGrid(point);
 3  sparseIndex = denseToSparseLUT[denseIndex];
    /* Searching for target node                                */
 4  targetNode = octree[sparseIndex];
 5  while !targetNode.isFinished do
 6      sparseIndex = octree[sparseIndex].parentNode;
 7      targteNode = octree[sparseIndex];
 8  end
    /* Evaluate output position                                 */
 9  outputPosition = node.dataIdx + atomicAdd(localOffset[sparseIndex], 1);
    /* Store point data                                         */
10  pointLUT[outputPosition] = index;
11  outputBuffer[outputPosition] = quantizePoint(point);
```

4.4 Point Subsampling

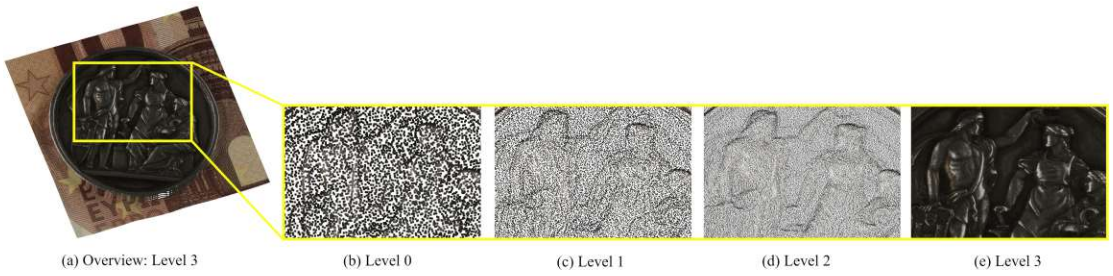


Figure 4.8: ICI Coin dataset: different levels of detail generated through subsampling. The resolution of the point cloud increases with each higher octree level.

Subsampling is the process of extracting point samples from finer levels to populate the lower levels of detail. At this point within the PotreeConverterGpu processing pipeline, all points reside within the octree's leaf nodes. During subsampling, the remaining octree nodes are filled with points and the actual LOD structure is finished. Subsampling is done in a bottom-up way and for each internal node separately. Afterwards, viewed from the root node (lowest LOD) the level of detail increases with each higher octree level (see Figure 4.8). Basically, subsampling works by visiting all internal nodes within the octree, starting from the bottom. Then, for each internal node, points within their direct child nodes are chosen and stored within the parent. As PotreeConverterGpu implements the replacement LOD scheme (see Section 2.3.3.1), the selected points are copied to the parent node. This approach ensures a higher sampling rate and helps to counteract aliasing artifacts during rendering. Continuing this process up to the root node creates the final LOD structure.

In PotreeConverterGpu, points are subsampled using a *random subsampling* method that is inspired by the *uniform random subsampling* approach from [KJWX19] and is applied in [SOW20]. In detail, a $128 * 128 * 128$ grid – the *subsamplingGrid* with the size of the internal node's bounding box – is placed over all direct child nodes. Afterwards, a single point within each grid cell is chosen randomly and propagated upwards. In PotreeConverterGpu we distinguish between *implicit* and *explicit* random subsampling. Implicit random subsampling utilizes the undeterministic order in which CUDA spawns the threads to choose a random point. Explicit random subsampling on the other hand, chooses points based on generated random numbers.

One of the novelties introduced by this master thesis is that subsampling is combined with a color filtering strategy for point clouds. This approach was already suggested in [RL01] and [WBB$^+$08] and has been introduced in Section 2.3.3. The basic concept behind this idea is to generate lower LODs using a replacement scheme (see Section 2.3.3.1) in combination with color filtering for subsampled points. In PotreeConverterGpu we distinguish between *intra-cell* and *inter-cell* color filtering. Whereas the intra-cell approach considers only points within the boundaries of a *subsamplingGrid* cell during color filtering, the inter-cell approach also considers points from neighbouring cells. In the implementation, random subsampling is performed within the *randomSubsampling* function. This function is displayed simplified in Algorithm 4.8. As one can see in the Lines 4-11, the following processing steps are performed sequentially for each internal node within the octree:

1. **Line 5**: First, the bounding box for the actual internal node is calculated to be able to create an appropriately sized *subsamplingGrid*. Calculating the bounding box is crucial for creating the *subsamplingGrid*, which is a virtual 3-dimensional regular grid. The idea is to place the grid over the currently processed node and to extract one point per grid cell from the child nodes. The world-size of the *subsamplingGrid* is equal to the actual node's bounding box. In PotreeConverterGpu, a default amount of 128^3 cells is considered for the grid. Thus, a maximum amount of $128^3 = 2{,}097{,}152$ points can be subsampled per internal node, but in practice the amount of sampled points is closer to 128^2 due to the surfacic nature of the (non-volumetric) input data.

2. **Line 6**: The CUDA kernel *kernelEvaluateSubsample* evaluates subsample data for the given internal node within its child nodes. Furthermore, if activated, intra-cell color filtering is also performed within this step. If implicit random subsampling is performed without any color filtering, this step can be omitted. A detailed explanation can be found in Section 4.4.2.

3. **Line 7**: The CUDA kernel *kernelInterCellColorFiltering* performs inter-cell color filtering, which means to average color information over the boundaries of the *subsamplingGrid* cells. Section 4.4.2.1 explains this concept in detail.

4. **Line 9**: The CUDA kernel *kernelGenerateRandom* generates a single random number for each occupied *subsamplingGrid* cell. The generated numbers act as point indices for the explicit random subsampling approach, which picks a single random point per cell based on these indices. The CUDA kernel is described in particular in Section 4.4.4.

5. **Line 11**: The CUDA kernel *kernelPointSubsampling* performs the actual subsampling of the previously selected points. This includes propagating the points upwards to the actual internal node, quantizing of (averaged) color and point values and exporting the points to the *outputBuffer*. This is the last processing step within the random subsampling approach and can be found in Section 4.4.5.

PotreeConverterGpu's subsampling phase can be parametrized to perform either implicit or explicit random subsampling and further to execute intra-cell, inter-cell or no color filtering. Thus, it is not always necessary to execute all of the described processing steps in Algorithm 4.8. Table 4.6 shows which processing steps (CUDA kernels) are executed under which subsampling configuration.

	Implicit			Explicit		
CUDA kernel	**Intra**	**Inter**	**None**	**Intra**	**Inter**	**None**
kernelEvaluateSubsamples	yes	yes	no	yes	yes	yes
kernelInterCellColorFiltering	no	yes	no	no	yes	no
kernelGenerateRandoms	no	no	no	yes	yes	yes
kernelPointSubsampling	yes	yes	yes	yes	yes	yes

Table 4.6: Executed CUDA kernels during implicit and explicit random subsampling when either intra-/inter-cell or no color filtering is performed.

Algorithm 4.8: Random subsampling overview: the *randomSubsampling()* function recursively executed for each octree node, starting at the root node

```
Input  : node, level, isExplicit
/* Call order:  post-order depth-first                          */
foreach $child \in node$ do
    randomSubsampling(child, level+1);
end
/* Node visiting & subsampling                                   */
if $node.isInternal$ then
    boundingBox = calcNodeBoundingBox(node);
    kernelEvaluateSubsamples(...);
    kernelInterCellColorFiltering(...);
    if $isExplicit$ then
        kernelGenerateRandoms(...);
    end
    kernelPointSubsampling(...);
else
    return;
end
```

4.4.1 The Subsampling Order

According to Section 4.4, subsampling is the process of extracting point samples from finer levels and populating them upwards. As a result, an LOD structure in which the level of details increases from top (root node) to bottom (leaf nodes) is created. To construct this LOD structure correctly, it is necessary to subsample octree nodes in a specific order. In PotreeConverterGpu, a post-order depth-first traversal is applied to create the subsamples. These are created in a bottom-up way starting with the highest level of detail at the octree bottom. To achieve this kind of subsampling order it is necessary in a first step to go from the root node downwards to find the lowest possible internal node. This can be seen in Algorithm 4.8 at Lines 1-3, where *randomSubsampling()* is called recursively for all child nodes of a internal starting at the root node. Figure 4.9 visualizes

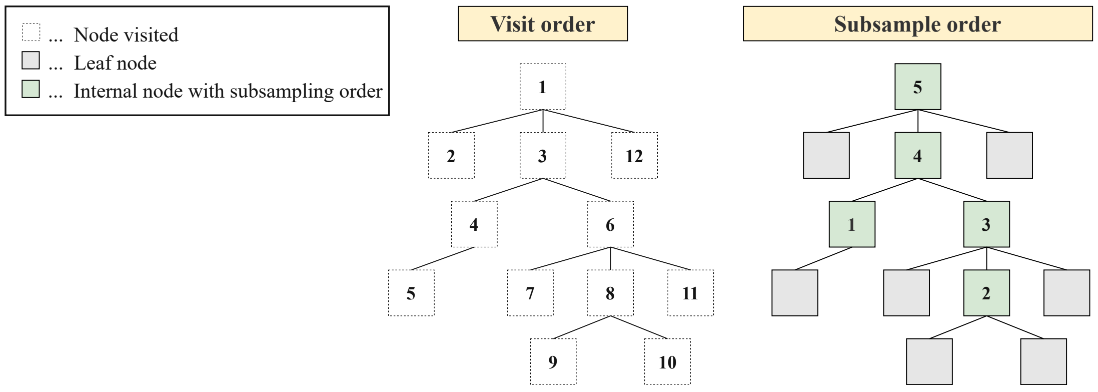


Figure 4.9: Octree traversal order: the image on the left shows the order in which nodes are visited. However, leaf nodes are not directly processed. The subsample procedure is only called for internal nodes in the order shown at the right. Each internal node's direct children, which can be leaf nodes or previously populated internal nodes, are the input, and the output is the subsample for the currently processed node.

the order in which nodes are visited at the left side. When the program actually reaches Line 4, the *node* variable points either to a leaf or an internal node. If it is a leaf node nothing happens, otherwise random subsampling is performed.

4.4.2 Subsample Evaluation and Intra-cell Color Filtering

Subsampling in PotreeConverterGpu works by placing a 3-dimensional grid – the *subsamplingGrid* – (128^3 cells per default) over the currently processed node and extract one point per cell from all the points of all child nodes (see Figure 4.10). One of the novelties within this master thesis over other layered-point-cloud-based approaches is that the RGB color values of the subsampled points do not correspond to their original values, but are created by averaging the color values of surrounding points – similar to the texture filtering in mip mapping. In particular, we distinguish between intra-cell and inter-cell color filtering. Intra-cell color filtering averages colors of points within the same cell, whereas inter-cell color filtering also takes adjacent cells into account. This section focuses on intra-cell color filtering.

Subsample evaluation is done within a single CUDA kernel (*kernelEvaluateSubsamples*). During this phase, the *subsamplingGrid* is placed over the currently processed node's child nodes (max. 8 nodes) and multiple steps are performed according to the subsample configuration:

- Point counting per grid cell
- Intra-Cell color filtering: encoding and accumulating all points mapped to a cell

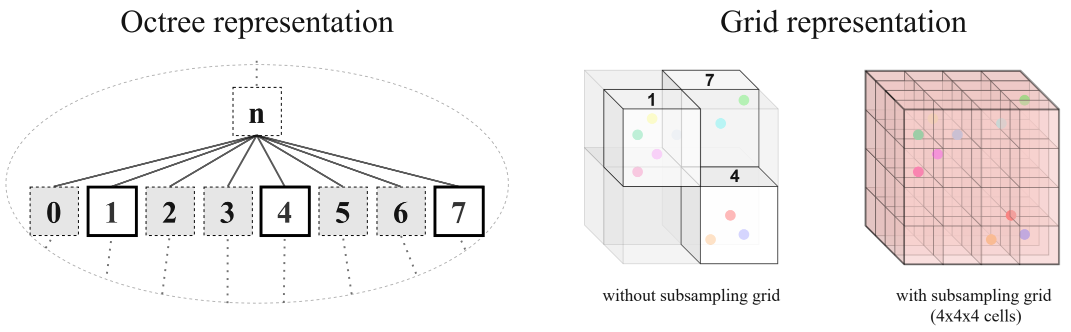


Figure 4.10: Subsampling overview: this conceptual representation shows the basic Subsampling procedure: a node's existing child nodes (missing nodes are displayed in grey) are mapped from octree space to grid space. Afterwards a regular Subsampling grid (viewed in red) is placed over those nodes to extract points.

- Selecting the subsample points (only when implicit random subsampling is performed)

The point counting works the same way as during the Chunking phase described in Section 4.3.1. Each point within the child nodes is mapped to a cell of the subsampling grid and a value within the corresponding *countingGrid* is incremented. The point amount per cell is crucial to be able to generate a random number for the explicit random subsampling and to calculate color averages. During Intra-cell color filtering the color values from all points within a grid cell are accumulated within a corresponding *averagingGrid*. If implicit random subsampling is performed, the points for subsampling (one point per cell) are also chosen within the kernel.

4.4.2.1 Intra-Cell Color Filtering

This Section gives a detailed explanation about the intra-cell color filtering approach applied in PotreeConverterGpu. The implementation details are explained in the next Section 4.4.2.2. The principle of the color filtering approach is based on [RL01] and [WBB$^+$08] and has already been explained in Section 2.3.3. The idea of this approach is to combine color values of merged/downsampled points to produce a blended color value that is representative of the chosen but also the discarded points. This procedure is similar to the mip-map generation for textures and aims to reduce aliasing artifacts, which result from a too low subsampling rate during LOD generation.

Figure 4.11 visualizes the basic subsampling procedure together with the intra-cell color filtering approach. In detail, an octree is displayed with 3 levels, where r is the root node and $r0$ is one of its child nodes. $r0$ itself has also four child nodes which have a total point amount of 18. According to the subsampling order (see Section 4.4.1) $r0$ is the first internal node for which subsamples are generated. To perform subsampling on $r0$'s child nodes a *subsamplingGrid* is generated and placed over the child nodes. In

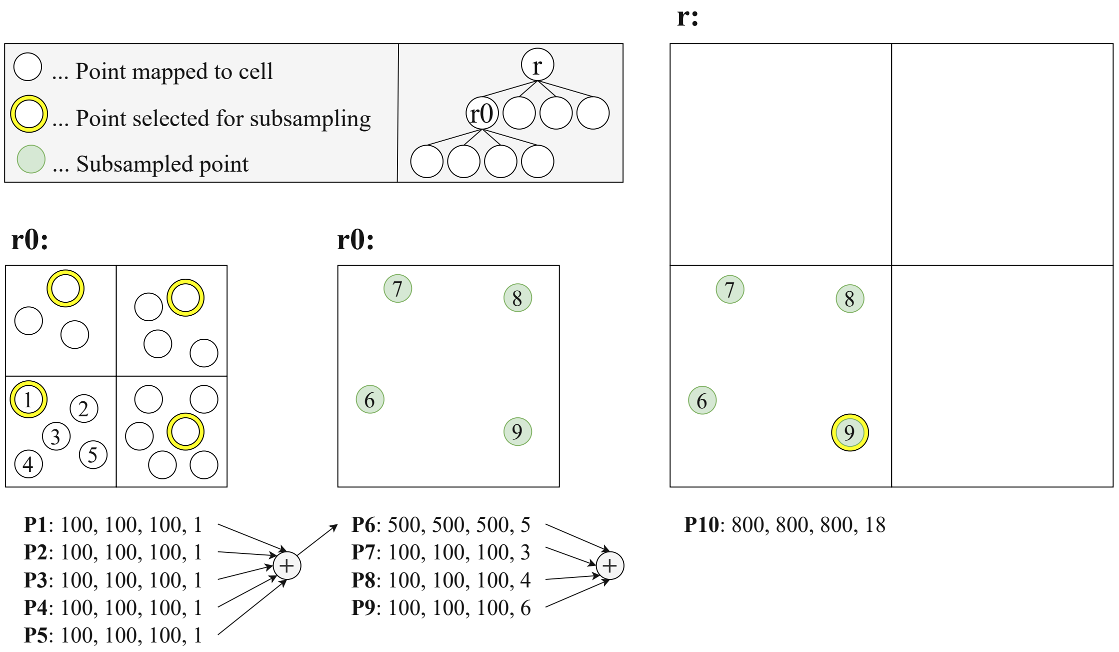


Figure 4.11: Intra-cell color filtering: the figure shows a 2-dimensional example of the intra-cell color filtering approach. A $2 * 2$ subsampling grid is placed over $r0$'s child nodes and one represantative point is chosen per cell. The color values of all points inside a cell are accumulated and assigned to the subsampled point.

the 2-dimensional example, this grid has a dimension of $2 * 2$. After determining the target cell for each point, the color values of all points inside a cell are accumulated. The example figure shows how the color values of $P1 - P5$ in the left bottom cell are summed up. In general terms, summing color values ($pointColor_n$) to a single $accumulatedColor$ can be expressed as follows:

$$accumulatedColor = \sum_{i=1}^{pointsInCell} pointColor_i \tag{4.10}$$

Afterwards, a single representative point is chosen per cell (marked with a yellow circle). These points form the actual subsampling points. Thus, their coordinate values are copied and stored inside $r0$. Instead of assigning the points' original color values to the subsampled points, the $accumulatedColor$ values are stored together with the amount of points per cell as a tuple of the form: $(accR, accG, accB, pointAmount)$. In the case of $r0$, exactly four points are subsampled which form $P6 - P9$. Afterwards the subsamples for r are generated. In this case, $r0$ itself is a child node. Thus, the color values of the newly created points $P6 - P9$ are accumulated and one representative point ($P9$) is chosen. This forms another subsampling point $P10$ which has an RGB color value of $800, 800, 800$ and an accumulated point amount of 18 which is equal to the amount of points in $r0$'s child nodes. The actual color value of a subsampled point ($averagedColor$)

is evaluated as follows:

$$averagedColor = \frac{accumulatedColor}{pointsInCell} \tag{4.11}$$

The intra-cell color filtering implementation within PotreeConverterGpu differs slighty from the previous explanation. For simplicity and to reduce memory usage, our actual implementation does not propagate the sums of colors and point counts through the hierarchy. Instead, it directly assigns the average color value inside the subsampled points. Subsampling in lower LODs therefore treats points in child nodes with the same weight, even though some points might represent the average of a larger amount of points than others.

4.4.2.2 Implementation Details

Variable	Datatype	Size
averagingGrid	uint64_t[ ]	Amount of subsampling grid cells

Table 4.7: Relevant data structures for subsample evaluation and intra-cell color filtering

Section 4.4.2.1 provided a detailed overview about the intra-cell color filtering algorithm. This section focuses on the implementation details and describes how subsample evaluation in combination with intra-cell color filtering is applied in PotreeConverterGpu.

Both steps are performed in a single CUDA kernel: *kernelEvaluateSubsamples*. Based on the actual subampling configuration (implicit/explicit random subsampling) and color filtering approach (intra-cell/inter-cell) a different version of the kernel is executed. The actual mutations of *kernelEvaluateSubsamples* are displayed in Figure 4.12. As described in the flow chart, the encoded and accumulated color values are stored in *averagingGrid* (see Table 4.7). This data structure is allocated on the GPU memory during the Preparation phase (see Section 4.2) and allows to accumulate color values per *subsamplingGrid* cell. Under specific configurations it is required to explicitly store the amount of points per *subsamplingGrid* cell in a separate data structure. For this purpose, the *countingGrid* from the Point Counting phase (see Section 4.3.1) is reset and reused. Furthermore, one can see that the actual selection of the subsample points is also performed in the kernel if intra-cell color filtering is active.

kernelEvaluateSubsamples is explained according to Algorithm 4.9, which provides a generalized overview. The CUDA kernel has to be executed for every point in each of the internal node's child nodes. Referring back to Figure 4.11, in this example the kernel would have to be executed for each of the 18 points in the child nodes of $r0$. Given the sparse *nodeIdx* and the local *pointIdx* within the child node the following Listing explains the CUDA kernel within Alogrithm 4.9 in detail:

- **Lines 1-5**: These lines are essentially equal in all versions of *kernelEvaluateSubsamples*. First, the actual parent (internal) and child nodes are retrieved from the octree.

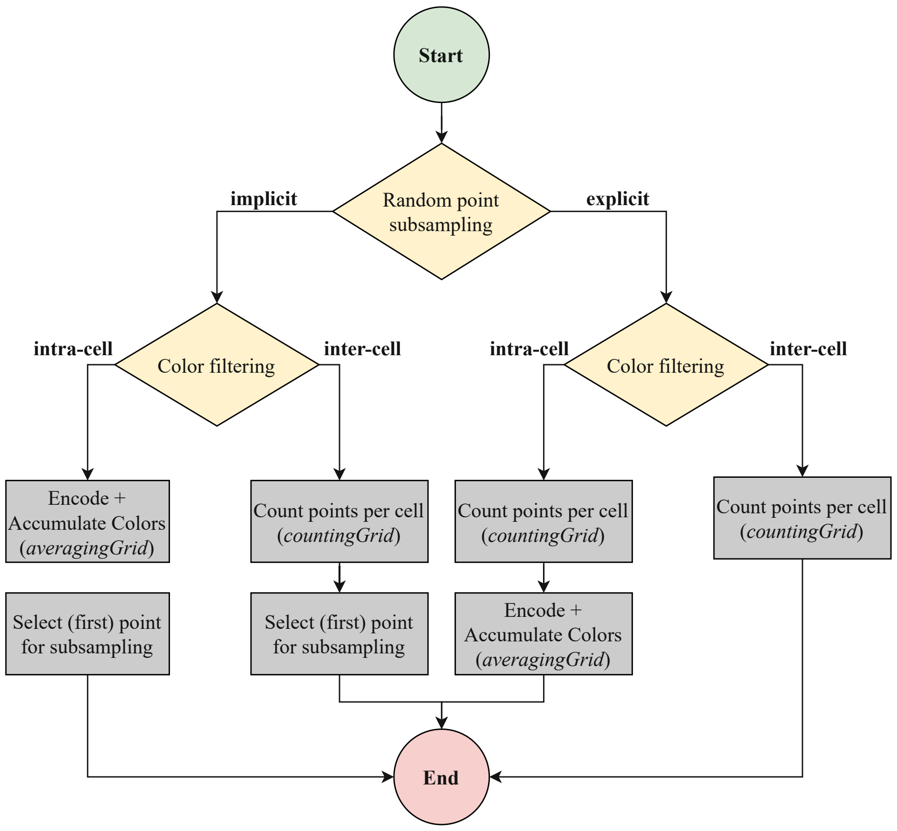

Start
Random point subsampling
implicit
explicit
intra-cell
Color filtering
inter-cell
intra-cell
Color filtering
inter-cell
Encode + Accumulate Colors (*averagingGrid*)
Count points per cell (*countingGrid*)
Count points per cell (*countingGrid*)
Count points per cell (*countingGrid*)
Select (first) point for subsampling
Select (first) point for subsampling
Encode + Accumulate Colors (*averagingGrid*)
End

Figure 4.12: Subsample evaluation flow chart: processing steps performed within the subsample evaluation based on the subsample configuration

Then the original point within the point cloud as well as the exported point data within the $outputBuffer$ are retrieved. Getting the original point data is necessary, as we want to map the point to a *subsamplingGrid* cell with the original coordinate floating-point precision. Accessing the exported point is required to get access to the exported (averaged) point colors. The position of a point's data can be retrieved from the *pointLUT* (see Figure 4.15).

- **Line 6**: The actual point is mapped to a cell within the *subsamplingGrid*. This generates a $denseIndex$ for the mapped cell, which is used to index the *averagingGrid* and the *countingGrid*.

- **Lines 7-9**: If intra-cell color averaging is performed, the point's RGB color values are encoded into a single `uint64` value. Afterwards this value is added to the cell's accumulated color values stored in *averagingGrid*[$denseIdx$]. As multiple threads might try to accumulate *averagingGrid* simultaneously, this addition is done atomically.

- **Lines 11-12**: If explicit random subsampling is performed, it is necessary to additionally count the points per grid cell and to store the result in *countingGrid* to determine the random point afterwards. The same applies if implicit random subsampling in combination with inter-cell color filtering is performed. Incrementing the value in *countingGrid* is done atomically, as multiple threads might access the data structure simultaneously.

- **Lines 14-21**: These lines are only executed if the actual point is the first point in the cell, determined by the *old* value.

  - **Lines 15-16**: The parent node's point counter is incremented. This operation is done atomically as multiple threads might access the parent node simultaneously. The *old* value returned by `atomicAdd` corresponds to the sparse index of the actual cell and is stored in $denseToSparseLUT$. This is necessary to be able to find the unique consecutive writing position in the $outputBuffer$ afterwards.
  - **Lines 17-18**: The idea of implicit random subsampling is to select the first point per *subsamplingGrid* cell. Thus, if implicit random subsampling is performed, the actual point is considered to be the selected one and the point's index is stored in $pointLUT$.

For intra and inter-cell color filtering, exported point RGB color values are re-imported and are encoded into a single `uint64` value. This is necessary to be able to easily accumulate color values within the *averagingGrid* using a single CUDA `atomicAdd` instead of three. The color encoding is implemented within a CUDA `device` function and is explained in Listing 4.2. The idea is to encode the color values plus the point count per cell in the `uint64` value using an $RGBA(18Bit - 18Bit - 18Bit - 10Bit)$ schema, where the last 10 Bits are occupied for the counter. Applying this schema allows to summarize the color values of $2^{10} - 1 = 1023$ points per cell. To pack the four values into a single one, the RGB values are bit-shifted to the left respectively. Afterwards the RGBA values are combined into a single `uint64` by applying the logical OR operator.

```
__device__ uint64_t encodeColors (uint16_t r, uint16_t g, uint16_t b)
{
    return
    static_cast<uint64_t> (r) << 46 |
    static_cast<uint64_t> (g) << 28 |
    static_cast<uint64_t> (b) << 10 |
    static_cast<uint64_t> (1);
}
```

Listing 4.2: CUDA device function for packing an RGB color together with a point count into a single uint64 value.

Algorithm 4.9: CUDA kernel: subsample evaluation and intra-cell color filtering. The kernel is executed for each point in a node's child nodes.

```
   Input   : parentIdx, childIdx, pointIdx, isIntra, isImplicit
   /* Get the octree node                                          */
1  parent = octree[parentIdx]
2  child = octree[childIdx]
   /* Get original and exported point data                         */
3  pointdataIdx = pointLUT[child.dataIdx + pointIdx]
4  originalPoint = cloud[pointdataIdx]
5  exportedPoint = outputBuffer[pointdataIdx]
   /* Map point to cell and get dense index                        */
6  denseIndex = mapPointToGrid(originalPoint)
   /* Encode point colors and accumulate them for intra-cell color
      filtering                                                    */
7  if isIntra then
8      encoded = encodeColors (exportedPoint.r, exportedPoint.g, exportedPoint.b)
9      old = atomicAdd (averagingGrid[denseIndex], encoded)
10 end
   /* Increase point count in cell and create dense-to-sparse entry */
11 if !isImplicit or !isIntra then
12     old = atomicAdd (countingGrid[denseIndex], 1)
13 end
   /* The actual point is the first in the cell                    */
14 if old == 0 then
       /* Create dense-to-sparse entry                             */
15     sparseIndex = atomicAdd (parent.pointCount, 1)
16     denseToSparseLUT[denseIndex] = sparseIndex
       /* Store subsampled point index in pointLUT                 */
17     if isImplicit then
18         dst = parent.dataIdx + sparseIndex
19         pointLUT[dst] = pointdataIdx
20     end
21 end
```

4.4.3 Inter-Cell Color Filtering

In this master thesis we provide another novelty and extend the intra-cell color filtering approach from [WBB$^+$08] to work across cell boundaries. This approach is referred to as inter-cell color filtering and is visualized in Figure 4.13. The goal is to further improve anti-aliasing for lower LODs during rendering. Intra and inter-cell color filtering can be applied interchangeably. In this section we show that inter-cell filtering is highly

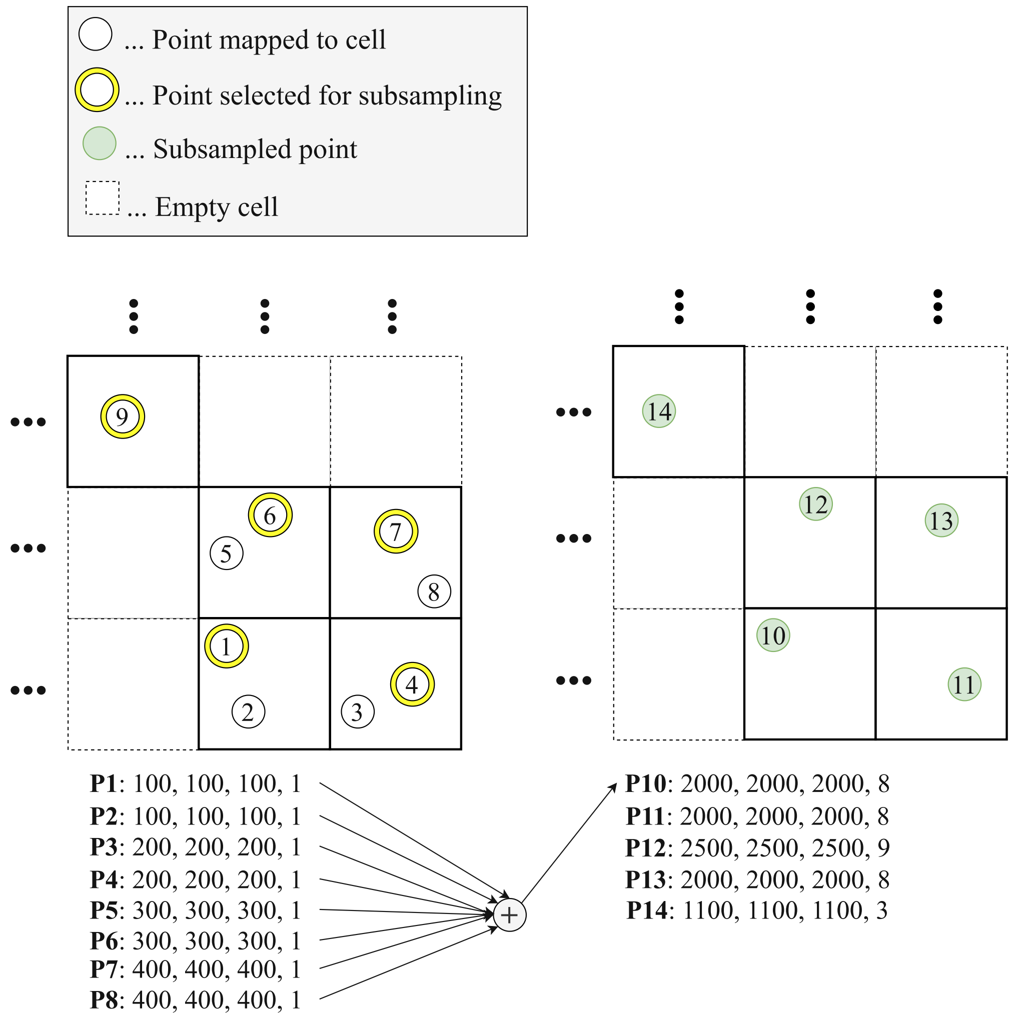


Figure 4.13: Inter-cell color filtering: compute the average color of all points in the current and adjacent cells. P10 comprises the sum of P1 to P8, but not P9 as it is not in an adjacent cell.

parallelizable on a GPU within a CUDA kernel – *kernelInterCellAvg*. Figure 4.13 shows a cutout of a two-dimensional *subsamplingGrid* placed over child nodes. The basic idea behind inter-cell color filtering is to sum up point attributes (colors) from all adjacent *subsamplingGrid* cells. Thus, the subsampled point $P10$ for example contains accumulated color values from the points $P1$-$P8$ but not from $P9$. The actual implementation is documented in Algorithm 4.10. The basic idea is that each point contributes its color value not just to its own but also to the adjacent cells. The reason behind it is that points near the border of a cell might be close to the subsampled point of an adjacent cell, and they should therefore also contribute. The following Listing explains Algorithm 4.10 in detail:

- **Lines 1-4**: These lines are basically identical to Lines 1-5 in Algorithm 4.8 (*kernelSubsampleEvaluation*). In fact they describe how the original point data within the point cloud, as well as the exported point data in the $outputBuffer$ are accessed.
- **Line 5**: The actual point is mapped to a *subsamplingGrid* cell. The cartesian coordinates of this cell are stored in ix, iy and iz. The original point data is used for the mapping to ensure the original floating-point precision.
- **Line 6**: The exported RGB colors are encoded into a single `uint64` value. This value is afterwards used for color accumulation.
- **Lines 7-9**: The goal is to add the encoded color value to each neighbouring cell. In particular, the color should be accumulated within the *averagingGridl* cell of each direct-adjacent neighbour. Thus, a double-nested `for-loop` is created which corresponds to an iteration in each of the 3-D axis within an integer range of $[-1; 1]$.
- **Line 10**: The actual dense index of the neighbouring cell is calculated from its cartesian coordinates within the *subsamplingGrid*.
- **Lines 11-15**: Due to the iteration, it might by possible to index a neighbouring cell which is not exisiting – in particular, which lies outside of the actual *subsamplingGrid*. Thus, we have to check if *cellIdx* does not under-/ or overflow the grid's dimension.
- **Lines 16-17**: The last step is to perform the actual color accumulation for the neighbouring cell. But before it is required to check if the adjacent cell is occupied by any points to avoid unnecessary accumulations. This is done by ensuring that the *countingGrid* entry for the actual cell is not zero. If it is zero, then no single point has been mapped to the cell during subsample evaluation (see Section 4.4.2). Otherwise, the encoded color value is accumulated to the *averagingGrid* entry for the actual cell.

4.4.3.1 Extension: Distance Weighting

The disadvantage of inter-cell color filtering is that while this approach has been shown to reduce aliasing artifacts (see Section 4.4.2) during rendering, it also loses fine texture detail. To counteract this, we extended the approach to include a distance weighting. In particular, point color values from adjacent cells are weighted based on their euclidean distance to the target cell before they are accumulated: colors from nearer points contribute more, colors from farther points contribute less to the color sum. As can be seen in the Figure 4.14, the weighting function is modeled on the Gaussian bell curve – featuring its maximum value 1 at 0 and converging against 0 at 1:

$$f_{weighting}(x) = e^{-\frac{x^2}{0.1}} \tag{4.12}$$

Algorithm 4.10: CUDA kernel: inter-cell color filtering. The kernel is executed for each point. The point's color value is accumulated to each neighbouring cell within the *averagingGrid*.

```
   Input   : childIdx, pointIdx, grid
   /* Get the octree node                                        */
 1 child = octree[childIdx]
   /* Get original and exported point data                       */
 2 pointDataIdx = child.dataIdx + pointIdx
 3 originalPoint = cloud[pointDataIdx]
 4 exportedPoint = outputBuffer[pointDataIdx]
   /* Calculate cartesian coordinates of actual cell in the grid */
 5 ix, iy, iz = calculateCoordinates(originalPoint)
   /* Encode colors for accumulation                             */
 6 encoded = encodeColors (exportedPoint.r, exportedPoint.g, exportedPoint.b)
   /* Iterate over all adjacent cells                            */
 7 for $ox \leftarrow -1$ to 1 by 1 do
 8     for $oy \leftarrow -1$ to 1 by 1 do
 9         for $oz \leftarrow -1$ to 1 by 1 do
               /* Calculate neighbour cell index                 */
10             cellIdx = calculateCellIdx(ix, iy, iz, ox, oy, oz)
               /* Check if neighbouring cell exists              */
11             underflow = cellIdx.x < 0 or cellIdx.y < 0 or cellIdx.z < 0
12             overflow = cellIdx.x >= grid or cellIdx.y >= grid or cellIdx.z >= grid
13             if underflow or overflow then
14                 continue
15             end
16             if countingGrid[cellIdx] != 0 then
17                 atomicAdd (averagingGrid[voxelIndex], encoded)
18             end
19         end
20     end
21 end
```

Thus, the weighted colors $C_{weighted}$ for a point at P residing a certain distance from the target cell T and considering a maximum point-to-cell distance of $maxDist$ are calculated as follows:

$$weight = f_{weighting}\left(\frac{\left|\vec{T} - \vec{P}\right|}{maxDist}\right)$$
$$C_{weighted} = C * weight \tag{4.13}$$

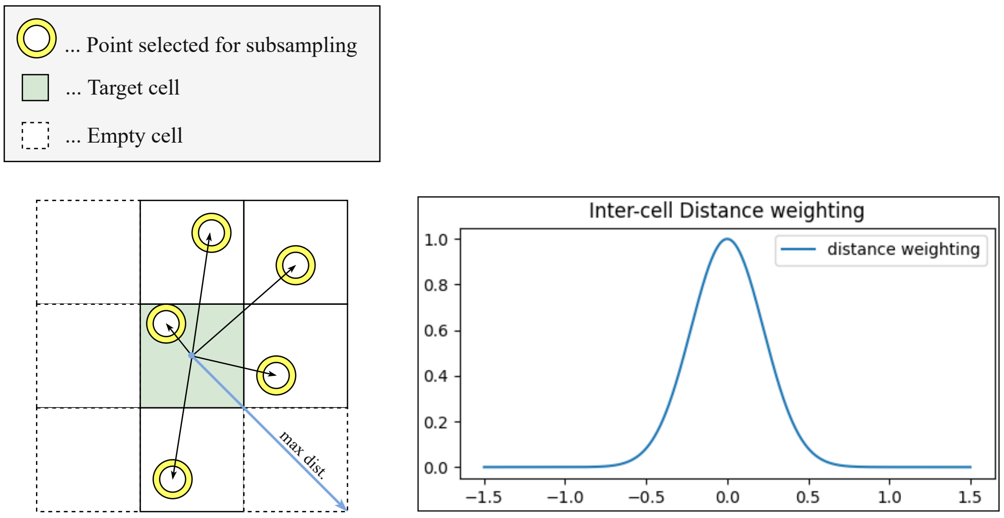


Figure 4.14: Inter-cell color filtering with distance weighting: point color values are accumulated based on their euclidean distance to the target cell.

Afterwards, the weighted colors as well as the *weight* are accumulated in *gridRGBW* (see Table 4.8) instead of the *averagingGrid*. During the actual subsampling pass, where point colors and coordinates are exported, the accumulated colors are divided by the accumulated weights to get the final distance-weighted color values.

Variable	Datatype	Size
gridRGBW	float[ ]	Amount of subsampling grid cells

Table 4.8: Relevant data structures for inter-cell color filtering with distance weighting

4.4.4 Random Point Calculation

To perform explicit random subsampling it is necessary to generate a random value per occupied *subsamplingGrid* cell. These values are used afterwards to pick a single point per cell randomly. The random value generation is done inside a CUDA kernel – *kernelGenerateRandoms* – which is executed for each *subsamplingGrid* cell and is visualized in Algorithm 4.11. The corresponding data structures are listed in Table 4.9. The idea is to generate a random value which represents a point index within the cell. For this purpose a C++ array (*randomStates*) of 1024 `curandState_t` is used, which is allocated during the Preparation phase. This variable is used for generating a single random number per thread index (max. 1024 threads per thread block). In order to generate this index value, the CUDA kernel needs to know how many points are in the actual *subsamplingGrid* cell at position *denseIdx*. For this purpose the kernel accesses *countingGrid* which holds the point counts per cell from the subsample evaluation. Referring to Line 1, the first step is to fetch the actual point amount within the *subsamplingGrid* cell. If this value (stored in *pointsInCell*) is greater than zero, then the sparse index is fetched from *denseToSparseLUT* (see Line 3). For creating the actual *random* variable, `curand_uniform` is used on Line 4 which generates a uniformly distributed value between 0 and 1. In the last step (Line 5) this *random* variable is multiplied with *pointsInCell* to generate an index variable ranging from: $(0, pointsInCell]$.

For more information about random number generation with CUDA, refer to [NVIa].

Variable	Datatype	Size
randomIndices	uint32_t[ ]	Amount of subsampling grid cells
randomStates	curandState_t[ ]	1024

Table 4.9: Relevant data structures for the random point calculation

Algorithm 4.11: *CUDA kernel: random point index calculation. The kernel is executed for each cell within subsamplingGrid and generates a random point index for each non-empty cell.*

```
  Input   : denseIdx
1 pointsInCell = countingGrid[denseIdx]
2 if pointsInCell > 0 then
3     sparseIndex = denseToSparseLUT[denseIdx]
4     random = curand_uniform(randomStates[threadIdx.x])
5     randomIndices[sparseIndex]=ceil(random * pointsInCell)
6 else
7     return
8 end
```

4.4.5 Random Subsampling and Data Distribution

The last step within our proposed subsampling approach is to actually propagate points from the child nodes to their parent node and to export these points to the *outputBuffer*. This also requires to decode the accumulated color values and to calculate the actual averaged colors. As described, PotreeConverterGpu performs random subsampling either implicitly or explicitly. The subsampling logic is implemented in a CUDA kernel – *kernelPointSubsampling* – of which there exist separate versions for implicit and explicit random subsampling. Both approaches basically follow the same algorithm and only differ in minor aspects. In fact, the only crucial difference between both approaches is the way the actual subsampling points are selected from the *subsamplingGrid* cells. In case of the implicit random subsampling approach, the undeterministic execution order of CUDA threads is exploited for selecting a random point per cell. The explicit approach on the other hand selects the points based on the generated random number per *subsamplingGrid* cell (see Section 4.4.4), combined with the implicit randomness behaviour. The complete random subsampling and point distribution procedure is visualized in Algorithm 4.12. The following Listing provides a fine grained explanation based on this Algorithm:

- **Lines 1-6**: These lines are essentially equal to the ones in $kernelSubsampleEvaluation$ (Algorithm 4.9). In fact, their purpose is to evaluate the actual parent (internal) and child node. Furthermore, the actual point in the original point cloud as well as in the $outputBuffer$ are accessed and mapped to a *subsamplingGrid* cell to retrieve the dense cell index.
- **Line 7**: The sparse index of the mapped *subsamplingGrid* cell is fetched, which has been stored in $denseToSparseLUT$ during subsample evaluation.
- **Line 8**: The subsampled points are stored continuously in the $outputBuffer$ and their corresponding indices are stored in $pointLUT$. The target position for each point ($outputDataIdx$) is calculated as follows:

$$outputDataIdx = parent.dataIdx + child.sparseIdx \tag{4.14}$$

.

- **Lines 9-13**: The goal is to select one point per *subamplingGrid* cell on a random basis. In case of explicit random subsampling we want to select the nth point in a cell (see Figure 4.15). To ensure a high point throughput and low run times we do not enumerate points which fall into *subamplingGrid* cells. Instead, we let the CUDA threads simultaneously decrement the point count value stored in *countingGrid* using `atomicSub`. Using this approach, each thread gets assigned a different descending value (stored in *old*). This value is the basis for selecting the nth point. If *old* equals to the generated random number stored in $randomIndices$ than the actual point is considered as the randomly selected one. Otherwise the CUDA thread exits.
- **Lines 15-18**: In case of implicit random subsampling the point selection has already been done during subsample evaluation. No we have to check if the actual point equals to the previously selected one. For his purpose we check if the point index is identical to the one stored in the $pointLUT$ for the actual cell. If this is not the case, the actual point is not the selected one and thus the CUDA thread exits.
- **Lines 20-22**: If explicit random subsampling is performed the point's index value is stored in $pointLUT$. This index is needed in a next subsampling pass during subsample evaluation (see section 4.4.2). In the case of implicit random subsampling this has already been done during subsample evaluation (see Section 4.4.2)
- **Lines 23-25**: The encoded color values are looked up from *averagingGrid* and are decoded (see Listing 4.3). Afterwards the point data (coordinates and colors)

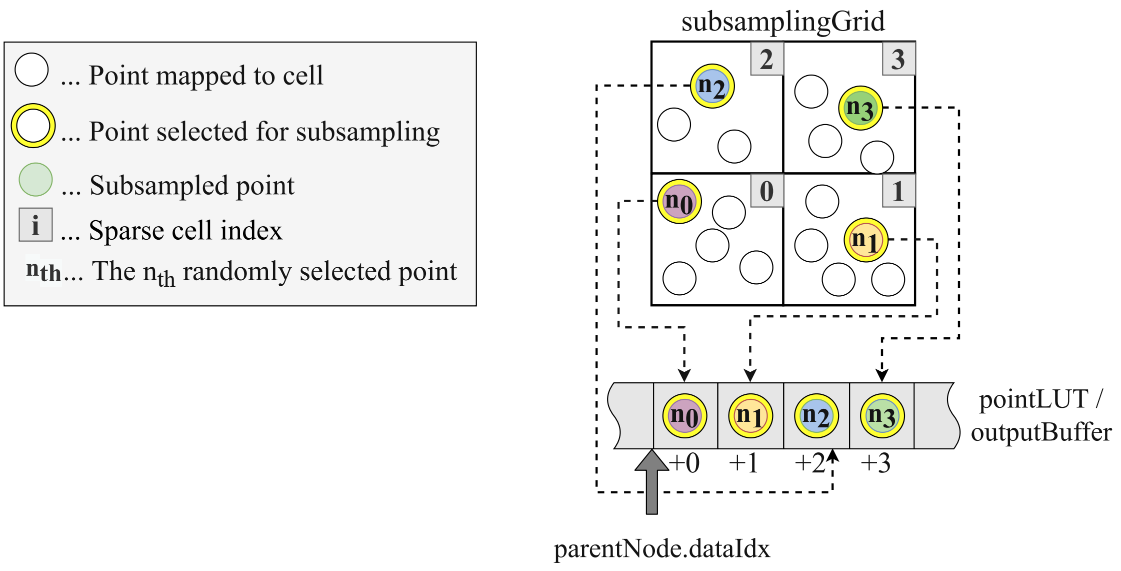


Figure 4.15: Random point subsampling: one random point n is chosen per *subsamplingGrid* cell. Based on the cells' sparse indices and their parent's *dataIdx* as offset, these points are stored in the *pointLUT* and the *outputBuffer*.

are quantized and are exported to the *outputBuffer* at the position stored in *outputDataIdx*.

- **Line 26**: Finally, the required temporary data structures are reset for the next subsampling pass.

Decoding of the packed colors works exactly the opposite way as the encoding. Each `uint64` value is decoded into three 18 Bit color values and a 10 Bit accumulator value which stores the amount of points per *subsamplingGrid* cell. Decoding is implemented in a CUDA `device` function (`decodeColors`) that is described in Listing 4.3. In addition to the color decoding, this function calculates the actual color averages by dividing the decoded color components by the amount of points in the cell. Afterwards the colors are stored in three `uint16` variables.

```
__device__ uint64_t decodeColors (const uint64_t &encoded, uint16_t
    (&decoded)[3])
{
    auto pointsInCell = static_cast<uint16_t> (encoded & 0x3FF);
    decoded[0] = ((encoded >> 46) & 0xFFFF) / pointsInCell;
    decoded[1] = ((encoded >> 28) & 0xFFFF) / pointsInCell;
    decoded[2] = ((encoded >> 10) & 0xFFFF) / pointsInCell;
}
```

Listing 4.3: CUDA device function for decoding an packed RGB color.

Algorithm 4.12: CUDA kernel: random subsampling and point distribution. The kernel is executed for each point in a node's child nodes.

```
Input    : parentIdx, childIdx, pointIdx, isExplicit
/* Get the octree node                                              */
parent = octree[parentIdx]
child = octree[childIdx]

/* Get original and exported point data                             */
pointDataIdx = child.dataIdx + pointIdx
originalPoint = cloud[pointDataIdx]
exportedPoint = outputBuffer[pointDataIdx]

/* Get dense and sparse cell index                                  */
denseIndex = mapPointToGrid(originalPoint)
sparseIndex = denseToSparseLUT[denseIndex]

/* Evaluate output position                                         */
outputDataIdx = parent.dataIdx + sparseIdx

/* Decrease counter in countingGrid                                 */
if isExplicit then
    old = atomicSub (countingGrid[denseVoxelIndex], 1)
    /* Check if the actual point is the selected point              */
    if old != randomIndices[sparseIndex] then
        return
    end
else
    if pointDataIdx != pointLUT[outputDataIdx] then
        return
    end
end

/* Store point index in pointLUT                                    */
if isExplicit then
    pointLUT[outputDataIdx] = pointDataIdx
end

/* Decode encoded color and export quantized point to outputBuffer  */
encodedColor = averagingGrid[denseVoxelIndex]
decodedColor = decodeColor(averagingGrid[denseVoxelIndex])
outputBuffer[outputDataIdx] = quantizePoint(originalPoint, decodedColor)

/* Reset temporary needed data structures                           */
reset(denseToSparseLUT, countingGrid, averagingGrid)
```

4.5 LOD Data Export

The *LOD data export* is the last step in the processing pipeline (Figure 3.3), thus deals with already converted point clouds.

As mentioned in previous sections (i.e., Table 3.1), PotreeConverterGpu produces LOD data in Potree format 2.0, as well as several optional debugging data files. The Potree format 2.0 is based on the *SCANOPY*[Sch14] data format. Data sets which are exported in this format require the following data files:

- **octree.bin** - a binary file, holding all point data.
- **hierarchy.bin** - a binary file, describing the octree hierarchy and the location of each node's point data within octree.bin
- **metadata.json** - a JSON file, containing metadata.

In Sections from 4.5.1 to 4.5.4, the structure of those files and their generation by PotreeConverterGpu are discussed in detail.

4.5.1 Generating octree.bin

PotreeConverterGpu exports the converted multi-resolution point cloud into a binary file named *octree.bin*. During subsampling, points are quantized (see Section 4.5.2) and exported to a output buffer residing on device (GPU) memory. The exported points are grouped by octree nodes, which are unsorted and arranged in an arbitrary order. To export the data to the file system, the buffer is first copied from device to host memory, and afterwards written to octree.bin without any additional processing. An auxiliary binary file – *hierarchy.bin* – is also generated (see Section 4.5.3), which describes the nodes and the location of their point data in octree.bin.

4.5.2 Quantizing Coordinates and Colors

Due to their coordinate range, georeferenced point clouds typically use double values for processing. In order to preserve high coordinate precision with just 32 bits, all exported coordinates are converted from their initial floating-point representation to fixed-point coordinates with a 32-bit integer representation [Sch16]. The quantized integer coordinates are computed as follows:

$$quantized = \frac{original - boxMin}{scale}. \tag{4.15}$$

$boxMin$ is the coordinate vector of the minimum point of the cubic bounding box. $scale$ is a factor that defines the fixed-point precision, i.e., 10^{-3} if we want to preserve 3 fractional digits (millimeter precision). The coordinate quantization targets two aspects:

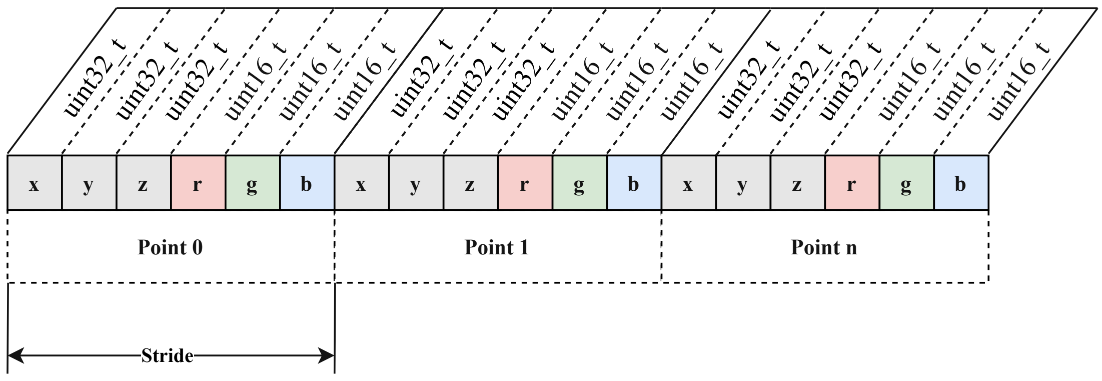


Figure 4.16: Data layout of the binary outputBuffer: its structure is very similar to the input data layout but with a few important changes

1. Point clouds are moved to the origin by subtracting the minimum of the cubic bounding box. This step is done to avoid potential integer-overflows after scaling.
2. Coordinate values are scaled by 10^{-D} to preserve D fractional digits during conversation from floating-point to integers.

Finally, the quantized coordinates are cast to `uint32_t`. Note that all coordinates are in the range of $[0, \mathit{boxSize}/\mathit{scale}]$, where *boxSize* is the side length of the cubic bounding box. Unlike coordinates, colors are stored in their original format but are cast to `uint16_t` – the default color format of the LAS and Potree file formats.

4.5.3 Generating hierarchy.bin

After octree.bin has been generated and exported, hierarchy.bin is created on host (CPU) side. As shown in Figure 4.17, for each node in octree.bin, a binary metadata entry is stored in hierarchy.bin. Each entry contains information about the byte offset, the amount of points, existing child nodes, etc.

In contrast to octree.bin, nodes in hierarchy.bin are stored in a specific order. In particular, PotreeConverterGpu traverses the octree in a breadth-first-order starting with the root node and stores each node's metadata in hierarchy.bin sequentially. The reason of adhering to this storage schemata is that it allows to store metadata in a compact way without the need for additionally saving the metadata nodes' positions. During rendering, Potree can then traverse nodes within hierarchy.bin and can access the belonging point data within octree.bin.

4.5.4 Potree Metadata

In order to render point clouds, Potree needs the data stored in octree.bin (Section 4.5.1), hierarchy.bin (Section 4.5.3) and *metadata.json*. The latter file contains general infor-

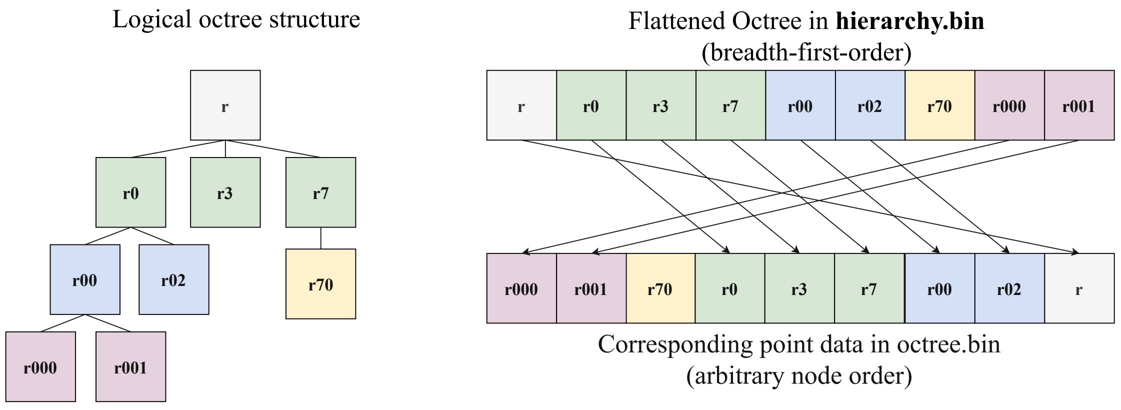

Logical octree structure
r
r0
r3
r7
r00
r02
r70
r000
r001
Flattened Octree in **hierarchy.bin**
(breadth-first-order)
r r0 r3 r7 r00 r02 r70 r000 r001
r000 r001 r70 r0 r3 r7 r00 r02 r
Corresponding point data in octree.bin
(arbitrary node order)

Figure 4.17: Logical octree structure vs. flattened, breadth-first ordered representation in hierarchy.bin. The flattened version in hierarchy.bin additionally stores the byte offset and the byte size for each node, necessary for indexing nodes within octree.bin.

mation about the exported multi-resolution point cloud and the octree data structure behind it. Some of these metadata are, for example, the Potree file format version, the global bounding box, the amount of exported points, or global flags denoting point cloud properties. Furthermore, metadata.json additionally contains a section for two point attributes, i.e., the coordinates (x, y, z) and the color (r, g, b).

In addition, PotreeConverterGpu introduces new global flag parameters to Potree [Sch16]:

- `ADDITIVE`: Is assigned when the additive LOD scheme was used during subsampling. Assumed by default, since PotreeConverter 1.x and 2.x only produce additive LODs.
- `REPLACING`: Is assigned when the replacement LOD scheme was used during subsampling.
- `INTRA-CELL-COLOR-FILTERING`: Is assigned when intra-cell color filtering was performed.
- `INTER-CELL-COLOR-AVERAGING`: Is assigned when inter-cell color filtering was performed.
- `IMPLICIT-RANDOM-SUBSAMPLING`: Is assigned when implicit random point subsampling was performed.
- `EXPLICIT-RANDOM-SUBSAMPLING`: Is assigned when explicit random point subsampling was performed.

CHAPTER 5

Results and Evaluation

PotreeConverterGpu offers several improvements over
PotreeConverter 1.x [Sch16] and PotreeConverter2.x [SOW20]. Beside color filtering and faster LOD generation, PotreeConverterGpu also provides the ability of being used as a library.

This section focuses on analysing PotreeConverterGpu in depth. In particular, we provide a qualitative LOD evaluation (Section 5.2), as well as a quantitative performance and memory analysis (Section 5.3). Both evaluations focus on the LOD generation under different subsampling configurations, including implicit vs. explicit random subsampling and intra-cell vs. inter-cell color filtering, as well as the comparison between PotreeConverterGpu and PotreeConverter 2.1. Whereas the qualitative evaluation provides a visual analysis of generated LOD results, the quantitative evaluation especially focuses on the LOD generation runtime, the consumed amount of GPU memory and the executed CUDA kernels.

5.1 Laboratory Setup

According to Table 5.1, the test system consists of a *Dell Precision 5820 Tower* equipped with a *NVIDIA TITAN RTX* GPU. All tests and evaluations have been performed on *Microsoft Windows 10*, and PotreeConverterGpu was built with *CMake* and *CUDA 11.3*. All rendering experiments were conducted using *PotreeDesktop 1.8* and *Potree 1.8*.

Hardware / Software	Description / Version
Dell Precision 5820 Tower	Intel Xeon W-2145 CPU with 8 cores Windows 10 64 GB RAM
NVIDIA TITAN RTX	4608 CUDA Cores 24.6 GB Memory
CUDA	11.4
CMake	3.21.1
MSVC	14.28.29333
Potree / PotreeDesktop	1.8

Table 5.1: Laboratory setup: deployed hardware and software components

5.2 Qualitative Result Evaluation

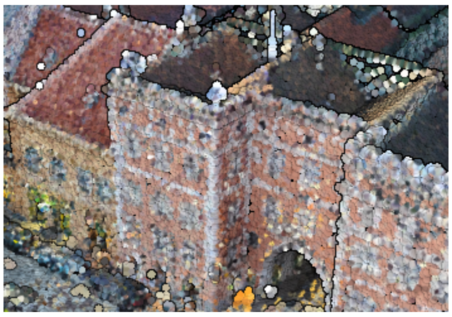
This section focuses on the visual comparison of LODs generated with PotreeConverterGpu. The differences between subsamples generated with different subsampling configurations are visually represented and evaluated. During the comparisons, the main focus will be on the ocurrence of aliasing artifacts. Furthermore, the LODs are compared with those generated with Potreeconverter2.1.

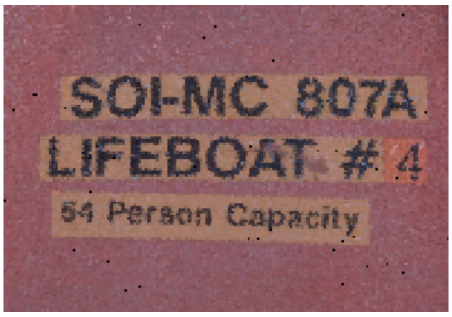
5.2.1 Point Cloud Color Filtering: a Visual Comparison

In order to visually compare the quality of different conversions, we use point-cloud data sets with high-frequent color information and fine texture details. These data sets are then converted with PotreeConverterGpu and are rendered afterwards in Potree. To achieve the best results, high-quality splatting is enabled in Potree which is complimentary to color-filtering for point clouds. Figure 5.1 shows screenshots from these renderings which were generated using explicit random point subampling with four different subsampling configurations.

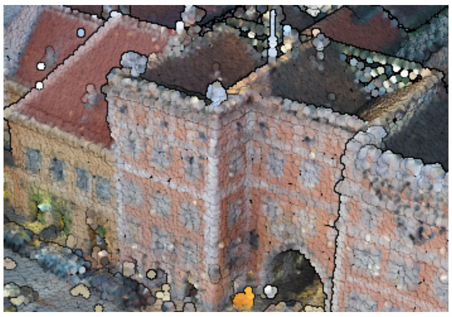
A clear quality increase can be noticed when comparing results generated without color filtering against results generated with intra-cell color filtering. While LODs without color filterings (Figures 5.1a , 5.1b) show clearly visible aliasing artefacts, the LODs generated with intra-cell color filtering feature a significant lower amount of visible artefacts (Figures 5.1c , 5.1d). Nevertheless, a certain amount of noise is still perceptible. The reason for this effect is that intra-cell color filtering only considers local color values within a single subsampling cell but completely ignores neighbouring points outside the cell.

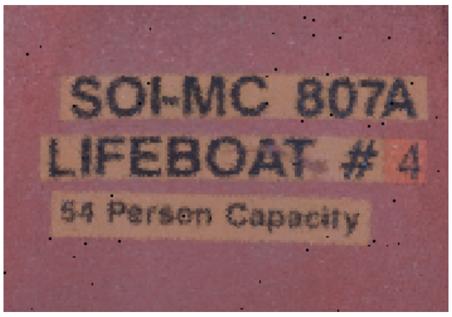
Investigating the results generated with inter-cell color filtering (Figures 5.1e , 5.1f), one can see that considering points from neighbouring subsampling grid cells further minimizes visible artifacts, genearting a noise-free and smooth representation. The drawback with this approach is that fine texture details are mainly blurred and not visible.


(a) Without color filtering

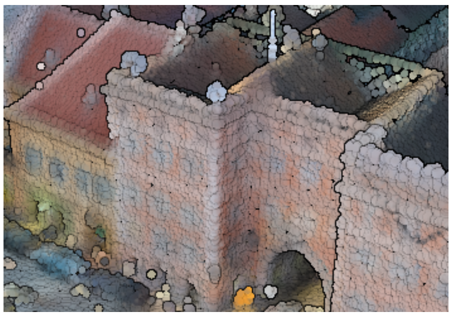


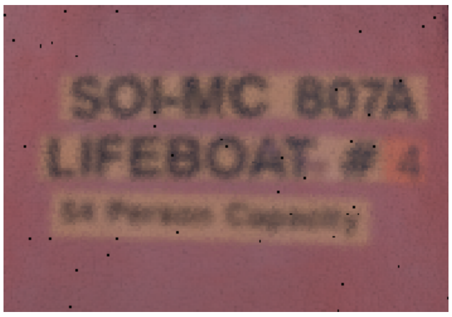
(b) Without color filtering


(c) Intra-cell filtering

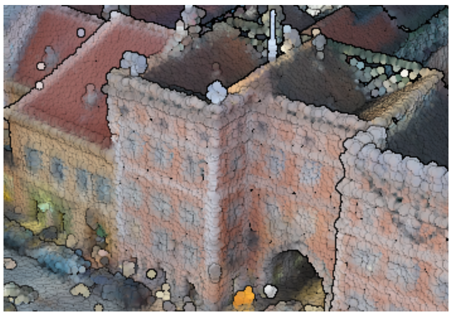


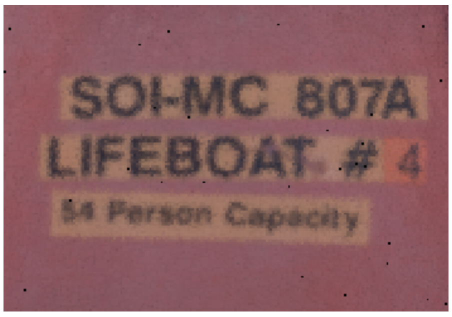
(d) Intra-cell filtering


(e) Inter-cell filtering



(f) Inter-cell filtering


(g) Inter + distance weighting



(h) Inter + distance weighting

Figure 5.1: Visual quality comparison of generated LOD data: LODs of the Retz [Sys] and Lifeboat [AG] data generated with PotreeConverterGpu using different subsampling configurations.

Figures 5.1g , 5.1h show renderings from LODs generated with inter-cell color filtering in combination with distance weighting. While noise and aliasing artifacts are equally well reduced as in pure inter-cell filtering, fine texture details are still preserved and clearly visible.

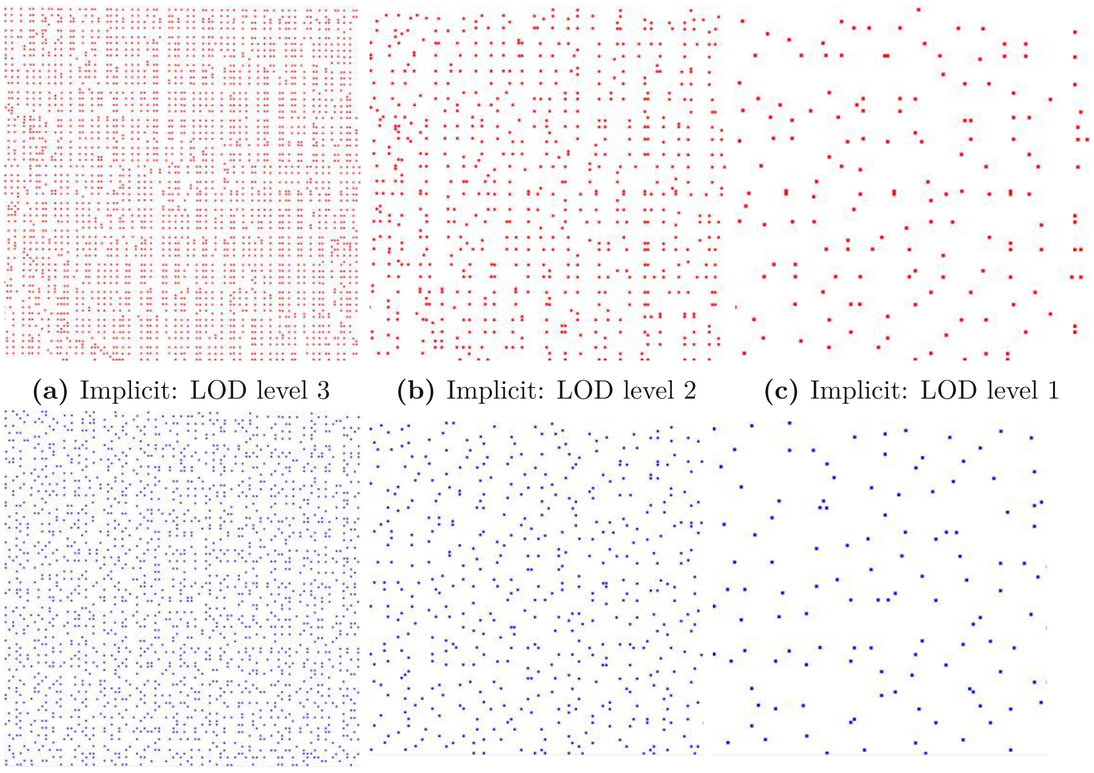
5.2.2 Implicit vs. Explicit Random Subsampling

The goal of this section is to evaluate the randomness of the point distribution within lower LODs. For this purpose, the AIT coin data set was processed with PotreeConverterGpu, once using implicit and once using explicit random point subsampling. The reason for choosing exactly this data set is that points within ICI 3D data sets are arranged in an equally sampled regular grid. This makes it easier to detect weak points regarding the random point distribution during subsampling. Figure 5.2 shows cutouts from three LODs generated from the coin data set, each processed once with implicit and once processed with explicit random subsampling. The LODs are exctracted from the generated Potree data sets using the Python tool *potree_to_ply* [1] and visualized in CloudCompare. When comparing the LODs of *level*3 (Figure 5.2a and 5.2d), one can see that the LOD generated with implicit random subsampling shows clearly visible regular point structures as opposed to the one generated with explicit random subsampling. This allows the conclusion that the non-deterministic execution order of the CUDA threads, which is used for selecting random points within the implicit approach, does not feature the expected randomicity. However, the undesired regular patterns decrease significantly with lower LODs, as one can seen in the Figures (5.2b and 5.2c) and (5.2e and 5.2f).

5.2.3 PotreeConverterGpu vs. PotreeConverter 2.1: a Visual Comparison

In order to compare the visual quality, we compare the root nodes of the octrees generated by PotreeConverterGPU and PotreeConverter 2.1. Figure 5.3 shows three different root nodes generated with PotreeConverterGpu in the top row and generated with PotreeConverter 2.1 in the bottom row. It is clearly visible that our color filtering approach implemented in PotreeConverterGpu efficiently reduces high-frequent color information, thus significantly reduces the occurrence of high-frequency noise.

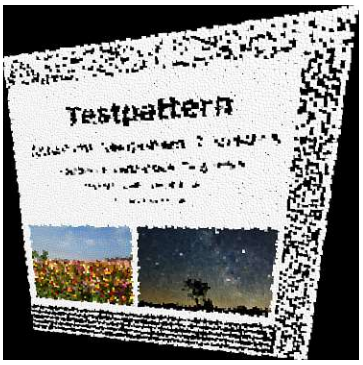
[1] *potree_to_ply* was implemented during this master thesis and generates a single PLY file for each LOD in a Potree data set. It is available on the Github repository [Kla]



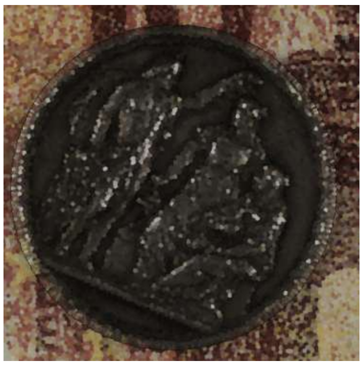
(a) Implicit: LOD level 3 **(b)** Implicit: LOD level 2 **(c)** Implicit: LOD level 1

(d) Explicit: LOD level 3 **(e)** Explicit: LOD level 2 **(f)** Explicit: LOD level 1

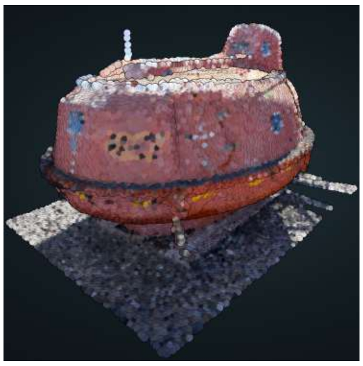
Figure 5.2: Implicit (red) vs. explicit (blue) random point subsampling: comparing LODs from three different octree levels, rendered in CloudCompare.



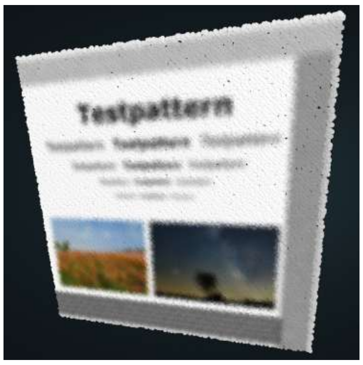
(a) PotreeConverter2.1: Testpattern data set



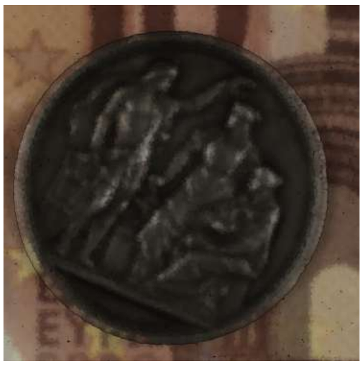
(b) PotreeConverter2.1: ICI coin data set



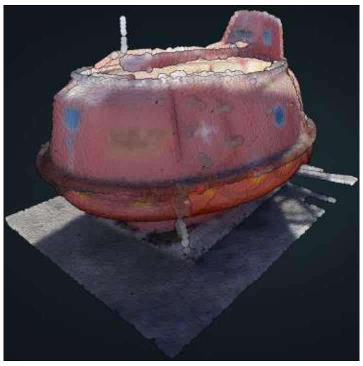
(c) PotreeConverter2.1: Lifeboat data set



(d) PotreeConverterGpu: Testpattern data set



(e) PotreeConverterGpu: ICI coin data set



(f) PotreeConverterGpu: Lifeboat data set

Figure 5.3: Visual comparison: root nodes generated with PotreeConverter2.1 vs. PotreeConverterGpu and visualized in PotreeDesktop with high-quality splats enabled. The root generated by PotreeConverterGpu were converted using explicit random point subsampling in combination with inter-cell color filtering and distance weighting.

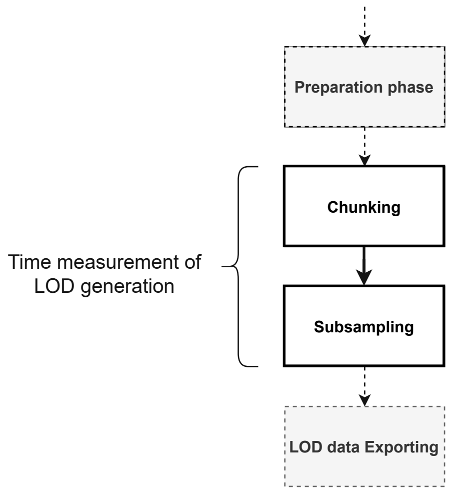


Figure 5.4: Time measurement: we focus on the raw LOD generation runtime including chunking and subsampling, but neglecting GPU memory allocation and data import/export.

5.3 Quantitative Result Evaluation

PotreeConverterGpu can be parametrized in different ways. In particular, it is possible to configure:

- The size of the chunking grid and subsampling grid
- The subsampling method: implicit or explicit random point subsampling
- Color filtering: intra-cell and inter-cell color filtering

Thus, it is interesting to evaluate how different configurations affect the GPU memory consumption and the runtime of the LOD generation. This is also crucial to find the optimal parametrization in terms of runtime and visual rendering quality.

5.3.1 Runtime Evaluation

This section focuses on the runtime of the implemented LOD generation algorithm within PotreeConverterGpu (see Figure 5.4). In particular, we want to measure the raw computation time of the multi-resolution point cloud, while neglecting the host-to-device memory transfer from the point cloud, as well as the time to export the generated LOD data. We justify this by assuming that the point clouds in targeted use cases (such as ICI) reside already in GPU memory. The experiments were conducted using a chunking grid of size 512^3, a subsampling grid of size 128^3 and an input point cloud with 119.7M points (Morrobay [2] data set). The time measurements have been conducted on the host

[2]The Morrobay data set is available on the GitHub repository [Kla]

	Implicit r. subsampling		Explicit r. subsampling	
	Runtime	Points/sec	Runtime	Points/sec
No color filtering	286.1 [ms]	418M	658.4 [ms]	182M
Intra-cell	401.5 [ms]	299M	668.7 [ms]	179M
Inter-cell	636.1 [ms]	188M	894.7 [ms]	134M
Inter-cell / dist. weighted	1596.7 [ms]	75M	1881.4 [ms]	64M

Table 5.2: PotreeConverterGpu runtime comparison: the LOD generation runtime [ms] and the point throughput [points/sec] measured for the Morro Bay data set with 119.7M points.

Kernel	Invocations	Runtime portion[%]
kernelInitRandoms	1	0.017
kernelPointCounting	1	0.998
kernelPropagatePointCounts	9	0.059
kernelInitLeafNodes	1	0.074
kernelMergeHierarchical	9	0.030
kernelDistributePoints	1	2.598
kernelEvaluateSubsamplesInter	8431	7.088
kernelInterCellFiltering	8431	66.066
kernelGenerateRandoms	8431	11.116
kernelRandomPointSubsample	8431	11.954

Table 5.3: CUDA kernel runtime distribution: the runtime portion of each CUDA kernel in relation to the overall GPU computation time. The LOD generation has been performed with inter-cell color filtering + distance weighting and with explicit random point subsampling on a point cloud with 119.7M points.

side including a `cudaDeviceSynchronize` call at the end to ensure that all remaining GPU tasks have finished.

Table 5.2 shows the LOD generation runtimes under different subsampling configurations. The table compares the runtimes without color filtering, with intra-cell and with inter-cell color filtering, as well as in combination with distance weighting. Additionally, each measurement was performed once with implicit and once with explicit random point subsampling. Beside the runtimes, the point throughput in points per second is also listed. The targeted throughput of 100M points/sec has been clearly achieved in all test cases except the one including distance weighting. Without any color filtering and using implicit random subsampling, it was even possible to achieve a point throughput of 418M points/sec. Also with explicit random subsampling in combination with inter-cell color filtering, a throughput of 134M points/sec could be achieved. Furthermore, a clear runtime difference between implicit and explicit random subsampling was measured. In fact, implicit random subsampling is approx. 60% and 40% faster than explicit random subsampling when no color filtering or intra-cell color filtering is performed and approx.

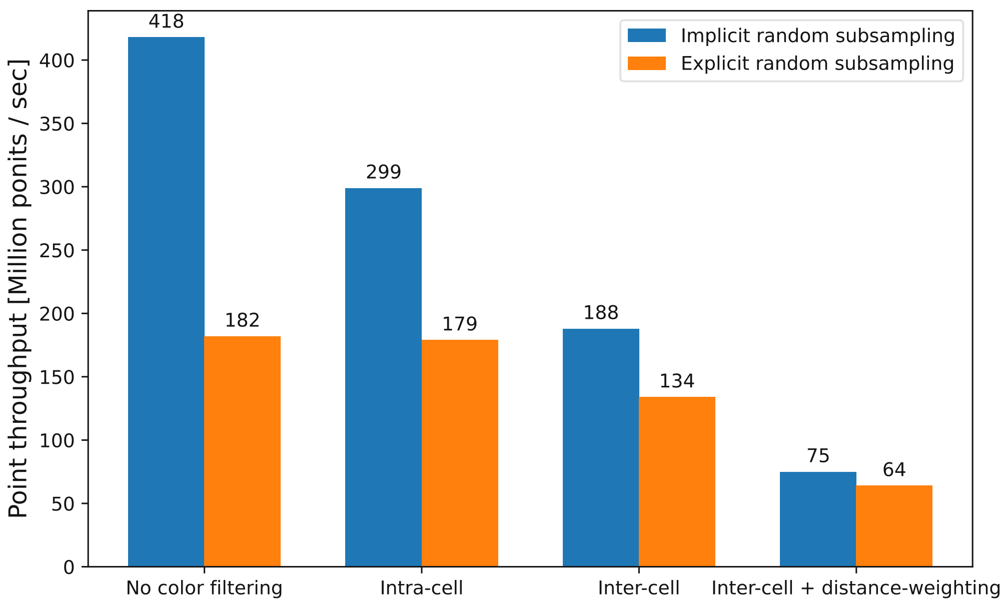


Figure 5.5: PotreeConverterGpu point throughput comparison. This diagram visualizes the point throughput measurements per second from Table 5.2.

30% and 15% faster when conducting inter-cell color filtering and with distance weighting respectively. The runtime measurements from Table 5.2 are visualized in Figure 5.5.

The time spent for generating the LOD data also includes some host ⇔ device memory copies as well as some computations on the CPU side. As their contribution to the overall LOD generation time are only minor, they are neglected. Therefore, we focus on the raw GPU computation time – in particular on the CUDA kernel runtimes. Table 5.3 provides an overview of all executed CUDA kernels together with their invocations and their accumulated runtime portion. Performing the LOD generation with explicit random point subsampling and inter-cell color filtering + distance weighting on a point cloud with 119.7M points, 33,746 CUDA kernels are executed. One can see that the contribution of chunking to the overall processing runtime is only approximately 3.8%, whereas the rest of the runtime is spent for subsampling, especially for the *kernelInterCellFiltering* (approx. 66.066%) which performs distance-weighted inter-cell color filtering (see also Figure 5.6).

5.3.1.1 PotreeConverterGpu vs. PotreeConverter 2.1

It is not possible to measure the raw processing time within PotreeConverter 2.1 as the converter is fundamentally designed for out-of-core processing. Thus, we conducted an experiment for getting a rough estimation of the performance. Within this experiment we wanted to measure the point throughput per second of PotreeConverter 2.1, but excluding the time to load the point cloud from hard disk. For this, we performed a hot-start of

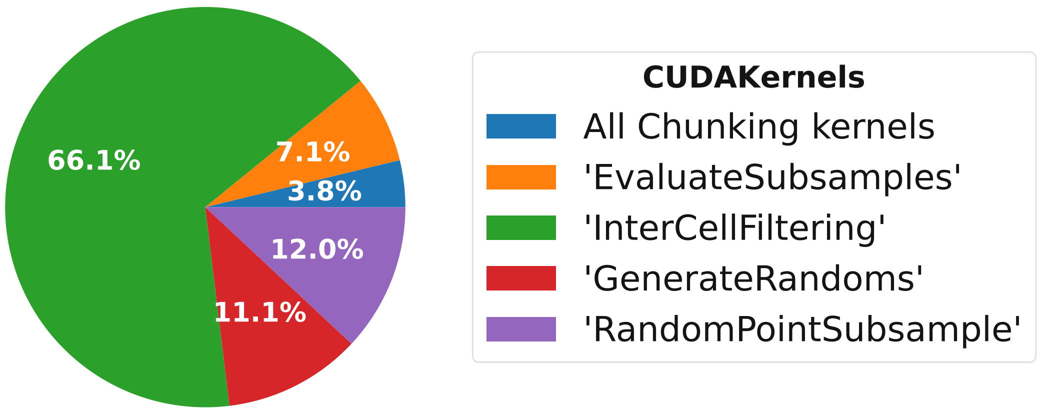


Figure 5.6: CUDA kernel runtime distribution. This diagram visualizes the CUDA kernel runtime distribution from Table 5.3. Chunking related kernels are displayed aggregated.

PotreeConverter 2.1 before evaluating the point throughput. In particular, we executed the converter twice on the same data set (Morrobay data set), but before the second execution we ensured that the point cloud was cached in RAM using the program *RAMMap* [Mic]. After the second execution, PotreeConverter 2.1 responded with a point throughput of 4.9M points per second. When comparing the point throughput measurements from PotreeConverterGpu (excluding IO operations) and PotreeConverter 2.1 (including IO operation) we achieve according to Table 5.4:

- a **minimum** performance improvement of factor **13** (performing explicit random subampling and inter-cell color filtering with distance weighting)

	Performance gain (factor)	
	Implicit r.subsampling	**Explicit r.subsampling**
No color filtering	85	37
Intra-cell	61	36
Inter-cell	38	27
Inter-cell / dist. weighted	15	13

Table 5.4: PotreeConverterGpu vs. PotreeConverter 2.1: the table shows the performance gain of PotreeConverterGpu in comparison to PotreeConverter 2.1 under several configurations.

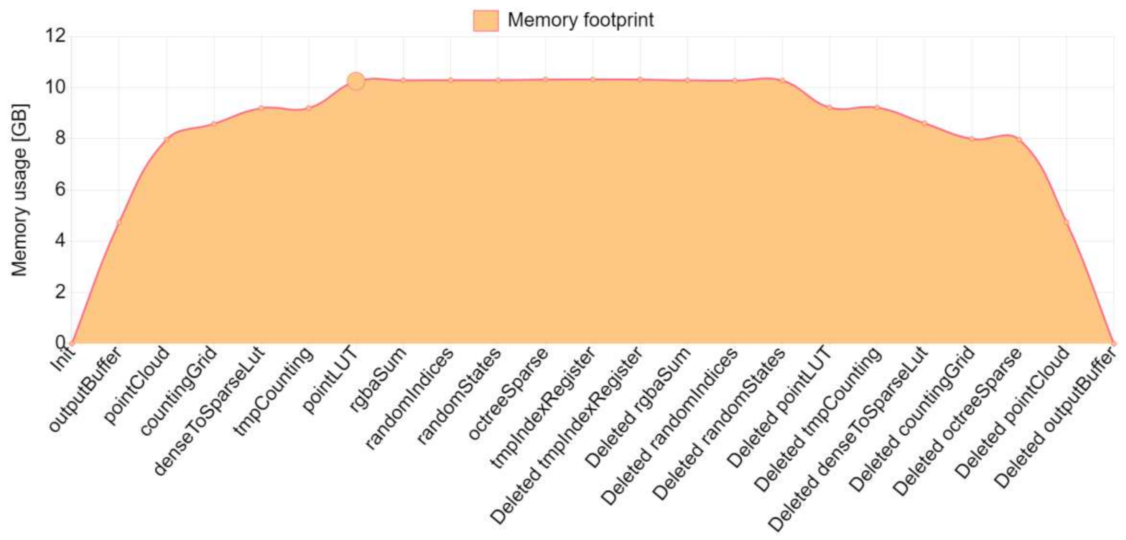


Figure 5.7: PotreeConverterGpu GPU memory consumption: the occupied GPU memory in GB over time. Each data point corresponds to the actual GPU memory occupancy when a data structure is allocated or de-allocated. The maximum occupied memory is marked with a bigger dot.

- a **maximum** performance improvement of factor **85** (performing implicit random subampling without any color filtering)

5.3.2 GPU Memory Consumption

PotreeConverterGpu performs LOD generation in-core on the GPU. Thus, the size of processable point clouds is limited by the amount of available GPU device memory. This section investigates the GPU memory consumption of PotreeConverterGpu when performing explicit random subsampling in combination with inter-cell color filtering. Processing is done for the Morrobay data set with 119.7M points and double precision coordinates (3.2GB). The *outputFactor* (see Section 4.2) is set to 2.2. Figure 5.7 visualizes the GPU memory consumption of PotreeConverterGpu over time – in particular, the memory consumption whenever a data structure is allocated or deallocated. According to the diagram, the maximum memory consumption is 10.3GB. As one can see in the diagram as well as in Table 5.5, approximately 77% of the occupied GPU memory is dedicated to the input point cloud and the output buffer. The rest of the memory (~22.8%) is used for the LOD calculation and only ~0.2% is used to store the actual sparse octree (hierarchy). One has to keep in mind that the size of the *sparseOctree* data structure heavily depends on the spatial point distribution of the input point cloud. Thus the value of 0.2% is only valid for the Morrobay data set.

To estimate the maximum size at which point clouds can still be processed on a GPU device, the GPU memory consumption is drawn as a function of the point amount in

Data structure	Memory size [GB]	Percentage
outputBuffer	4.740	45.9%
pointCloud	3.232	31.3%
pointLUT	1.053	10.2%
countingGrid	0.614	5.9%
denseToSparseLut	0.614	5.9%
octreeSparse	0.025	0.2%
rgbaSum	0.034	0.6%
randomIndices	0.008	
tmpIndexRegister	0.002	
randomStates	4.915e-05	
tmpCounting	4e-09	

Table 5.5: GPU memory occupation: data structures residing in GPU memory with their memory sizes [GB] and the percentages of the total memory occupied. The entries are arranged according to their memory usage in descending order. The values where measured while processing the Morrobay data set with 119.7M points.

Figure 5.8. The diagram shows the minimum required GPU memory for a given point amount or vice versa, the maximum processable amount of points for a GPU with a given memory. As the size for storing the sparse octree depends on the spatial distribution of the points within the point cloud, it is not considered within the diagram. The memory consumption also depends on the precision of the provided point coordinates. Thus, the diagram differentiates between single and double precision coordinates. As one can see, for an NVIDIA TITAN RTX with a memory size of 24.576 GB, it is possible to process a point cloud with a maximum size of ~309M (double precision) and ~367M (single precision) points or less. The diagram has been created considering an *outputFactor* of 2.2, a *chunkingGrid* of 512^3 and a *subsamplingGrid* of 128^3.

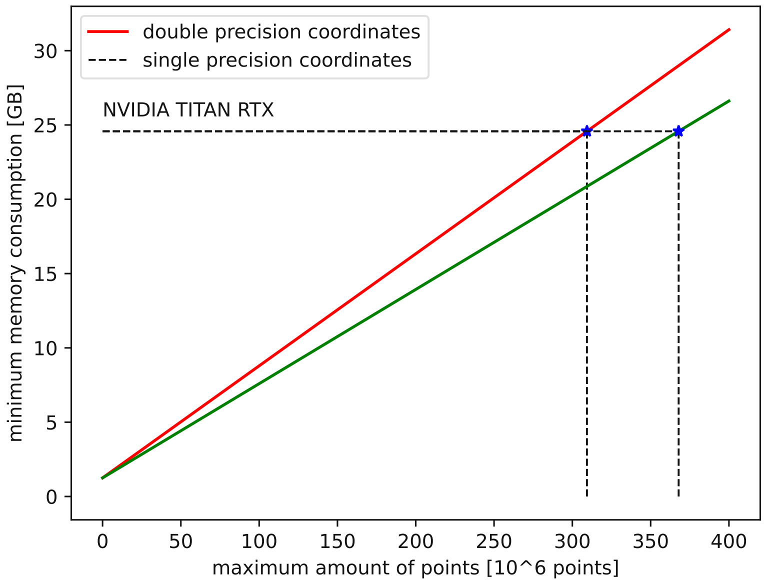


Figure 5.8: PotreeConverterGpu GPU memory estimation: the diagram shows the minimum memory consumption as a function of the point amount when generating an LOD using an *outputFactor* of 2.2, a *chunkingGrid* of 512^3 and a *subsamplingGrid* of 128^3. The diagram has been generated with the *memory_consumption.py* Python script which is available on the project GitHub repository [Kla].

CHAPTER 6

Conclusion

We present PotreeConverterGpu, an in-core GPGPU-based implementation of PotreeConverter. PotreeConverterGpu is able to generate Potree-compliant LOD data from point clouds, residing either on host or device memory. In addition to its function as a standalone executable, PotreeConverterGpu can also be included as a library into third-party software. For this purpose we additionally provide a C-API for simpler integration. To improve the runtime and point throughput compared to previous PotreeConverter versions (1.x and 2.x), we provide a GPU-optimized version of LOD generation algorithms based on [SOW20]. Furthermore, we reduce the occurrence of aliasing artifacts by providing color filtering for point clouds (similar to mip-mapping), which has been first proposed by Rusinkiewicz et al. [RL01], in combination with a replacement LOD scheme (suggested by [WBB+08] and [SOW20]). Color filtering is performed during subsampling, which uses a 3-dimensional subsampling grid to randomly extract a single point per cell from higher LOD nodes. We provide two color-filtering approaches – intra-cell, which averages point colors on a cell basis, and inter-cell with distance-based weighting, which also considers points from neighbouring cells. While both approaches significantly reduce high color frequencies, inter-cell color filtering clearly outperforms the intra-cell approach but leads to a higher runtime. Random point subsampling can be performed either implicitly (exploiting the GPU's random warp execution order) or explicitly (calculating a single random number per cell). While the former provides a better runtime behaviour, the latter one provides better randomicity during point selection, which can be perceived especially in higher LODs.

We benchmarked our implementation with a point cloud data set consisting of 119.8M points on a NVIDIA TITAN RTX and were able to achieve a point throughput ranging from 64M points/second (inter-cell color filtering with distance-based weighting and explicit random subsampling) up to 418M points/second (implicit subsampling without color filtering), which is 13 and 85 times higher than with PotreeConverter 2.1. The effect of the implemented color filtering approach is clearly perceptible, especially in lower LODs, where high-frequent color information is effectively reduced.

6.1 Limitations and Future Work

In addition to the proposed improvements especially in terms of performance and anti-aliasing, the actual PotreeConverterGpu implementation still provides room for improvement. Thus, future work includes:

1. Processing of additional point attributes in addition to color values. Currently, other attributes than color values (e.g., scalar values) are not considered during LOD generation. In particular, these additional values are discarded during processing and thus not exported to the generated multi-resolution point cloud.

2. LOD generation based on an out-of-core processing implementation. Through the in-core processing, the size of the point clouds process-able with PotreeConverterGpu is currently limited by the GPU memory. Implementing an out-of-core LOD generation would eliminate this limitation.

3. Considering an alternative point distribution strategy. In PotreeConverterGpu, points are iteratively propagated through the hierarchy until their target leaf node is found. This procedure might be more efficient using a lookup table, which would require just a single lookup for every point to determine the target leaf node. This strategy was proposed by Schütz et al. in [SOW20].

4. Propagating the amount of contributing points during color accumulation. To save GPU memory, subsampled points are directly exported and their color accumulation data structures are reset after each subsampling pass. This implies that the point amounts per cell are discarded when subsampled points are exported. When these points contribute to another subsample later on, their exported and averaged color values are used for accumulation and their amount of contributing points is considered to be 1. This may lead to an imprecision in further color filtering calculations.

5. A random sample heuristic which could give similar good results as explicit random sampling, but has better random behaviour than implicit random subsampling.

6. Processing of more than one point per CUDA thread. In the current implementation – independent from the processing step – each point is processed by a single CUDA thread. To increase performance as well as scalability of the application it might be more appropriate to assign and process multiple points per CUDA thread.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AAB+83] Edward Adelson, Charles Anderson, James Bergen, Peter Burt, and Joan Ogden. Pyramid methods in image processing. *RCA Eng.*, 29, 11 1983.

[AG] Weiss AG. `https://weiss-ag.com/de/`. (Accessed: 2023-02-23).

[ASP18] Doris Antensteiner, Svorad Stolc, and Thomas Pock. Variational fusion of light field and photometric stereo for precise 3d sensing within a multi-line scan framework. *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 1036–1042, 2018.

[ASV+17] Doris Antensteiner, Svorad Stolc, Kristián Valentín, Bernhard Blaschitz, Reinhold Huber-Mörk, and Thomas Pock. High-precision 3d sensing with hybrid light field & photometric stereo approach in multi-line scan framework. In *Electronic Imaging*, volume 2017, pages 52–60, 01 2017.

[Aut] Autodesk. 3DS MAX. `https://www.autodesk.de/products/3ds-max`. (Accessed: 2023-02-11).

[BSA18] Bernhard Blaschitz, Svorad Stolc, and Doris Antensteiner. Geometric calibration and image rectification of a multi-line scan camera for accurate 3d reconstruction. *Electronic Imaging*, 2018:1–6, 01 2018.

[BSHZK05] M. Botsch, Alexander Sorkine-Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's gpus. In *Point-Based Graphics, 2005 - Eurographics/IEEE VGTC Symposium Proceedings*, pages 17– 141, 07 2005.

[CG] Inc. 2021 Cesium GS. Cesium. `https://github.com/CesiumGS/cesium`. (Accessed: 2023-02-11).

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[Con] W3C World Wide Web Consortium. WebGPU W3C Working Draft, February 2023. `https://www.w3.org/TR/webgpu/`. (Accessed: 2023-02-11).

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Trans. Graph.*, 22(3):657–662, July 2003.

[FAR] FARO. Scene Web Share. https://www.faro.com/en/Products/Software/WebShare. (Accessed: 2023-02-11).

[Fou] Blender Foundation. Blender. https://www.blender.org/. (Accessed: 2023-02-11).

[fPS] ASPRS American Society for Photogrammetry and Remote Sensing. LAS Specification Version 1.4-r13. https://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf. (Accessed: 2023-02-11).

[Gam] Epic Games. Unreal Engine. https://www.unrealengine.com/en-US/. (Accessed: 2023-02-11).

[GBT22] Laurin Ginner, Simon Breuss, and Lukas Traxler. Fast inline microscopic computational imaging. *Sensors*, 22:7038, 09 2022.

[Geo] Leica Geosystems. Leica Tru View. https://leica-geosystems.com/products/laser-scanners/software/leica-truview. (Accessed: 2023-02-11).

[GM] Girardeau-Montaut. CloudCompare. https://www.danielgm.net/cc/. (Accessed: 2023-02-11).

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

[Groa] KHRONOS Group. OpenCL. https://www.khronos.org/opencl/. (Accessed: 2023-02-11).

[Grob] Khronos Group. WebGL 2.0 specification. https://www.khronos.org/registry/webgl/specs/latest/2.0/. (Accessed: 2023-02-11).

[HCZ16] Liang Hu, Xilong Che, and Si-Qing Zheng. A closer look at gpgpu. *ACM Comput. Surv.*, 48(4), 2016.

[Insa] ESRI Environmental Systems Research Institute. Arcgis geographic information system. https://www.arcgis.com/index.html. (Accessed: 2023-02-11).

[Insb] Ludwig Boltzmann Institute. Heidentor data set. https://archpro.lbg.ac.at/. (Accessed: 2023-02-11).

[Int] Intel. Intel Core i9-10980XE Extreme Edition Prozessor. https://www.intel.com/content/www/us/en/homepage.html. (Accessed: 2023-02-11).

[KJWX19] Lai Kang, Jie Jiang, Yingmei Wei, and Yuxiang Xie. Efficient randomized hierarchy construction for interactive visualization of large scale point clouds. In *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*, pages 593–597, 06 2019.

[Kla] Philip Klaus. PotreeConverterGPU GitHub repository. `https://github.com/PhilipKlaus/octree-cuda`. (Accessed: 2023-02-11).

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 31–42, New York, NY, USA, 1996. Association for Computing Machinery.

[Mic] Microsoft. RAMMap v1.60. `https://docs.microsoft.com/en-us/sysinternals/downloads/rammap`. (Accessed: 2023-02-11).

[NVF] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. CUDA, release: 12.0. `https://developer.nvidia.com/cuda-toolkit`. (Accessed: 2023-02-11).

[NVIa] NVIDIA. CUDA-C-Programming-Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide`. (Accessed: 2023-02-11).

[NVIb] NVIDIA. GEFORCE RTX 4090. `https://www.nvidia.com/de-de/geforce/graphics-cards/40-series/rtx-4090/`. (Accessed: 2023-02-11).

[RL01] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. *Proceedings of SIGGRAPH*, 2000, 10 2001.

[Sch] Markus Schütz. PotreeDesktop 1.8.0. `https://github.com/potree/PotreeDesktop`. (Accessed: 2023-02-11).

[Sch14] Claus Scheiblauer. *Interactions with Gigantic Point Clouds*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, 2014.

[Sch16] Markus Schütz. Potree: Rendering large point clouds in web browsers. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, sep 2016.

[Ske] Sketchfab. `https://sketchfab.com/`. (Accessed: 2023-02-11).

[SOW20] Markus Schütz, Stefan Ohrhallinger, and Michael Wimmer. Fast out-of-core octree generation for massive point clouds. *Computer Graphics Forum*, 39(7):1–13, nov 2020.

[SP11] Claus Scheiblauer and Michael Pregesbauer. Consolidated visualization of enormous 3d scan point clouds with scanopy. In *Proceedings of the 16th International Conference on Cultural Heritage and New Technologies*, pages 242–247, nov 2011.

[Sur] USGS United States Geological Survey. Entwine. `https://usgs.entwine.io/`. (Accessed: 2023-02-11).

[Sys] Riegl Laser Measurement Systems. `http://www.riegl.com/`. (Accessed: 2023-02-23).

[Sze11] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag, 2011.

[Tec] Unity Technologies. Unity. `https://unity.com/`. (Accessed: 2023-02-11).

[TGBB21] Lukas Traxler, Laurin Ginner, Simon Breuss, and Bernhard Blaschitz. Experimental comparison of optical inline 3d measurement and inspection systems. *IEEE Access*, PP:1–1, 04 2021.

[Ver] Butler Verma. Plasiso. `https://github.com/verma/plasio`. (Accessed: 2023-02-11).

[WBB+08] Michael Wand, Alexander Berner, Martin Bokeloh, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier, Dirk Staneker, and Andreas Schilling. Interactive editing of large point clouds. *Chen, Baoquan; Zwicker, Matthias; Botsch, Mario; Pajarola, Renato: Symposium on Point-Based Graphics 2007 : Eurographics / IEEE VGTC Symposium Proceedings, Eurogrpahics Association, 37-46 (2007)*, 08 2008.

[Woo92] Robert Woodham. Photometric method for determining surface orientation from multiple images. *Optical Engineering*, 19, 01 1992.

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

[YGW+15] Dong-Ming Yan, Jianwei Guo, Bin Wang, Xiaopeng Zhang, and Peter Wonka. A survey of blue-noise sampling and its applications. *Journal of Computer Science and Technology*, 30:439–452, 05 2015.

[ZPBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen Baar, and Markus Gross. Surface splatting. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 2001, 08 2001.