FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Integrating Annotations into a Point-based Rendering System

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Markus Tragust
Matrikelnummer 0827047

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Claus Scheiblauer

Wien, 19.11.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Integrating Annotations into a Point-based Rendering System

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Markus Tragust
Registration Number 0827047

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Claus Scheiblauer

Vienna, 19.11.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Markus Tragust
Malfattigasse 18/32-33, 1120 Wien

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

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Many thanks to Claus Scheiblauer who provided valuable tips and comments during the whole creation process of this thesis.

I also would like to thank Michael Wimmer for his helpful comments and input during the writing of this thesis.

Thanks to the FWF funded START project “The Domitilla Catacomb in Rome. Archaeology, Architecture and Art History of a Late Roman Cemetery” for providing the point model of the Domitilla catacomb. That project is possible by commission and with the help of the Pontificia Commissione di Archeologia Sacra/Roma.

Finally, I want to thank my parents Günther and Sabina but also my better half Tamara for their support during my studies and the writing of this thesis.

Abstract

The preservation of archaeological sites is an important task in cultural heritage. Classical methods conserve archaeological objects in museums and provide restoration of archaeological sites threatened by decay. The improved digitalization provides the possibility to generate an accurate representation of archaeological sites by using laser scanners. The resulting point clouds can preserve the archaeological site and provide the possibility to view it in its digital form even if it no longer exists.

Usually, the archaeological site comes with a lot of different material, which has been created over the years. This material provides information about the digitalized object, which helps to gain a deeper understanding about the presented archaeological site.

This thesis presents an annotation system for a point-cloud renderer. The system allows adding annotations in the 3D space next to the part of the point cloud it belongs to. This helps to provide the additional information of the point cloud in the context it belongs to. Moreover, each annotation should present interesting information about specific annotated parts of the archaeological site to the viewer. Besides simple textual annotations, a variable amount of documents, such as images and PDFs, can be attached to each annotation to provide all kind of information.

Several filtering techniques, including viewpoint-dependent priority filtering, are presented to control the visibility of the annotations. Moreover, a guidance system based on graphs is introduced to lead viewers to different points of interest, which are represented as annotations.

To provide a clear connection between annotations and the annotated part of the point cloud, a point-selection method and a point-marking method are presented. To allow the connection of a large set of annotations to a single point cloud, these methods are developed in CUDA. This is done by extending existing methods, which create octrees in CUDA. The developed methods allow fast execution on the GPU while a CPU-based method is not able to handle such a large amount of point selections in real-time.

Kurzfassung

Die Konservierung von archäologischen Stätten ist ein wichtiger Schritt um das Kulturerbe zu bewahren. Klassische Methoden stellen wertvolle Ausstellungstücke in Museen aus oder betreiben aufwendige Restaurierungsarbeiten von Zerfall betroffener Exponate. Die fortgeschrittenen Digitalisierungsverfahren der Informatik bieten heute die Möglichkeit eine exakte Abbildung eines beliebigen Objektes mittels Laserscanner zu erstellen. Die daraus resultierende Punktwolke ermöglicht es ein Objekt lange über sein natürliches Bestehen hinaus für die Nachwelt zu konservieren.

Für gewöhnlich existieren neben der archäologischen Stätte selbst eine Vielzahl an Daten, welche über die Jahre gesammelt wurden. Diese Daten beinhalten Informationen über das digitalisierte Objekt, sodass dies besser verstanden werden kann.

Diese Diplomarbeit präsentiert ein Annotationssystem, welches in einen Punktwolkenrenderer integriert wird. Das System ermöglicht es Annotationen so im virtuellen Raum zu positionieren, dass diese in direktem Bezug zu dem Teil der Punktwolke stehen für den sie zusätzliche Informationen bereitstellen möchten. Die so bereitgestellten Informationen sollen dem Betrachter die Möglichkeit bieten mehr über einzelne Teile des Objektes zu erfahren. Dabei sollen nicht nur Texte als Informationsquelle dienen, sondern auch Bilder, Dokumente und Webseiten können eingebunden werden. Um die Sichtbarkeit einzelner Annotationen steuern zu können, werden verschiedene Filtermöglichkeiten angeboten. Unter anderem soll es möglich sein Annotationen blickpunktabhängig anhand ihrer Wichtigkeit darzustellen. Zudem wird ein Leitsystem vorgestellt, das es ermöglicht den Betrachter zu den verschiedenen Annotationen innerhalb der Punktwolke zu führen.

Um eine klare Verbindung zwischen Annotationen und den Teilen der Punktwolke herzustellen welche annotiert werden, wird ein Selektionsverfahren und ein Markierungsverfahren für Punkte präsentiert. Die Verfahren heben die für die Annotationen relevanten Punkte heraus. Damit diese mit einer großen Anzahl von Annotationen funktionieren werden die Methoden in CUDA entwickelt. Die entwickelten Methoden erweitern existierende Verfahren, welche Octrees für CUDA bereitstellen. Dadurch ist es möglich beide Verfahren schnell auf der Grafikkarte auszuführen, während es auf dem Hauptprozessor nicht möglich wäre diese Verfahren für eine große Anzahl an Annotationen in Echtzeit zu berechnen.

Contents

CHAPTER 1

Introduction

The digitalization of parts of the real word to make it processable in software is a common task nowadays. Such parts can be small objects such as screws and go up to entire countries or even the whole world. A first example what can be done with the data once it is collected can be seen in “Google Earth”, which combines image data and height information with a model of the world. As a first step to get a three-dimensional digital representation of the desired object, its geometry has to be recorded and converted into a computer-readable form.

There exist several techniques differing in both accuracy and cost to fulfill this task. Photogrammetry can be seen as first attempt to bring real-word positional information into digital form. It uses different image-based approaches to calculate positional data from 2D images. The usage of laser scanners allows the measurement of the complete surrounding by emitting a laser beam and recoding the reflected light. Such a laser scanner can be mounted on a tripod to collect the positional data of objects or buildings. This type of scanning is referred to as terrestrial laser scanning (TLS). To scan large areas of land, typically starting at one square kilometer, a laser scanner can be mounted on vehicles or airplanes. Another cheap method to gain positional data is the usage of sensors such as they are integrated into consumer electronic devices like the „Kinect“. But they are limited to small objects and provide only limited accuracy compared to the other presented methods. All of these methods provide a point cloud which is processable and displayable within a computer. An example of a point cloud is presented in Figure 1.1

The benefits of the digitalization can be found in the field of cultural heritage, where they allow a contactless and therefore non-destructive recording of protected monuments. This provides the possibility to preserve its actual condition and allow people to visit the structure at least in a digital form if it is threatened by decay or if there are other restrictions which prohibit access. The creation and visualization of point clouds is only a first step. The point data allows detailed distance- and area measurements, but also deformation analysis.

Figure 1.1: A sample point cloud of the St. Stephen's cathedral in Vienna.

1.1 Motivation

Working on large structures as they appear in cultural heritage or huge industrial complexes, three-dimensional digital representations such as point clouds can help to give an overview of the represented facilities. But often there exists more than only the point information of the structure, which usually consists of positional data and optional per-point data such as color, normal and intensity of the reflected laser light. In cultural heritage, there exists data regarding the historical background of certain parts of the building, information about its creational process or even information located underneath the currently visible surface of the walls.

All this additional information, available as textual description, images or illustrations, is strongly correlated to different positions in the overall object. Therefore a solution which is able to display the additional information next to the location it belongs to would be beneficial for both archaeologists and visitors. First, the integration of this data into point clouds supports the archaeologist in the following ways: Having all the available data of an object in the same application makes it easier to organize the data. Moreover, it helps to see the additional information in the context it appears to the object. Visitors get the possibility to learn more about the archaeological site while traversing the point cloud, using the annotations as source of information. Besides cultural heritage, for example in the petroleum industry, exist large facilities. Detailed information about significant parts of a facility is available. This helps understand the usage of the parts and how they might influence other sections in the same facility.

1.2 Problem statement

Such annotations for point clouds are currently not available. Similar systems exist in three-dimensional space only for meshes. They lack a number of functionality. Current annotation system do not provide a detailed preview of the annotation in 3D space before opening it. A solution to filter between important and less important annotations is currently only available by defining groups which can be set visible or invisible [Díaz et al., 2011]. A clear connection between the annotation and the annotated part is only provided using boxes to surround the annotated parts. To provide an exact link between the annotated part of the object and the annotation, some kind of marking would be required.

The system proposed by Yu et al. [Yu et al., 2011] works only for meshes, it highlights selected parts of the mesh to visualize a link between parts of the mesh and their corresponding annotations. Mapping this concept to point clouds requires a point-selection method. Current laser scanners are able to produce one million points per second. A final point cloud, consisting of separate (up to several 1000) single scanned point clouds, can hold more than 10^9 points. This makes it necessary to develop a fast point-selection method. Existing point-selection solutions are not fast enough to be used for a large number of annotations in a point cloud.

1.3 Aim of work

The aim of this thesis is to provide a solution which brings the available non-positional data into relation with the part of the object it belongs to. Using this information as annotations should provide a deeper understanding of the whole object represented as point cloud. To avoid confusion when a large amount of annotations are present in the point cloud, a priority-dependent visualization of annotations should be used. Additionally, a guidance system should be provided which allows finding points of interest represented as annotations inside the point cloud. This should help to navigate in complex point clouds such as the Domitilla catacomb presented in Figure 1.2. The solution should be integrated into an existing point-cloud renderer.

To link the annotations to the part of the point clouds they belong to, a solution is required to visualize the connections. Since the underlying model of the real-world object is always a point cloud, a point-selection method has been chosen to create an accurate mapping between the annotations and their belonging points. In the existing point-cloud renderer, there exists a CPU-based point-selection approach which works with more than 10^9 points. However, it is not able to work with more than 10 selections at an interactive frame rate. The final system should be able to hold much more than 10 annotations in a single point cloud. Therefore, a method in CUDA should be created which allows creating and marking more than 10 point selections in real time.

1.4 Contribution

This thesis has three main contributions:

1. The development of the first known system which adds annotations holding textual and image information as annotations inside large point clouds, i.e., with typically up to several billion points. This system is combined with a solution to visualize a clear connection between the points of a point cloud and the annotations. This is done by extending the solution for meshes proposed by Yu et al. [Yu et al., 2011] to point clouds using point selections.

2. A point-selection method completely working on the GPU and implemented in CUDA, due to the strict requirements on performance regarding the point-selection method. This method consists of selecting and marking single points from the point cloud. The required data structures will be completely represented on the GPU using CUDA-based octrees proposed by [Karras, 2012]. This method creates large octrees when used with point clouds. For this, a compression algorithm on the used data structure for a faster per-frame test of newly visible points of the point cloud against all existing selections is created.

3. The comparison of the GPU-based point-selection method with an already existing CPU-based method proposed by Scheiblauer et al. [Scheiblauer and Wimmer, 2011].

1.5 Structure of the work

This thesis continues with Chapter 2, explaining some basic concepts regarding required data structures and algorithms needed to implement a point-selection method which is able to fully run on the GPU using CUDA.

Chapter 3 gives an introduction into the currently available annotation solutions before reflecting about state-of-the-art annotations in the domain of cultural heritage. The next sections of this chapter explain how current methods on point selection for point clouds work and introduce previous work which is important to implement a CUDA-based point-selection and marking solution.

Chapter 4 gives a detailed explanation on how an annotation-based system for point clouds can be integrated into the existing point-cloud renderer Scanopy.

Chapter 5 covers all the details that are necessary to implement a method which connects points of a point cloud with the corresponding annotations.

In Chapter 6, the implemented solution will be compared between different generations of CUDA-capable devices. The performance impact of the developed solution regarding the different compute capabilities will be analyzed. Moreover, the CUDA solution will be compared with the already existing CPU-based point selection.

This thesis will be finished with Chapter 7, reflecting the outcome and discussing potential future work.

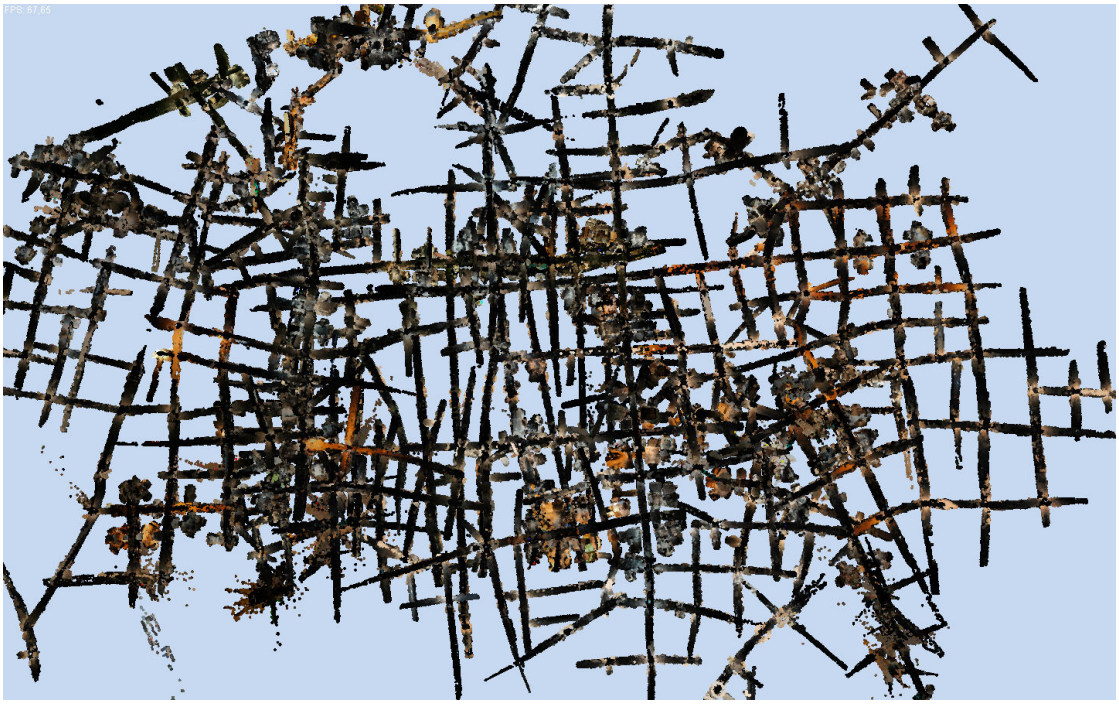


Figure 1.2: An overview of the complex structure of the Domitilla catacomb. Annotations combined with a guidance system could help to navigate through it.

CHAPTER 2

Basic Concepts

Usually, each annotation provides information for a specific part of the point cloud. This requires to visualize a link between the annotation and this specific part of the point cloud. Therefore, a highlighting method is required. To provide the possibility to highlight specific points, a point-selection method is provided to select all points that belong to a given annotation. In the first part of this Chapter (Sections 2.1 to 2.3), all the data structures and algorithms required to implement a point selection with CUDA are explained. The most significant part is the used octree structure. Not only is it already used in the CPU-based point selection, but it is also one key part for rendering point clouds in Scanopy as will be discussed in Chapter 3 Section 3.2. The second part of the Chapter (Section 2.4) will introduce the different general purpose computing methodologies. Since the chosen methodology is CUDA, a detailed description of this architecture follows.

2.1 Spatial Data Structures

Spatial data structures subdivide the space the data lies in. As this thesis covers topics in computer graphics, 2D and 3D data is of special interest here. Besides the pure representation of the data, they make certain tasks easier or faster to compute. Examples are collision detection and visibility checks. Searching for neighbors of a given point can be done in $O(log(n))$ time using k-d trees [Bentley, 1975].

Octrees help to improve the performance on intersection tests when using ray tracing. They can be used to speed up radiosity [Samet and Webber, 1988b] and are used for fast volume visualization [Boada et al., 2011]. Octrees are used for the visualization of point clouds because they help introducing a level-of-detail algorithm as will be explained in Section 3.2.

2.1.1 Octrees

An octree is a hierarchical data structure to partition the space. It has been proposed independently by Gregory M. Hunter [Hunter, 1978] in 1978 and Donald Meagher [Meagher, 1982] in

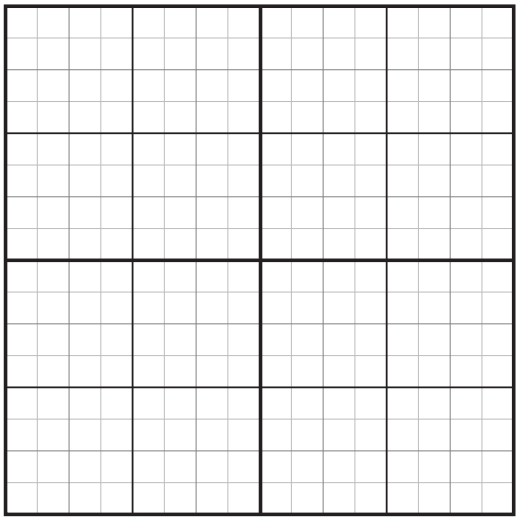


Figure 2.1: An example of a regularly subdivided quadtree with depth 4.

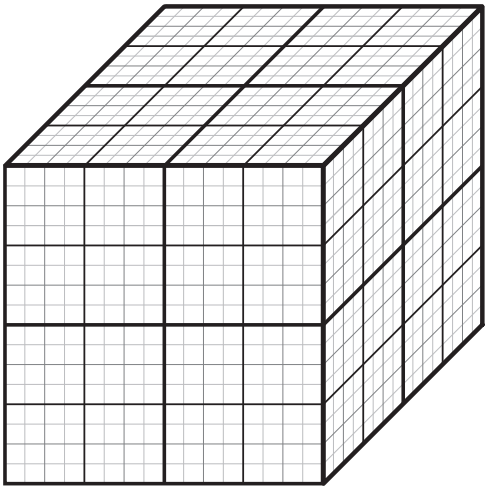


Figure 2.2: An example of a regularly subdivided octree with depth 4.

1981.Octrees extend the concept of quadtrees, which work in 2D, to 3D, and both can be seen as binary trees extended to 2 respectively 3 dimensions.

Quadtrees have a square as root node element. This square is divided into 4 non-overlapping squares of equal size called quadrants. It can handle both raster data, as it is used to represent images, and vector data used to represent geometry. When working with raster data, the data is divided until all quadrants hold only one single value. Those nodes will be referred to as leafs. The same holds for the usage of vector data, where it is divided until no two vectors share the same quadrant. [Samet and Webber, 1988a]. An example of a quadtree can be seen in Figure 2.1.

When extending the quadtree into the third dimension, the root node is a cube with a center and an expansion in x, y, z direction. Its space can be subdivided into eight cubes of equal size called octants. They all share one common corner, namely the center of the parent. This approach of subdividing can be continued recursively for each cube individually until the desired subdivision level is reached. Although in theory the refinement of the octree can be continued to any level, in practice a lower bound has to be defined. One criterion to stop the splitting step can be the amount of data required in one cell [Samet and Webber, 1988a]. This is important for the point-cloud renderer as can be seen in Section 3.2. Figure 2.2 shows an octree with depth 4.

Instead of splitting each octree node at its center, it can be split at inserted points [Samet, 1990]. Inserting a new point into the octree therefore requires the following steps: First the correct octant in the octree is searched where the point can be inserted. This is the first octant found in the hierarchy which does not hold a point already and the position of the point lies within the volume of the octant. The point will be inserted in this octant and the octant will be split into eight octants at the position of the inserted point. This can lead to octants with

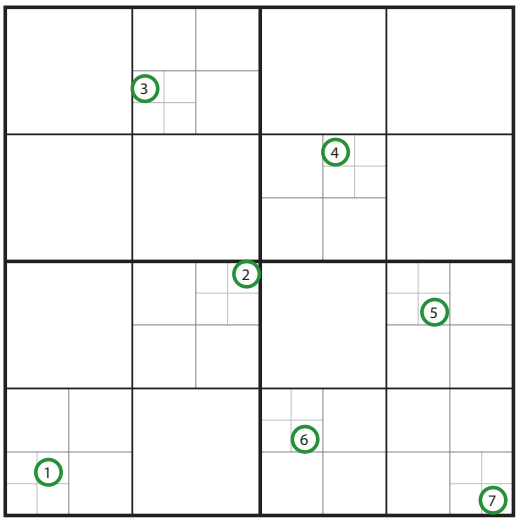


Figure 2.3: An example of a regularly subdevided quadtree with 7 inserted points. Note how the quadtree is only refined where it is necessary because it holds points at those positions.

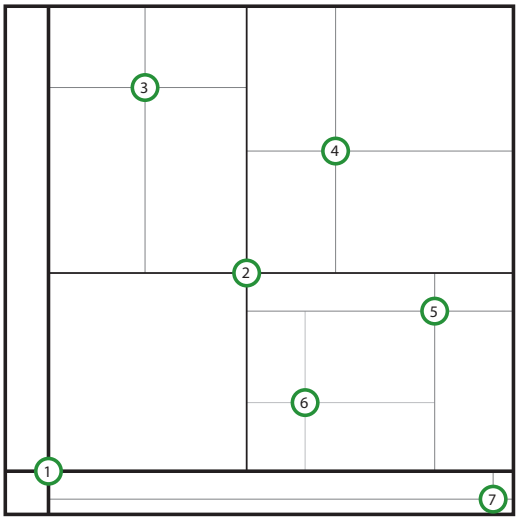


Figure 2.4: A quadtree having its nodes split at the positions of the inserted points. Compared to the regularly subdivided quadtree in Figure 2.3 on the left, the number of subdivisions is reduced while the space is subdivided irregularly.

different volumes but can provide a better distribution of the points over the nodes of the octree. Figure 2.4 shows one possible outcome of such a splitting procedure. For a better visibility of the splitting positions and the resulting cells a quadtree is used, the concept holds however for the third dimension and octrees as well. The insertion procedure can be immediately compared with Figure 2.3 which holds exactly the same points but it is regularly subdivided. The k-d tree in Section 2.1.2 is a similar solution providing a tree which holds less children per node. The number of children per node is denoted as branching factor [Samet, 1990].

Octrees allow fast searching in space. Its search complexity depends on the depth of the octree. The test whether a point lies in a filled or empty region of an octree can be done in $O(log(n))$ time if the octree is balanced along the octants. Another important factor why octrees are used in this thesis is that two octrees can easily be combined or intersected [Meagher, 1982]. Octrees can be used to search for neighborhoods [Samet, 1995], however there exist better data structures for this task, for example the k-d tree described in Section 2.1.2.

Although the concept of the octree stays the same, there exist different representations in memory. The used computer architecture has an impact on the used representation in memory. This means not all possible representations can be used under a given architecture. Therefore the following three subsections will cover three different concepts of representations in memory.

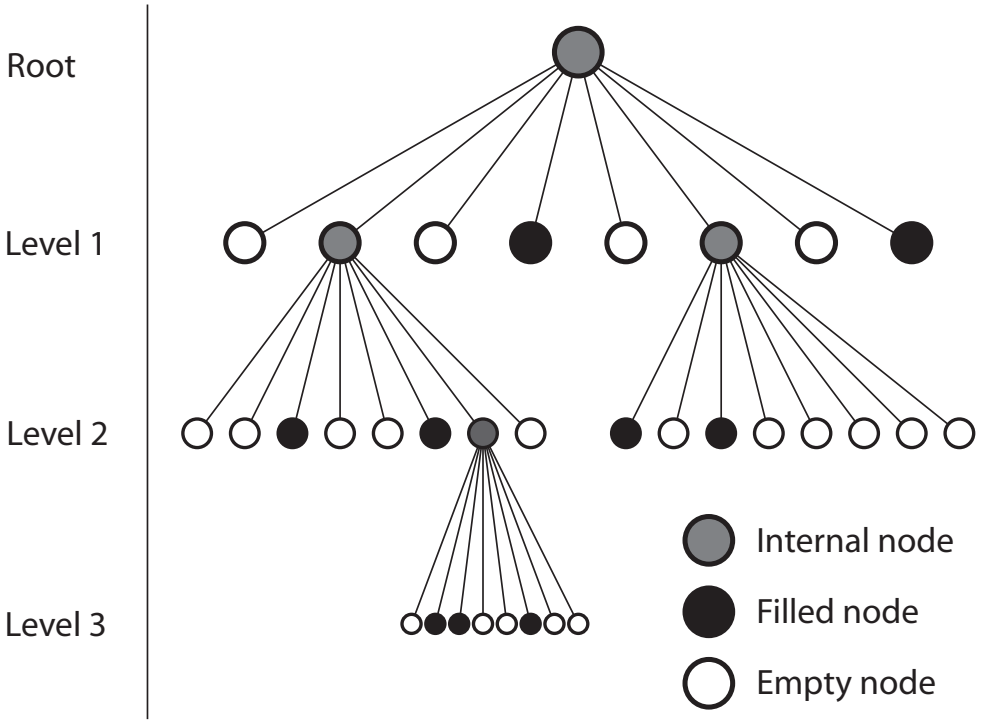


Figure 2.5: A pointer-based octree of level 3, the lines represent the pointers to the nodes. Black circles represent filled nodes, empty nodes are represented by white circles and intermediate nodes are represented by gray circles.

2.1.1.1 Pointer-Based Octree

A common representation in memory when working with trees is the usage of pointers to refer to a node's children. The same applies for pointer-based octrees. Intermediate nodes are marked as partially filled while the leaf nodes are either marked as full or empty. Often the terms black and white are used for filled and empty nodes, while intermediate nodes are referred to as gray [Samet, 1995]. An example of such a pointer-based octree can be seen in Figure 2.5. In its simplest implementation, an octree would hold a pointer for each octant per node, which would not be memory efficient. To reduce the number of pointers required, as a first step only the filled/black nodes are stored in the data structure [Samet and Webber, 1988a]. It can be reduced even further if each octant holds only a pointer to its first child and to one of its siblings. Although this reduces the memory consumption, the required amount of operations during traversal increases [Lewiner et al., 2010].

2.1.1.2 Linear Octree

When linear octrees and linear quadtrees were originally proposed by Gargantini [Gargantini, 1982a] [Gargantini, 1982b], the purpose was memory efficiency. As already stated above, pointer-based octrees are not memory efficient in the sense that they hold more pointers to its

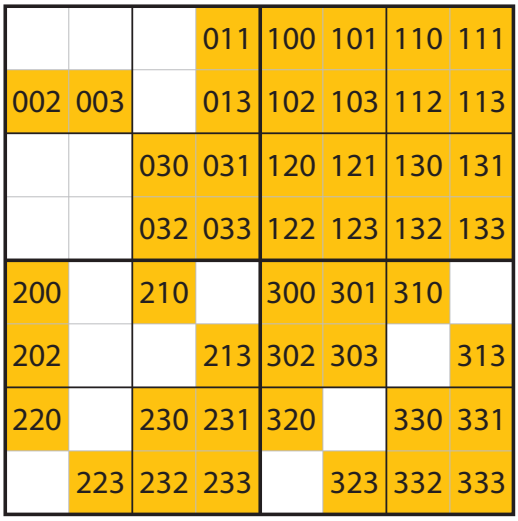


Figure 2.6: An example of a linear quadtree with depth 3. Note how much information is duplicated especially in the upper right quadrant. Figure adapted from [Gargantini, 1982a]

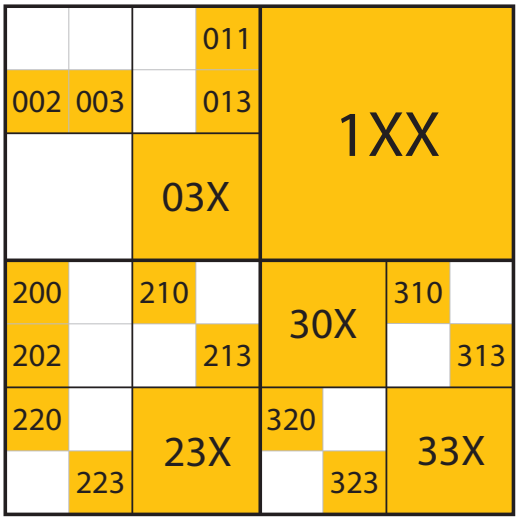


Figure 2.7: The compressed version of the linear quadtree from Figure 2.6. The duplicated information is now reduced. Figure adapted from [Gargantini, 1982a]

children than necessary. Gargantini therefore proposed the following method which does not require the usage of pointers: Each quadrant of the quadtree denotes its children using following encoding: 0 for the NW quadrant, 1 for the NE quadrant, 2 for the SW quadrant and 3 for the last quadrant. Using this encoding, each node can be represented as a digit of base 4 storing its hierarchy. Figure 2.6 shows the approach on data resulting in a linear quadtree of depth 3. After the encoding is completed for all nodes, they can be sorted and result in the linear quadtree. Gargantini further claimed that it is enough to store only the black/filled nodes in the linear quadtree to save space in memory. A pointer-based quadtree can naturally represent different levels of subdivision by stopping the splitting process at a given level. In the linear quadtree, all quadrants have the same depth. This is inefficient in terms of memory usage if all children of one node are filled. Therefore Gargantini proposed a method to combine such cases. If four nodes differ only in the last digit, they can be combined into just one node, setting the last digit to a value higher than 3. This step can be repeated until not all four children of a node are available or the first digit is reached. Figure 2.7 visualizes the outcome of this process. A linead quadtree that has not been reduced in size requires 2 bits to encode on level. This number increases to 3 bits per node when the node reduction is applied because the reduction information is encoded with a value larger than 3 [Gargantini, 1982a].

The same representation can be done with octrees, just extending the encoding to the octants, therefore having numbers from 0-7 used to represent the hierarchy of an octant in the octree. Besides the memory efficiency, linear octrees can be used on architectures where pointers are not available or have other side effects. This is the case when working with CUDA (explained

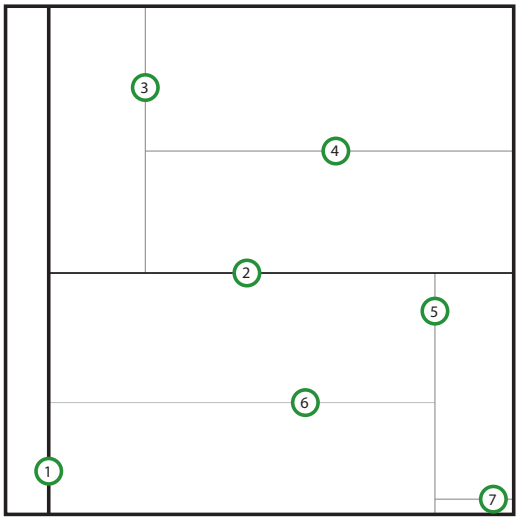


Figure 2.8: The data already used in Figure 2.3 and Figure 2.4 represented in a k-d tree.

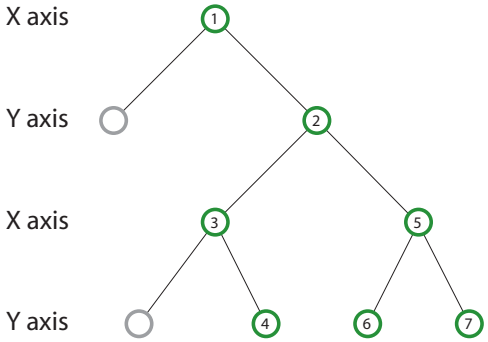


Figure 2.9: The k-d tree from Figure 2.8 as tree. Note how the splitting axis changes after each level of the tree.

in Section 2.4.1). As one main part of this thesis is the development of a CUDA-based point-selection and marking method based on octrees, a pointer less octree data structure is needed.

2.1.1.3 Hashed Octree

A variation of the linear octree is the hashed octree. As hash function, which defines the position of a specific node inside the hash table, a function like the Morton code presented in Section 2.2 can be used. Usually, there are only the k last bits of the Morton code used as a key for the position in the hash table [Warren and Salmon, 1993]. Collisions can occur when he hash function maps two nodes to the same entry in the hash table. They can be resolved by using a linked list which allows to add several nodes at the same position of the hash table [Warren and Salmon, 1993].

2.1.2 K-d Tree

Another spatial data structure is the k-d tree, also referred to as multidimensional binary search tree. It has been developed by Bentley [Bentley, 1975]. It is able to represent n-dimensional data. In its name k-d tree the k refers to the dimensionality. Each inserted point splits the space along one axis. In the two-dimensional case, this would mean the first point inserted splits along the x axis, the second point splits along the y axis. This results in a tree where a node holds two children representing the space at the left and the right side of the node. This can make it memory efficient compared to the octree depending on the complexity of the represented data [Samet, 1990]. A representation of a k-d tree can be seen in Figure 2.8 with its tree representation in

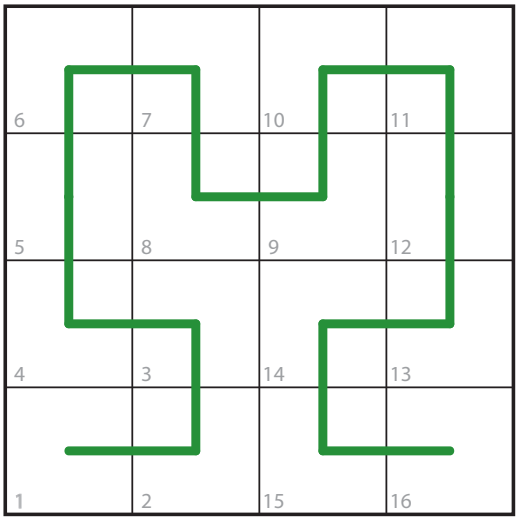


Figure 2.10: Construction of the Hilbert curve at iteration 2. Note how the squares are labeled from 1 to 16 such that each numbered cell shares an edge with its numbered predecessor. Once the labeling is completed the line can be drawn continuously starting from cell 1 till cell 16. Image from Hilbert [Hilbert, 1891].

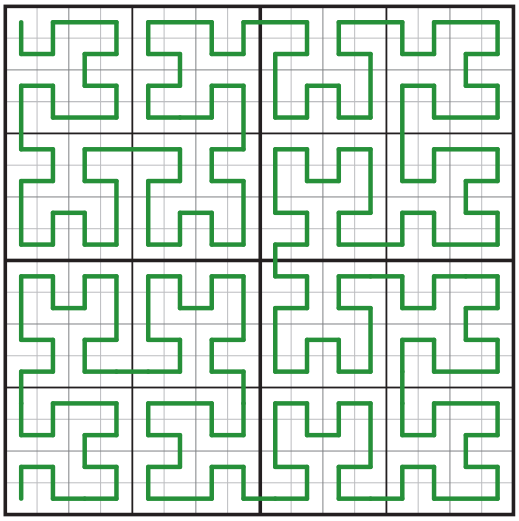


Figure 2.11: An example of the Hilbert curve at iteration 4. It can be observed that it traverses a quadtree of level 4. However its traversing is different to the Morton code represented in Figure 2.12. Figure adapted from Hilbert [Hilbert, 1891].

Figure 2.9. Moreover, the k-d tree is more suitable for neighborhood search with complexity of O(log n), supposed that the k-d tree is balanced. Bentley therefore proposed an optimizing approach to create a balanced k-d tree [Bentley, 1975]. A generalization of the k-d tree is the BSP-tree, which does not require the splitting axes to be orthogonal to each other [Fuchs et al., 1980].

2.2 Morton Code

As shortly mentioned in Section 2.1.1.2, linear octrees help avoiding pointers and are therefore suited for CUDA. For easier construction of linear octrees the Morton code can be used. Morton code or Morton order has been introduced by Morton [Morton, 1966].

Generally, the Morton code provides a mapping from an n-dimensional space to the 1D space, which results in one type of a space-filling curve. Space-filling curves as the name implies, are nothing else but curves which are able to fill an area or space with one curve. Besides the Morton curve there exist other space-filling curves such as cantor-diagonal order and spiral order [Samet, 1990]. The curve closest related to the Morton curve is the Hilbert curve, which has been introduced by Hilbert [Hilbert, 1891].

When describing the construction of its curve, Hilbert showed it at the example of the two-dimensional case. He used the unit square and split it into four equal non-overlapping squares along two orthogonal axes. Those four squares are sequentially numbered such that each numbered square shares one edge with the square holding the previous number. Now the line can be inserted starting from the center of the first labeled square and continuously passing the center of the remaining squares. This process can be continued by applying the splitting and labeling process with each of the four squares. This leads into the Hilbert curve. See Figure 2.10 for a construction example after two iterations. The same division approach can also be seen in the quadtree and the octree when extending the Hilbert curve into the third dimension. A combination of a quadtree and its represented Hilbert curve is visualized in Figure 2.11.

The Morton code shows a similar behavior when constructing it for the two-dimensional case. As shown by Orenstein [Orenstein and Merrett, 1984], there exists a conversion from n-dimensional data to linear space. Consider the three-dimensional case with the variables x, y, z. The encoding concatenates the variables by shuffling the n bits as shown in Formula 2.1

$$x_0, y_0, z_0, x_1, y_1, z_1, x_2, \ldots, x_{n-1}, y_{n-1}, z_{n-1} \tag{2.1}$$

This encoding can be done in any dimension bitwise concatenating the values. Once the encoding has been computed for all points, they must be sorted, and the n-dimensional data is represented as one-dimensional data. Along the sorted points, data can now be searched for and the reversal of the encoding can be applied to retrieve the original data. Orenstein further observed that the mapping of the Morton code in two-dimensional space shows Z-like structures, so he introduced the name Z-ordering. This structures can be seen in Figure 2.12.

Both, the Hilbert curve and the Morton curve traverse all the quadrants in the quadtree. They can therefore be seen as a linear representation of a quadtree and can be used as index structure for it. Considering the encoding used by Gargantini [Gargantini, 1982a] in her proposed linear quadtree structure and the traversal of the two-dimensional Morton code, it can be observed that they are the same. The Morton code differs from the Hilbert curve in the sense that it has discontinuities along the curve while the Hilbert curve is continuous. On the other side, the conversion from n-dimensional space into the representation of the Morton code is easier to compute compared to the Hilbert curve [Samet, 1990]. This can also be observed when comparing Figure 2.12 and Figure 2.11. The Hilbert curve does not traverse all the quadrants in the same order.

Although the construction of the Morton code is simple, it has a property which maps well to quadtrees and octrees. In the case of the octree, each block of three bits represents one level of the octree. Therefore, when the list of Morton codes is sorted all codes that have the same prefix share the same parent node. Due to this properties and the simple construction, the Morton code is used in this thesis to build the linear octree.

2.3 Parallel Sorting

In the sections above, necessary concepts for an implementation of a pointer-less octree have been introduced. As has been mentioned, the last missing basic concept is a sorting algorithm. Sorting has been well studied over the last decades. However, as the octree for selecting points

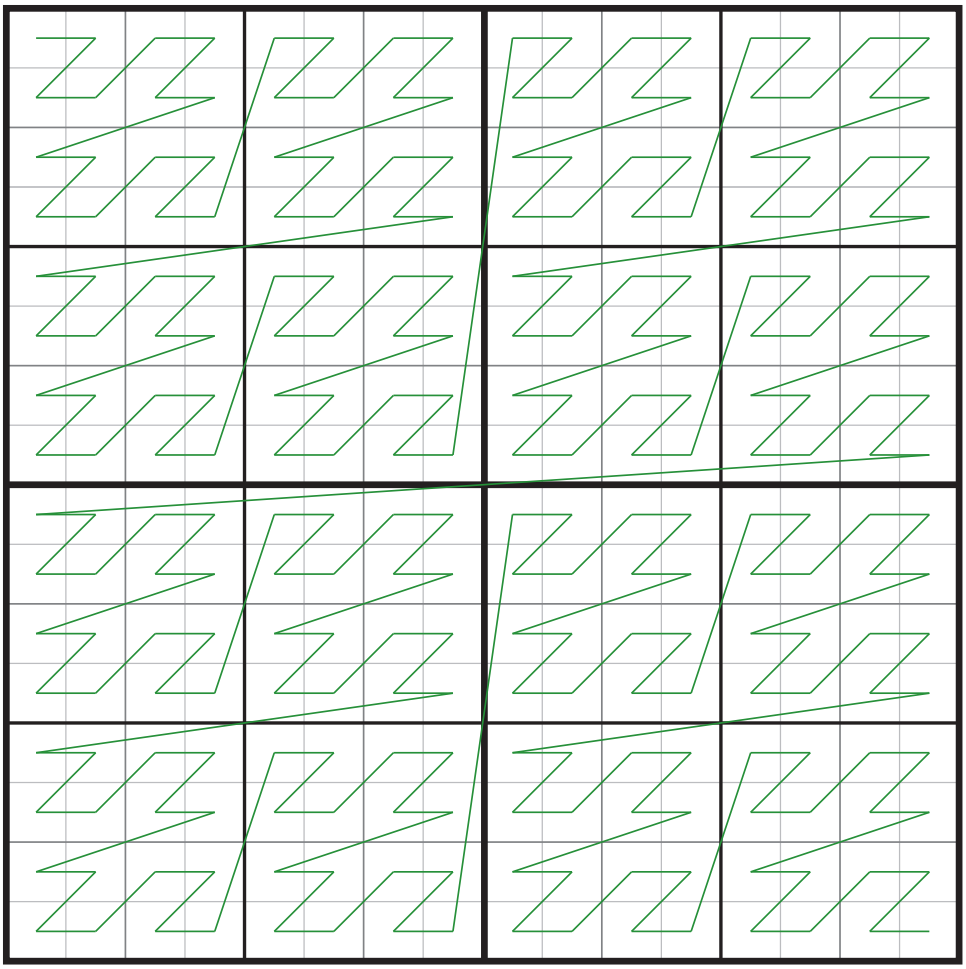


Figure 2.12: A Morton Code represented as curve. It can be observed, that its traversal is the same as the linear quadtree proposed by Gargantini [Gargantini, 1982a]. Compared to the Hilbert curve in Figure 2.11 discontinuities can be observed when moving from one quadrant of the quadtree to the next. Figure adapted from [Orenstein and Merrett, 1984]

should be implemented completely on the GPU in CUDA, a parallel sorting approach is needed. The first parallel sorting algorithm was introduced by Batcher [Batcher, 1968] and several others followed. While the theoretical complexity is the same for several sorting algorithms, it cannot simply be used as a reference to choose from. It is not straight forward to map the theoretical complexity of an algorithm to its practical running time on a specific parallel architecture [Blelloch, 1996]. Flynn distinguished between four architectural types. The only non-parallel system is the *single-instruction stream-single-data (SISD)* architecture. In it one process is operating on one piece of data at a time. *Single-instruction stream-multiple-data (SIMD)* provides several data items which are all processed by one instruction on a given timeslot. *Multiple-*

instruction stream-multiple-data (MIMD) processes several data streams with possibly different instructions at the same time. *Multiple-instruction stream-single-data (MISD)* is mentioned for completeness [Flynn, 1972]. Blelloch et al [Blelloch et al., 1991] analyzed three different sorting algorithms namely bitonic sort, radix sort, and sample sort on a SIMD architecture. They showed that radix sort was the fastest of those three, however, sample sort outperforms the radix sort when the ratio of number of sorted keys per processor increases. Amato et al. [Amato et al., 1998] further extended the findings of Blelloch et al. by investigating an implementation of the same three sorting methods using three different parallel architectures. They could verify that the architecture has indeed an impact on the runtime of the sorting algorithm. Both surveys came to the conclusion that the radix sort shows a good performance compared to the other sorting approaches. It worked well on different amounts of sorted data and different numbers of key/processor ratios. Additionally, radix sort is easier to implement and it is stable [Blelloch et al., 1991]. While sample sort worked well on the architectures investigated by Blelloch and Amato, Tsigas et al. [Tsigas and Zhang, 2003] showed that it preformed 50% slower than the compared quicksort on a their chosen architecture. All these comparisons show that it is rather difficult to choose a sorting algorithm for a specific parallel architecture because its performance heavily depends on the underlying architecture.

2.3.1 Radix Sort

Radix sort is not comparison based as for example quicksort, and its running time is $O(n)$. This means the sorting algorithm does not compare the elements with each other to sort them. Instead of comparing the elements with each other, the elements are divided into parts which can then be used to sort the elements. Therefore, running time does depend on the length of the elements, which are also referred as keys. It cannot be applied to non-number data types. It works naturally with integers but it can be extended to work with floats [Blelloch et al., 1991]. The algorithm takes *n* bits of each key per pass and sorts the keys along these n bits. As the key is an integer with length 32 or 64 bit usually 4 or 8 bits per pass are chosen. To sort the *n* bits any sorting algorithm can be chosen. The only condition is that it must be stable. Stable means that the order of two elements, which are equal and follow after each other, must be preserved when the sorting algorithm has been completed. Since the digits sorted at one step in time are small (4 or 8 bits) *counting sort* with its running time $O(n)$ can be chosen as sorting algorithm [Leiserson et al., 2009]. To sort *n* keys, each holding a value between *0* and *k*, two additional fields besides the field which holds the sorting data are necessary. First a histogram is created storing all the occurrences from *0* to *k* of the keys. Next each field in the histogram is added to the previous entry in the histogram field. As a last step the value for each key *i* is used as the index for the histogram. The *ith* value of the histogram is the final position where *i* must be put in the output field to be sorted and the histogram entry is decreased by one. This algorithm is able to sort without comparing the keys as a whole. An example of the radix sort algorithm can be seen in Figure 2.13.

When converting this algorithm such that it can run on a parallel machine some changes are necessary as proposed by [Blelloch et al., 1991] and [Zagha and Blelloch, 1991]. This thesis will focus on the solution proposed by Zagha et al. as it introduces concepts necessary for the implementation in CUDA.

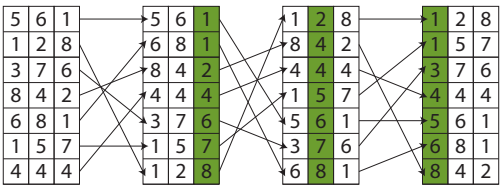


Figure 2.13: Radix sort sample. Figure adapted from Zagha et al. [Zagha and Blelloch, 1991].

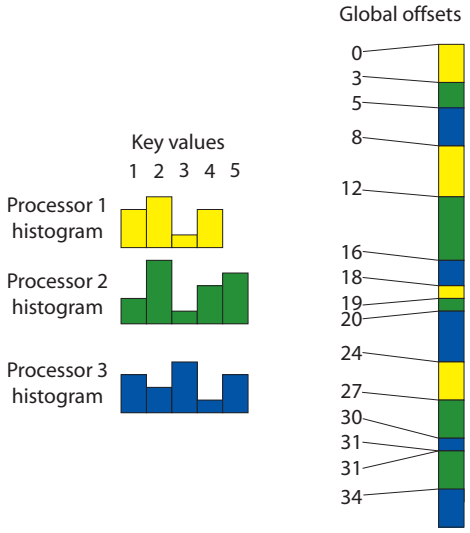


Figure 2.14: A sample of the histogram approach used by Zagha et al. Each processor uses its own histogram which are combined to global offsets. Figure adapted from Zagha et al. [Zagha and Blelloch, 1991].

In the parallel approach the n keys that have to be sorted are distributed over p processors. When p processors work in parallel on the same histogram field, read-write conflicts can appear when two or more processors try to access the same entry at the same time. When each processor locks the entry before accessing it, the performance of the algorithm can be reduced if many digits hold the same value. The solution to this problem proposed by Zagha et al. [Zagha and Blelloch, 1991] is to assign to each processor its own histogram. This solves the problem of concurrent access. As can be seen in Figure 2.14, each processor creates its own key-histogram according to the current position. After each processor has finished the histogram computation, the histograms are combined into a single one such that the keys can be globally sorted.

The combination of the histograms starts with the first element of each histogram continuing with all the second elements and so on. Summing up all the entries can be done in parallel using the method proposed by [Ladner and Fischer, 1980]. This resulting histogram can be used as final output by all processors in the same way as in the non-parallel version of the algorithm.

2.4 General Purpose Computing

General Purpose Computing refers to the possibility to compute an arbitrary task on the graphics processing unit (GPU). Initially, GPUs as the name implies processed nothing else but graphical data to display an image on the screen.

The rise of APIs like *DirectX* by *Microsoft* and *OpenGL* supported the developers by implementing graphical tasks on the GPU. A direct programming of the GPU however was still not possible. It was more like a definition on what should be rendered how, the details where done inside the API not allowing the developers to introduce their own code for rendering. The developer defines the geometry that should be rendered and defines the parameters for lighting and the final output was calculated by the API/GPU combination. This is referred to as fixed-function-pipeline. Starting from *DirectX 8* and *OpenGL 2.0*, it was possible for the first time to get control over certain parts in the fixed-function pipeline. The processing of the vertices and the rendering of the pixels can be controlled with *shaders*. They are basically small programs which run on the GPU but can be implemented by the developer. *Vertex shader* gave control over the input vertices and *pixel shader* over the rendered pixels [Segal and Akeley, 2004].

Both the computations for each pixel and each vertex can be executed in parallel, because the results for each pixel and vertex are independent of each other. This led to graphics cards with the possibility of running many shaders in parallel because the number of shaders continuously increased with each graphics card generation. Knowing that the graphics card evolved to a highly parallel architecture compared to the CPU, the GPU became an alternative to the CPU when the algorithm could be parallelized and ported to the GPU.

Although it was the purpose of shaders to control the appearance of vertices and rendered pixels, the methodology could be used to compute tasks beyond simple rendering. One possibility to do so was a three-step process. First a texture is filled with the data the algorithm has to deal with. Next the texture can be passed to the GPU and the pixel shader performs an operation on each texel. As the last step, the modified texture is read from the GPU memory and the result can be used on the CPU [Buck and Purcell, 2004]. This workaround has been used for several different tasks [Luebke et al., 2004].

In 2007, Nvidia came up with a first solution which makes workarounds with shaders obsolete. From that time on, developers could run general purpose algorithms directly on the GPU. Other solutions came up such as *OpenCL* and *DirectCompute*. The capabilities of these solutions will be discussed in the following subsections.

2.4.1 CUDA

From the *GeForce 8800* onwards any *Nvidia* graphics card has been able to work with CUDA. CUDA is basically an extension of C which allows directly running code on the GPU in parallel. Workarounds such as the one mentioned above were no longer necessary. This allowed also developers which were not familiar with the concept of shaders and graphics APIs such as *OpenGL* and *DirectX* to run general purpose code in parallel on the GPU [Garland et al., 2008]. When developing with CUDA it is typically distinguished between the *host* and the *device*. The host is hardware device on which the CUDA-capable device is installed. The device is the piece of hardware on which the CUDA code can be executed. Therefore on a common desktop machine, source code developed for the host runs on the CPU and uses the main memory, while the device code runs on the GPU using the faster graphics card memory. To distinguish between the device code from the code running on the host CUDA, therefore, introduced the __global__ and __device__ function prefixes. Functions with the __global__ prefix are also referred to as kernels and can be called from the host source code. The functions prefixed with __device__

can only be called from a __global__ function [Cook, 2013]. Any kernel function can run in parallel on the GPU. To specify the level of parallelism CUDA extended the function call with the following pattern:

```
function_name<<<num_of_blocks, num_of_threads>>>(paremeters);
```

CUDA hides the details regarding parallelism from the developer. This way, he can focus on the development of parallel algorithms and does not have to bother with specific details of the underlying hardware.

```
1  __device__ int sum(int number)
2  {
3          return number + number;
4  }
5
6  __global__ paralellSum(int *input, int *ouput)
7  {
8          // assume that thread_id is unique for each thread. Starting at 0
               for the first thread and then continuously increased for each
               thread.
9          ouput[thread_id] = sum(input[thread_id]);
10 }
11
12 void hostFunction()
13 {
14         int *data = new int[256];
15         //assign values to the data array
16         int *results = new int[256];
17         paralellSum<<<8, 32>>>(data, results);
18 }
```

Listing 2.1: Basic CUDA sample which calls one kernel function

Listing 2.1 shows a simple CUDA example. The kernel function is `paralellSum`, which is called from the host in the `hostFunction` function. To execute 256 threads in parallel, 8 blocks are used holding 32 threads each. The kernel processes each of the 256 input elements in parallel by calling the device function `sum`. This is a simplified example because the data has to be copied to the graphics card memory such that it can be used in CUDA. The copy instructions are omitted for simplicity.

2.4.1.1 Architecture

In Section 2.3 the problem that an algorithm can work very efficient on one type of parallel architecture while it does not on another was briefly discussed. Moreover it takes effort to optimize the algorithm such that it works well with the provided computer architecture. *Nvidia* had in mind that the level of parallelism will increase from each generation of GPUs to the next. Therefore, they designed the structure of their system in such a way that an adjustment of the algorithm should not be necessary when a new GPU is released.

On the conceptual level that is visible for the developer, the concepts threads and blocks are the dominant terms. Similar to the multi-core CPUs, threads run in parallel in CUDA. They are grouped to blocks and all blocks that are executed with one kernel call represent the grid. An overview can be seen in Figur 2.16. All threads within this grid are logically executed in parallel on the GPU by calling a given kernel function. This does not mean that all threads are executed in parallel on the hardware. How many threads are actually executed in parallel depends on the underlying CUDA hardware. However, the developer can assume that all threads are executed in parallel.

To support the development with volume and image data, *Nvidia* allows to order the blocks in the grid either three dimensional or in the case of images two dimensional. The same holds also for the alignment of the threads inside a block. This alignment of threads and blocks can be imagined as if a multidimensional array is defined. The benefit of this alignment is that the data can easily be accessed in a per-thread basis, such that each thread can work on one voxel or pixel. This allows the developer to define for each kernel a given amount of blocks and threads, independent of the number of parallel cores that are actually available on the GPU. Figur 2.16 shows two different block/thread combinations. CUDA maps the specified thread blocks to the available resources on the GPU. [Cook, 2013].

To make this work, the CUDA-capable hardware consists of one or several streaming multiprocessors (SM), which have been renamed to SMX on the latest Kepler architecture. Each of these SMs holds a defined amount of streaming processors (SP). The first CUDA-capable devices had eight SPs, this number continuously increased to 192 SPs in the current generation of GPUs [Nvidia, 2012]. Figure 2.15 shows the hardware structure of CUDA-capable devices. While each SM can execute up to 16 blocks, each block is executed in one SM. The threads of each block are grouped together to warps. A warp holds 32 threads of a block. The threads inside a warp are physically executed in parallel on the GPU. Therefore, all threads inside a warp execute the same instruction at a given step in time. This means branching must be considered. If the threads in one warp do not follow the same branch, those who do not enter the branch are idle until the other threads exit the branch. Although the divergence in the branches is fully handled by CUDA it still causes a loss in performance, since not all threads are active [Garland et al., 2008].

CUDA differs between four different types of memory. Each streaming multiprocessor holds an amount of registers. They are shared between all the streaming processors of the SM. A small block of 64K of Constant memory is available for a fast access to read-only variables. This memory can be read by all threads in the same grid [Huang et al., 2008]. The other two memory concepts are global and shared memory. Their properties are covered in the Sections 2.4.1.3 and 2.4.1.4.

The architecture of each CUDA device is called *single-instruction multiple-thread (SIMT)*, which is in principle a SIMD architecture. One crucial difference is that threads located in different warps do not have to enter the same branch or become idle otherwise [Nickolls et al., 2008].

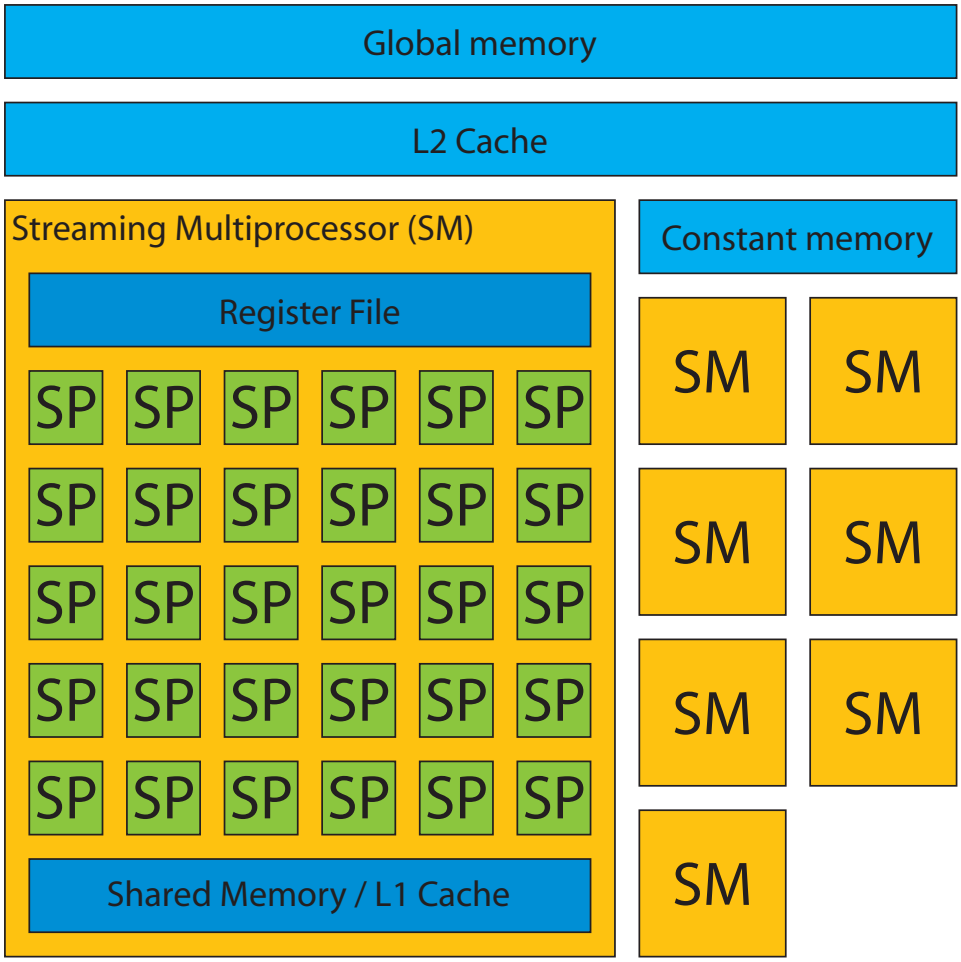


Figure 2.15: Hardware layout of a CUDA-capable device. The number of streaming multiprocessors (SM) and streaming processors (SP) varies through the different device generations and device types. Figure adapted from [Nvidia, 2009].

2.4.1.2 Compute Capability

Nvidia introduced different versions of compute capability (CC) to specify what kind of operations and hardware specifications a CUDA-capable device fulfills. It starts with CC version 1.0 and goes up until version 3.5 for the latest Kepler architecture. On the hardware side, the number of registers per streaming multiprocessor continuously increased with each version. The number of registers is important for the occupancy of the GPU. Each thread uses registers for its local variables. As the amount of registers is limited per SM, a high usage of registers per thread limits the amount of warps that can be executed in parallel on a SM. This can lead to a occupancy of less than 100% and causes a loss in performance because not all streaming processors are active,

or in the worst case the kernel cannot be executed if there are not enough registers available. A solution to this problem is to reduce the number of threads per block or to reduce the number of registers used per thread, which requires a change in the source code.

From version 2.0, which was introduce with the *Fermi* architecture, the amount of shared memory increased from 16Kb to 48Kb. What can be done with shared memory will be discussed in Section 2.4.1.4.

2.4.1.3 Global Memory

The global memory is located in the GDDR memory directly on the graphics card. This memory holds between 512 MB and 2 GB on current generation graphics cards. Data stored in global memory can be accessed from all threads that are located on the same grid.

Compared to the registers and the shared memory it is about ten times slower than shared memory. The data that should be processed has to be copied from the host to the global memory to make it available on the GPU. Once the final result has been computed, the results have to be copied back to the host to make the output available for further processing in the application.

To get fast access on the data stored in memory only a few patterns are allowed to get a so called fast coalesced access on global memory. Coalesced access allows reading 32, 64 or 128 bytes with one access to memory instead of accessing each field individually. CUDA devices with compute capability below version 1.2 had the strongest requirements for coalesced access. To get a coalesced access the *ith* thread of a half-warp has to access the *ith* word of the 128 bytes block. From version 1.2 onwards this restriction has been weakened, now the access is coalesced if all threads access data within one 128 byte segment.

From version 2.0, the L1 cache has been introduced. When caching is active the global memory is read as 128 byte per cache line. When a warp accesses different parts of the global memory in a non coalesced way, it can be switched to read 32 byte per segment to keep the bus utilization high.

2.4.1.4 Shared Memory

The shared memory is a fast on-chip memory available per multiprocessor and is also referred to as a developer controlled L1 cache [Cook, 2013]. Its size is limited to 16 KB per multiprocessor for CUDA cards with compute capability before 2.0. From version 2.0 onwards, the L1 cache and the shared memory share 64 KB. Either 16 or 48 KB can be reserved for shared memory. Shared memory can be accessed by all threads in the same block. Therefore, it can be used as a communication mechanism between threads.

Due to the bandwidth difference between global memory and shared memory it is common practice to load the data from global into shared memory. The data can be modified in shared memory and stored back to global memory. This is especially important to support coalesced data access and if the data is used multiple times and shared among the threads in a block. It is often the case that the data has to be stored back in a different order than it was read from global memory. Using the shared memory as a temporal variable allows reading and writing the global memory in a coalesced manner, as the shared memory does not have a strict access patters as global memory.

However, there is another restriction when accessing shared memory. In CUDA, shared memory is accessed by using banks. There exist 32 banks per warp and each bank can be used to access 32 bits per cycle. The data stored in shared memory is placed in consecutive order such that the first data value can be accessed using the first bank, the second value by the second bank and so on. After the 32nd value is reached the access starts again with bank one. Each bank can only be accessed by one thread in the same warp per cycle. Otherwise a bank conflict occurs. This bank conflict has to be resolved by CUDA and costs performance. Until this conflict is resolved, all other threads belonging to the same warp remain idle. The more threads access the same bank, the longer it takes to resolve the bank conflict. An exception to this problem is, when all threads of the same warp access the same bank. This does not result in a bank conflict. The developer has to be aware of this issue and resolve the bank conflicts by adding for example an offset to the shared memory index, when initially assigning values to shared memory [Cook, 2013].

2.4.2 Other Solutions

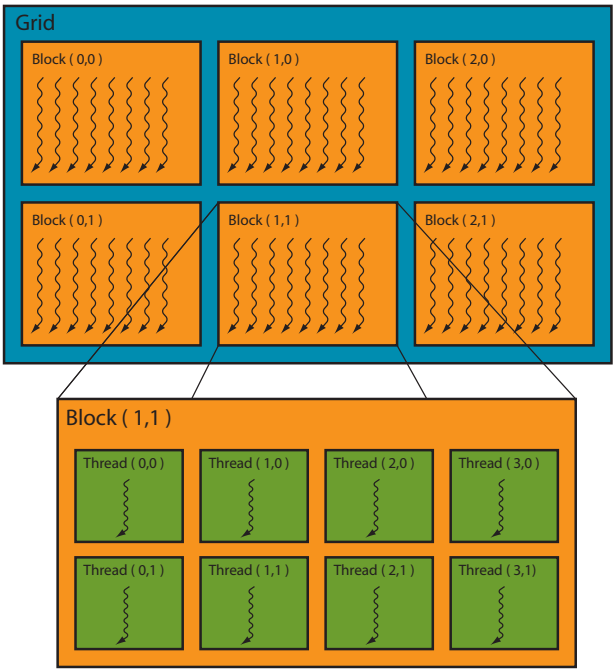
While Nvidia was the first introducing CUDA as a solution for general purpose computing on the GPU, there exist other methods too. Those are worth mentioning as CUDA is proprietary and it therefore only works on Nvidia GPUs.

The *Khronos* group created, similar to OpenGL as graphics API, an open standard for general purpose computing for parallel processors named OpenCL. OpenCL is not only supported by graphics cards from Nvidia and AMD but also multicore CPUs. The framework and architecture is similar to CUDA. Comparing the performance of CUDA and OpenCL, both show the same performance [Su et al., 2012] [Fang et al., 2011]. In OpenGL 4.3 *compute shader* have been introduced and allow an arbitrary computation in shaders.

Another alternative to CUDA created by Microsoft is *DirectCompute* solution. It is coupled to the latest DirectX 11 and is therefore only available for Windows.

The decision to use CUDA in this thesis was mainly because it was already used by other parts of Scanopy. This helps avoiding the dependencies to another general purpose solution.

KernelA<<<(3,2), (4,2)>>>();



KernelB<<<4,6>>>();

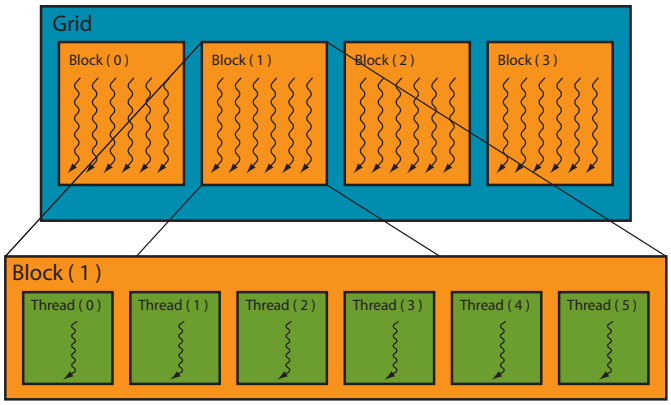


Figure 2.16: Two different configurations that can be used launching a kernel with CUDA. The first version uses a two-dimensional indexing which is useful to work on 2D data such as images. The second version uses a plain indexing which is suitable for one-dimensional data. Figure adapted from [Nickolls et al., 2008].

CHAPTER 3

Related Work

This chapter gives a detailed overview of the related work used in this thesis. The first sections describe the concept of annotations and how they are used in different fields. Special care is given to the sector of cultural heritage. Since the implemented point selection is integrated into Scanopy, the relevant parts of the point-cloud renderer will be described in Section 3.2. As one significant part of this thesis is the development of a point-selection method, a detailed overview of existing point-selection solutions is given. The last two sections cover state of the art methods to construct octrees on the GPU and give an introduction into Radix Sort in CUDA, which is crucial for certain CUDA octree creations.

3.1 Annotations

An annotation is defined as "...a note by way of explanation or comment added to a text or diagram..." by the Oxford Dictionary [Dictionary, 2004]. In computer science, an annotation can be any kind of added information [Bechhofer et al., 2002]. This can be as simple as defined in the Oxford dictionary but also images, audio or video files can be used as annotation. Also, the annotated data can be much more than text or diagrams as will be shown in the following Sections.

There exist a variety of different standards which allow annotating the data. They represent the annotation in a machine readable format, which allows querying the data using annotations. Common in the field of semantic web is the *Resource Description Framework* (RDF), which is used to describe resources and the relation between different resources [Hitzler et al., 2008]. Regarding computer graphics, MPEG-7 is of special interest. It allows the description of complete scenes using annotations.

There is one crucial distinction of annotations in computer science. The first type of annotations has the purpose to provide the viewer with additional information regarding a currently observed media. On the other hand, a semantic annotation adds information to data, such that the annotation becomes machine readable [Bechhofer et al., 2002]. Such semantic annotations

support searching for specific information and further allow automatically relating data by linking different pieces of information together using semantic annotations. This thesis focuses on annotations which provide additional information to the user. Semantic annotations will only be introduced briefly as they are not part of this thesis. However, some related systems combine both types of annotations.

One of the first fields where annotations have been used was in the field of image database systems [Tamura and Yokoya, 1984] but also image information systems used those annotations [Chang and Hsu, 1992]. Textual information is manually added to the images. Linking a textual description to the image should provide the possibility to search for images holding a specific content. Once the images have been annotated, query languages similar to SQL can be used to query the database [Chang and Hsu, 1992]. As the amount of images increased over the years, manual annotation became more and more time consuming or even infeasible. Several methods have been proposed to automatically annotate images with text by analyzing the content of the images [Zhang et al., 2012].

With the rise of the semantic web, other annotation based systems came up to annotate web pages. The system proposed by Bechhofer et al. [Bechhofer et al., 2002] allowed to add annotations to web pages in the form of texts and links to provide the viewer with detailed information. The annotated information is combined with the web page once it is loaded by the browser. Additionally, semantic annotations are added to the web pages. This information added to the web page allowed an automatic reasoning about the content of the web page and should support an automated information retrieval.

3D modeling tools and computer-aided design (CAD) software are used in teams. Instead of managing comments and change requests on the modeled object in a different file, the integration of such comments in the modeling software can improve the collaboration. This does not only reduce the management overhead by keeping fewer files in sync. It also gives the possibility to place information at the exact location it belongs to and allows to see it in its context. Kadobayashi et al. [Kadobayashi et al., 2005] described a system which allows to add comments on different parts of 3D models. As can be seen in Figure 3.1, those annotations are connected with lines to the corresponding parts in the 3D scene. Additionally, several annotations can be grouped such that they can be traversed consecutively along a path. To reduce the amount of visible annotations the concept of an *interactor* has been introduced. This should help to visualize only those parts of the annotations that are of interest to the viewer. Each annotation can be assigned to one interactor. The user can decide which interactor is active. Only the annotations which belong to the active interactor are visible.

Sin et al. [Sin et al., 2009] use the concept of annotations to sketch comments in 3D space. Similar to Kadobayashi et al., they put the comment directly at the referred position. This way, requests for changes can easily be expressed. They further proposed a so called *Sketch-Box*. This is a semi-transparent box, which is shown on the left side of Figure 3.2. Its form and size can be changed to approximate the annotated part of the 3D model. The sketch-box is combined with a sketched comment. This combined annotation has the advantage that it clearly states which part of the model is covered by this annotation as it is bounded by the box. Compared to other annotation systems providing comments in 3D scenes, this combination provides a better visibility among different point of views [Sin et al., 2009] as can be seen in Figure 3.2.

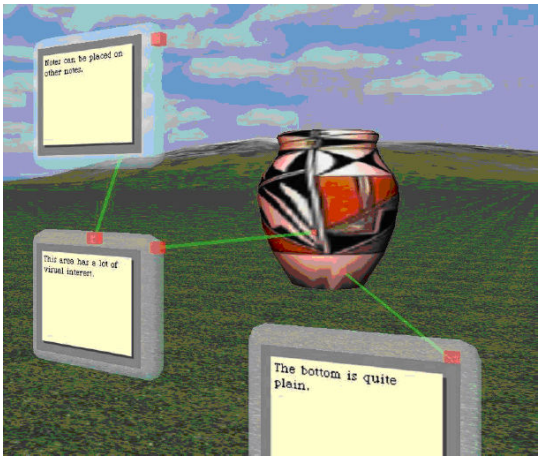


Figure 3.1: Annotation system presented by Kadobayashi et al. [Kadobayashi et al., 2006]. Textual annotations are added in the three-dimensional space. It further allows annotating already existing annotations with other textual annotations. The connection between annotation and annotated object is established with a line. Image courtesy of Kadobayashi et al. [Kadobayashi et al., 2006].

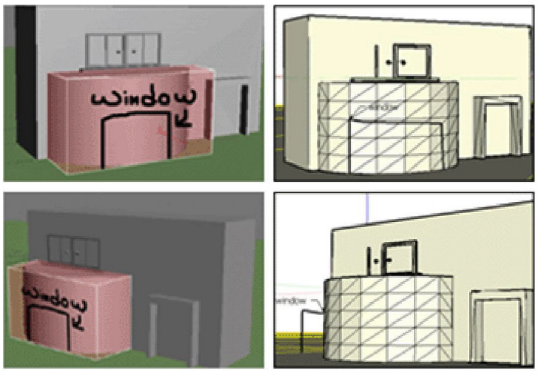


Figure 3.2: The annotations system created by Sin et al. [Sin et al., 2009] does not rely on lines as links between the annotations and the annotated objects. Instead, it creates a semi-transparent box around the annotated part of the object. This provides a better hint on what actually belongs to the annotation for different point of views compared to the line approach used by Kadobayashi et al. in Figure 3.1. Image courtesy of Sin et al. [Sin et al., 2009]

To improve the understanding of complex virtual 3D models, Sonnet et al. [Sonnet et al., 2004] proposed an method to add textual descriptions in the form of annotations to the model. Besides an explosion technique, which allows seeing inside the 3D model, several methods have been proposed to control the visibility of annotations. First of all, for each part of the model a center point is calculated, which is referred to as centroid. This is typically the closest point of the part compared to the center of its bounding box. Initially, annotations are not visible, but as soon as the user chooses a part of the model its annotation becomes visible. The amount of visible text of the current annotation depends on the distance between the mouse and the centroid of the chosen part and changes when the mouse is moved closer to or further away of the centroid. To place the annotation on the screen, the bounding box of the annotated part is used. The annotation is placed at one of the longest edges of the bounding box, which is also closest to the border of the screen. This helps to reduce the amount of space covered by the annotation on the object itself since the object should be located at the center of the screen. To show a clear connection between the annotation and object, a polygon is rendered between one edge of the annotation and the centroid.

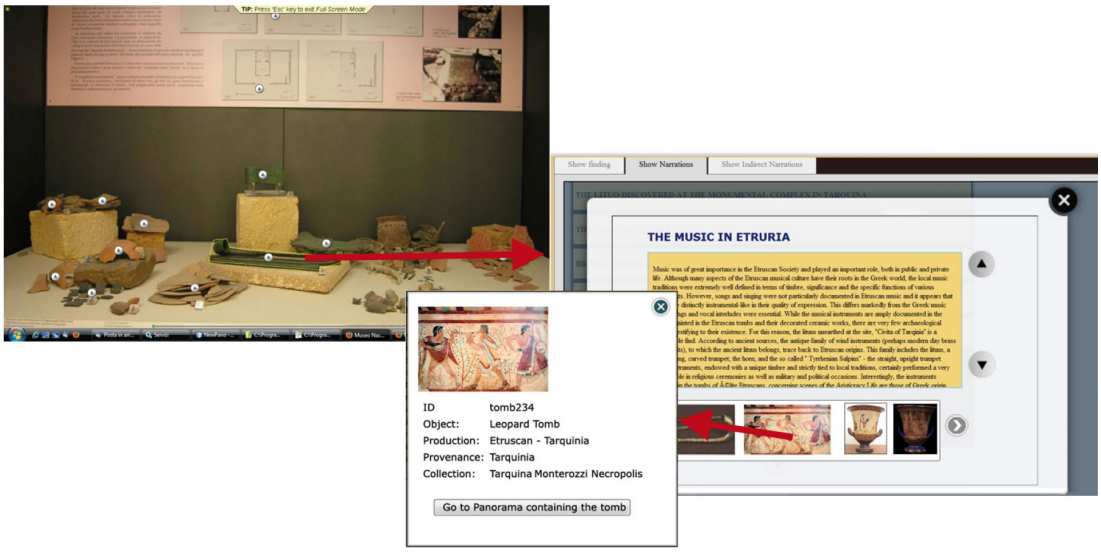


Figure 3.3: An annotation example from the DECHO framework proposed by Aliaga at al. [Aliaga et al., 2011]. The archaeological objects are represented as a picture. Annotations inside the picture are outlined and can be clicked by the viewer to receive additional information. The additional information is displayed in a new window, which provides a detailed textual description and a image gallery related to the archaeological object. Each image can be further investigated by opening it in a new window. Figure adapted from Aliaga at al. [Aliaga et al., 2011].

3.1.1 Annotations in Cultural Heritage

In cultural heritage the preservation of archaeological sites using computer technology gained importance over the last years. Besides the usage of laser scanners and photogrammetry as mentioned in the introduction, 3D modeling is another possibility to gain a virtual model of an archaeological site [Koller et al., 2010]. All annotation methods available have in common that they can support the archaeologists at their work. Providing additional data in form of annotations helps to structure the information. This is done by using a virtual model of the archaeological site. The available information is added to the model next to the position it belongs to. Another target group is the visitors of archaeological sites interested in the details about the structure.

A simple approach to represent cultural heritage with a computer represents the usage of images. With the DECHO framework, Aliaga at al. proposed a system to annotate images of archaeological sites [Aliaga et al., 2011]. The system is web based, and panoramic images are representing the archaeological objects. The annotated parts of the object are outlined and can be clicked on. All the related information about this object is collected from the database and presented to the user on a web page. Additionally, archaeological objects related with

the annotation can be further investigated virtually because they are all stored in the database. Objects not exposed in the current museum can be virtually visited in the museum they belong to by opening a new panoramic image. This image shows the room of the remote museum exhibiting the related object. The visualization is rather simple as can be seen in Figure 3.3. The complexity of this system lies on the database connecting the related objects and textual annotations with each other.

Another web based approach to annotate high resolution images has been proposed by Diaz et al. [Díaz et al., 2011] named ImaNote. The DECHO framework was static and could not be changed interactively. ImaNote allows the users to create annotations or change existing ones. Annotations are represented as squares on the high resolution image. They hold a title and a description. Images or other documents can be attached to the annotation by defining an URL to the file. To filter for specific types of annotations, they can be combined into groups. Although ImaNote allows specifying an icon or image in the annotation, both the DECHO and ImaNote framework have no possibility to show information about the annotation before opening it. The only hint about the annotation in the scene, represented as image, is the position and the outline of the annotation.

Attempts to introduce the third dimension for annotations in cultural heritage have been made by Yu et al. [Yu et al., 2011]. They introduced a web based application based on *X3D*. X3D is an XML based standard to represent 3D data such that it can be rendered in the web browser. The object data is represented as a mesh. To link an annotation with parts of the objects, a selection method was introduced. The user can create a polyline and the parts of the mesh in the polyline are selected. All selected parts of the mesh represent a link to the current annotation. The selection step can be repeated to refine the selection either by including other parts of the mesh into the selection or by removing previously selected parts. Whenever an annotation is chosen from the list, the selected polygons are highlighted to show the link between annotation and object. While Diaz et al. and Aliaga et al. did not introduce a possibility to show a preview of the annotations besides an outline, Yu et al. show the name of the annotation next to the highlighted selection. See Figure 3.4 for an example.

A completely different annotation approach has been proposed by Ma et al. [Ma et al., 2012]. They annotated traditional Chinese paintings, which are up to 5 meters long, with audio files. The paintings have been digitalized and are presented on a screen. From the visible geometry on the painting, a depth clue is estimated such that the audio can be embedded in the three-dimensional space. The visitor stands in a room surrounded by 5.1 channel audio. The 3D position of the viewer can be controlled by virtually panning and zooming through the painting. The previously annotated sound files can then be played depending on the position of the viewer.

3.1.2 Annotations in Point Clouds

Annotating point clouds is conceptually similar to meshes because both represent data in 3D. However, some important properties available in meshes are not present. Point clouds for example do not have a surface. Scanned areas might have holes resulting from occlusion issues when scanning an object. This makes it difficult to select a given area from the point cloud as it was done by Yu et al. in the case of 3D meshes [Yu et al., 2011]. Additionally, point clouds coming from devices such as laser scanners are often used to create a 3D mesh. Point cloud annotations

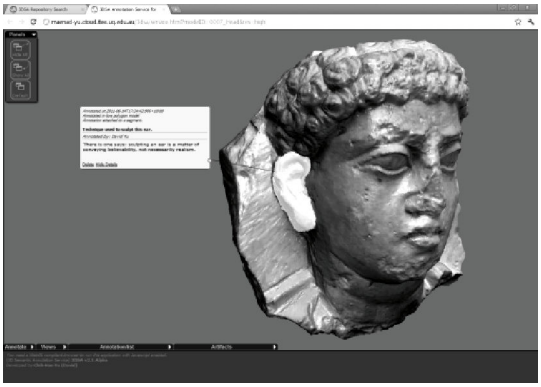


Figure 3.4: The annotation system described by Yu et al. The selected part of the 3D mesh is highlighted with white color. Textual annotations are linked to the highlighted part of the mesh over a thin line. Image courtesy of Yu et al. [Yu et al., 2011].

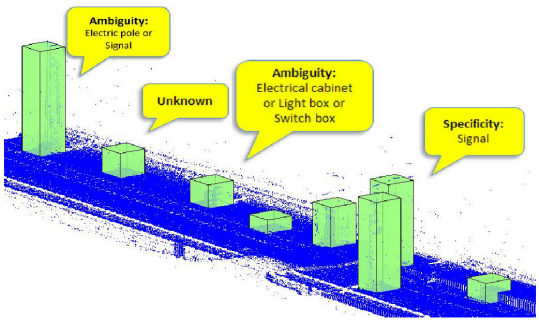


Figure 3.5: Simple point cloud annotations proposed by Truong et al. [Truong et al., 2012]. It uses speech bubbles to add textual descriptions. The annotations are linked with the point clouds with boxes. Image courtesy of Truong et al. [Truong et al., 2012].

are not very common until now. The most basic example of an annotation is the possibility to add a comment to a selected point. Such cases exist for example in the software packages "Bentley", "CloudCompare" and also Scanopy offers this feature. However, the possibility to create complex annotations linked to a specific part of the point cloud does not seem to be considered until now. Also the existence of complex annotations holding different types of media combined with the possibility of filtering the annotations could not be found in the literature.

The only system that uses annotations in point clouds, which has been found in the literature, was proposed by Truong et al [Truong et al., 2012]. It focuses on the topic of automatic semantic annotation of point clouds. Therefore, it is not in the scope of annotations providing deeper information to the user. However, they propose a solution to link parts of the point cloud with their annotations. As can be seen in Figure 3.5, this is done with boxes, and these boxes are annotated with labels to represent the detected structure of the point cloud.

3.2 Point-Cloud Renderer

With the increased point-cloud resolution produced by laser scanners and advanced registration techniques, point clouds have become bigger and bigger over the years. This progress has lead to point clouds which can hold more than 10^9 points nowadays. This amount of points can no longer be visualized completely due to several hardware limitations. First, the available main memory on actual computer hardware is not big enough to hold the whole point cloud. The same is true for the memory on graphics card on which the data must be copied to render the point cloud. The solution to this problem is to load only parts of a point cloud from the hard disk to the memory of the graphics card where it can be rendered. This is often referred to as *out-*

of-core [Gobbetti and Marton, 2004] [Scheiblauer and Wimmer, 2011]. It is crucial to decide which parts of the point cloud should be rendered since a wrong decision results into a rendering perceived as incorrect by the viewer. The two proposed methods use view frustum culling and level of detail rendering and depend on spatial data structures. Data structures which allow rendering only parts of the point cloud do not seem to become obsolete in the short future. Not only the memory storage and processing power of the hardware increases, but also the amount of points per point-cloud model increases at the same time. Whether or not a point is rendered depends on whether it is visible from the viewing direction and the distance to the viewer. Points that lie closer to the viewer are preferred when choosing points for rendering, compared to points laying further away.

The method proposed by Rusinkiewicz et al. [Rusinkiewicz and Levoy, 2000] uses a tree of bounding spheres holding the data of the point cloud. Nodes of the tree that do not lie within the view frustum can be culled away with this method. Level of detail rendering is introduced by comparing the size of a node's bounding sphere in screen space with a defined threshold. If the size of the bounding sphere is smaller than the threshold, the node is not rendered. To guarantee a specific frame rate, the threshold is adapted from frame to frame. Due to its nature, the octree is another possibility to render point clouds supported by view frustum culling and level of detail.

The point-cloud renderer *Scanopy*, used in this thesis to add annotations, organizes the points in a special octree named *Modifiable Nested Octree*(MNO) [Scheiblauer and Wimmer, 2011]. The MNO subdivides the space the point cloud is placed in such a way that each node holds small parts of the complete point cloud. The number of points that are allowed to lie within a given node is limited by the free cells of a 3D grid inscribed to the bounding box of the node. When points fall into already occupied grid cells, they are not stored at the node but rather moved into the according child node. This introduces level of detail in the MNO. Nodes close to the root hold a rather coarse representation of the point cloud. Nodes at the leafs of the MNO hold a detailed representation of the point cloud. This method allows a fast view frustum culling, because the individual nodes of the MNO can be tested against the view frustum. Also parts of the octree that are closer to the viewer are rendered in higher detail by rendering nodes closer to the leaf which hold more data about a given region of the point cloud.

Scanopy distinguishes between three basic storing positions for the point data. Each node of the MNO is available as a file on the hard disk [Scheiblauer and Wimmer, 2011]. When the node is required to render a different view of the point cloud, it is loaded to main memory and as a next step it is put on the graphics card memory where it can be rendered. Each node is either not loaded and, which means it is only located on the hard disk, or it is either placed in main memory or in the graphics card memory. However, it is not the case that the same node is available in main memory and graphics card memory at the same time.

3.3 CPU-based Point Selection

When visualizing annotations, a crucial part is to give a clear hint which part of the data is actually annotated. As shown in Section 3.1.1, an outline can be used to visualize such a connection when images are annotated. In the three dimensional case the outline can be extended to a box or

another polyhedron [Truong et al., 2012] [Sin et al., 2009]. However, highlighting the relevant areas of a point cloud is preferred because it provides a more accurate link between annotation and annotated object [Yu et al., 2011]. This makes it necessary to establish a point selection method.

Scheiblauer et al. integrated CPU-based point selection in Scanopy [Scheiblauer and Wimmer, 2011]. The key aspect when adding a point-selection method inside such out-of-core rendering systems is that not all data of a selected area is necessarily loaded when doing the selection process. When working with an in-core rendering system, all data is available in memory during the selection process. Therefore, each selected point could be marked with a flag to remember if it is currently selected or not. This does not work in the case of out-of-core systems because not all points of the currently selected region can be stored in memory. Loading all the necessary points from the hard disc to complete the selection process is inapplicable for two reasons. First, the data cannot be loaded fast enough from disc to be completely available while selecting. The other problem is the size of the data, which might not fit completely into memory.

The solution proposed by Scheiblauer et al. was the introduction of a so-called *selection octree*, which is used to store a selection volume instead of selecting the points individually. To select the points a three dimensional brush with the shape of a sphere or a box is used. The brush can be moved on the point cloud and everything in the brush's volume shall be marked as selected. Only currently visible points are used to build the selection octree. The root node of the selection octree has the same dimensions as the root node of the MNO. After each brush stroke, the just selected points are filled top-down into the selection octree. The points are filtered down the octree levels until each octree node holds only selected or unselected points. This step can be repeated until the desired selection is completed. The selected area is represented by a selection octree. Points that are loaded from disc because the viewpoint changed are tested against the selection octree and marked as selected if they are within a selection octree node which encompasses only selected points. Figure 3.6 shows a point cloud with a selection. The nodes of the selection octree are visible as well.

This kind of CPU-based selection works at interactive frame rates if only a few selections are used at the same time. The more point selections are used the slower the system becomes. This is due to the fact that new loaded points have to be tested against all existing selections. When more than 10 selections are used, the application can no longer work with an interactive frame rate. To use a point-selection method to highlight the annotated parts of the point cloud, it must be able to handle more selections. However, this is not the case with this CPU-base selection approach and another solution is required.

3.4 GPU-based Octree Creation

To overcome the CPU-based point-selection limitations a method that works on the GPU is proposed. To make this work, the selection octree must be ported to the GPU. The first proposed methods which bring octrees to the GPU could not rely on concepts such as CUDA. They used 3D textures as a workaround to represent the octree on the graphics card memory. The method proposed by Lefebver et al. [Lefebvre et al., 2005] encoded the octree in grids of 2x2x2 texels for each node to represent the data in it. All grids together are denoted as *indirection pool*. To

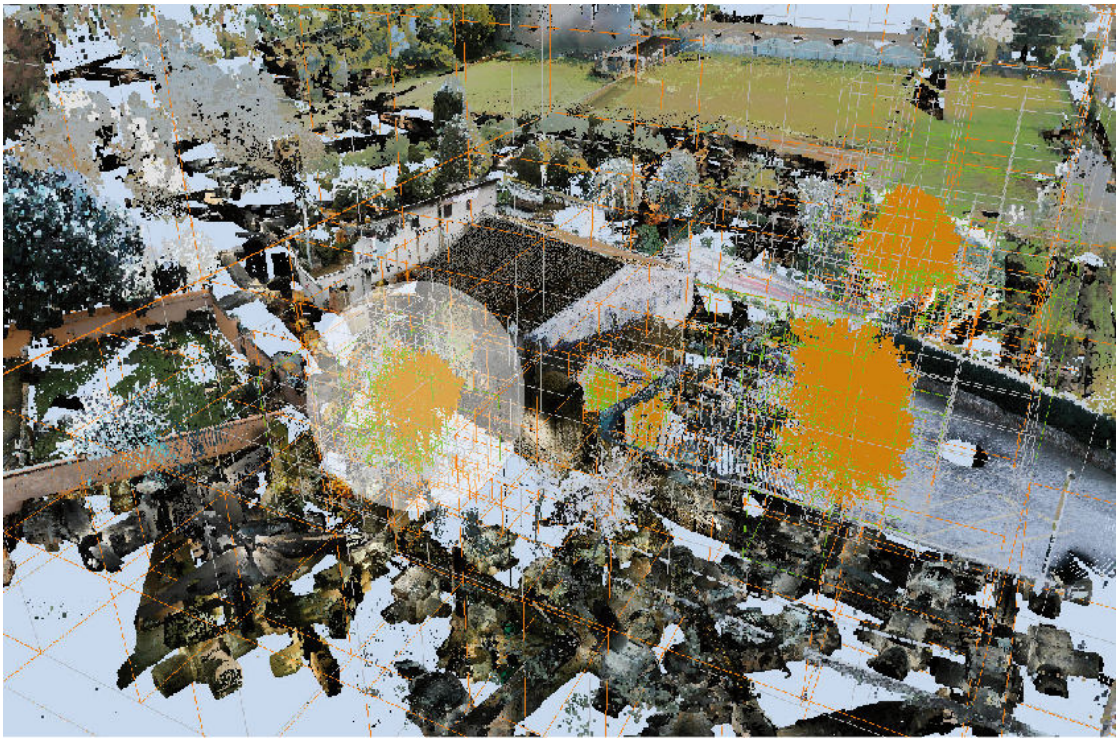


Figure 3.6: Point selection example of the solution proposed by Scheiblauer et al. The selected points are marked orange. The selection octree is visible as well. The following color coding i used: Octree nodes that hold only unselected nodes are colored gray, nodes that contain selected and unselected nodes are represented orange. The nodes which contain only selected octree nodes are represented in green. Image courtesy of Scheiblauer et al. [Scheiblauer and Wimmer, 2011].

specify the type of the octants the alpha value is used. An alpha value of 1 represents a leaf node which holds data. Alpha 0.5 is used for internal nodes, and alpha 0 is used to represent an empty node. The RGB information of the texel is either used to represent the data if it is a leaf or as an index to the child nodes if it is an internal node. The data in the octree can be accessed using the fragment shader.

With the introduction of general purpose methodologies, this workaround became obsolete. The main problem when implementing solutions that use octrees on parallel architectures such as the GPU is the fact that the existing serial solutions are not immediately parallelizable. The classical approach to build up an octree on the CPU is done by continuously inserting points into the octree. When inserted into the octree, the point subdivides the nodes of the octants such that the inserted points are distributed over the octree. This method does not work in parallel. Although it seems plausible to insert the points in parallel into the octree, this would result into memory conflicts as soon as different points try to change the same node of the octree [BéDorf

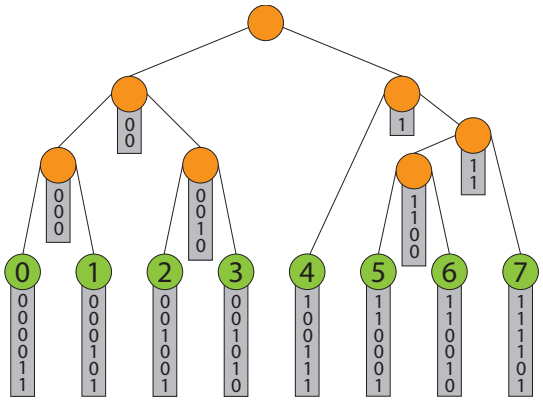


Figure 3.7: An example of a binary radix tree. Figure adapted from Karras [Karras, 2012].

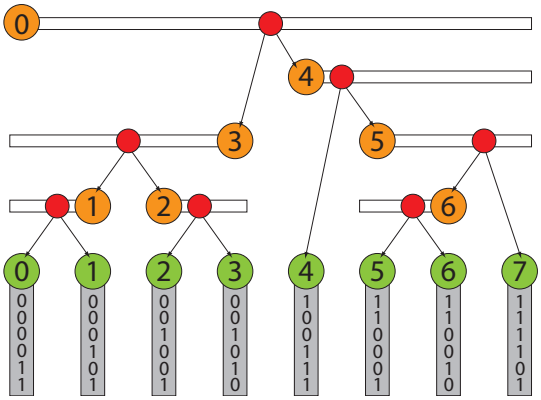


Figure 3.8: Internal representation of the binary radix tree from Figure 3.7. The orange nodes represent the internal nodes *I* and the green nodes represent the leaf nodes *L* The splitting position of each internal node is represented by a red dot. Figure adapted from Karras [Karras, 2012].

et al., 2012]. First versions came up using CUDA and space filling curves to represent and process octrees on the GPU in parallel [Ajmera et al., 2008].

The method proposed by BéDorf et al. [BéDorf et al., 2012] covered the problem of sorting the points the octree is created from. They proposed radix sort as it is currently the fastest sorting algorithm for parallel architectures supporting CUDA [Satish et al., 2009]. The disadvantage of this approach is that it needs one pass for each level of the octree.

This problem has been resolved by Karras introducing a parallel construction of the binary radix tree [Karras, 2012]. Similar to BéDorf et al., Karras first calculates the Morton code for each point that has to be inserted into the octree. Next, the computed Morton codes are sorted using radix sort. The crucial step for a fast parallel construction of the octree in CUDA presented by Karras is the usage of a binary radix tree. A radix tree stores the data in the leaf nodes of the tree. The internal nodes consist of the common prefixes of their corresponding child nodes. A special case of the radix tree is the *binary radix tree*. Each node holds exactly two children. Therefore, each binary radix tree with *n* leaf nodes holds exactly *n-1* internal nodes. See Figure 3.7 for an example of a binary radix tree.

Karras observed that each internal node can be computed independently from all the other nodes in parallel. To do so, the ordered Morton codes are stored in an array *L*, and the internal nodes are stored in a different array *I*. Therefore, the ID for *I* goes from *0* to *n-2*. The root of the tree is stored at position I_0 and it splits at the first difference found in the prefixes of the ordered Morton codes. All the other internal nodes represent a subpart of the whole list of Morton codes which share the same prefix. As can be seen in Figure 3.8, each internal node's ID *x*, represented by the orange dots, is also either the first or the last ID of the Morton codes it holds. Therefore,

the parallel algorithm can check the length of the common prefix with x and its right neighbor, and do the same with the left neighbor. As all Morton codes belonging to the same internal node of the radix tree have the same prefix length, this test can be used to decide whether the current internal node holds the Morton codes left or right of the ID x. Then the splitting position, represented as red dot in Figure 3.8, can be calculated in the same way as it is done in the case of the root. The left and right children are either another internal node with ID y when it is a left child or $y+1$ when it is a right child. However, the index stays the same if the child is a leaf, and it is marked as a leaf. Note that each consecutive group of 3 bits in a Morton code represents one level of the octree.

3.5 CUDA-based Radix Sort

As mentioned in the previous Section, a sorting algorithm is required to arrange the Morton codes in ascending order to build the octree with CUDA. The proposed method by Satish et al. [Satish et al., 2009] is based on the work by Zagha et al. [Zagha and Blelloch, 1991] which has been briefly described in Section 2.3.1. Since the access to global memory is expensive in CUDA, the method of Satish et al. makes heavy use of shared memory. Each block gets a number of keys assigned. These keys are sorted by repeating a bitwise *counting sort* four times. After the first step each block holds its key sorted according to the last 4 bits. As the next step the keys have to be sorted globally. Therefore, similar to the parallel sorting approach by Zagha et al., a histogram is calculated that can be used for global sorting. These steps are repeated with the next 4 bits until the keys are completely sorted.

3.6 Parallel Prefix Sum

While some tasks are very simple in the sequential case, where all the computation is done by one thread, they are not when they have to be ported to a parallel architecture. Such a task is the prefix sum or parallel prefix sum or *scan* in the parallel case [Blelloch, 1990].

The prefix sum is needed for the parallel radix sort to compute a global histogram out of the local histograms to put the data to the final position [Satish et al., 2009]. The binary radix tree construction method by Karras needs it to drop all duplicate entries [Karras, 2012] and it is also needed in this thesis in Section 5.4.2 to compute a compressed octree. It is therefore an important concept for this thesis and should provide a good running time, or it will slow down any algorithm constructed on top of it.

The prefix sum computes the sum of all elements in a list before the current element. It is distinguished between inclusive and exclusive prefix sum, depending on whether the current element is used for the computation of the sum [Harris et al., 2007]. See Table 3.1 for an example.

A sequential implementation is straight forward. The first input element can be written immediately into the output list. The next element is computed using the previous result plus the next element of the input list. This continues until all elements are summed up. This method does not work in the parallel because the current element needs the results of its previous elements.

Input	6	4	0	2	10	5	12	8
Output inclusive	6	10	10	12	22	27	39	47
Output exclusive	0	6	10	10	12	22	27	39

Table 3.1: Example for a prefix sum, distinguishing between inclusive and exclusive prefix sum.

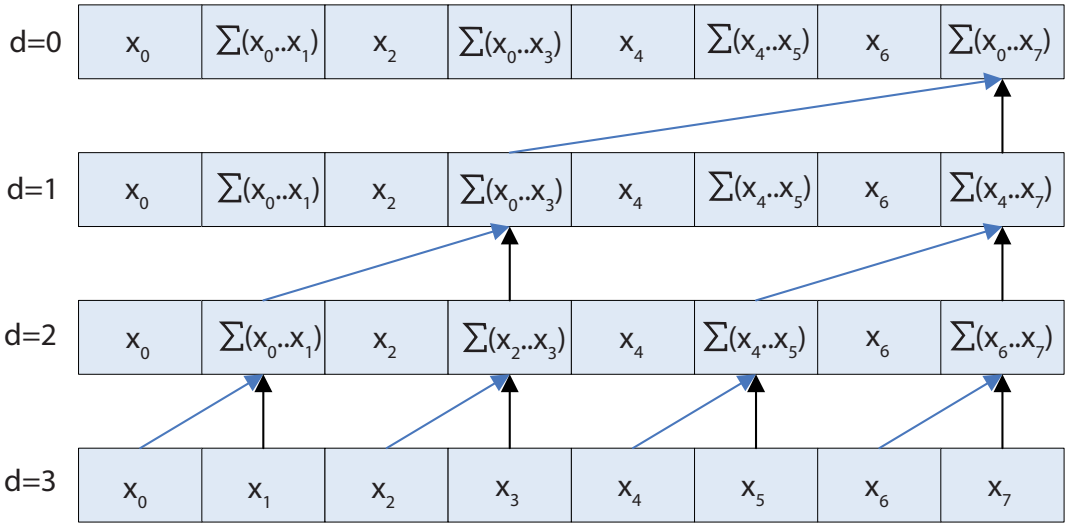


Figure 3.9: First phase of Harris parallel prefix sum algorithm. Image courtesy of Harris et al. [Harris et al., 2007].

An efficient implementation for CUDA has been provided by Harris et al. [Harris et al., 2007] adopting the parallel approach by Blelloch [Blelloch, 1990]. It uses a two step process to produce the final outcome. The first phase partially computes the sums of the input list from bottom up. The second phase uses the outcome of the first phase summing the items up from top down. Each thread can work on two numbers contained in the list to compute the output. The steps of each phase can be seen in Figure 3.9 and Figure 3.10 resulting in the final outcome. The computation of the prefix sum can be seen as the traversal of a balanced tree. It is not actually a tree. However, the concept helps to determine how the different elements of the list have to be accessed during the different steps to compute the prefix sum in parallel. The variable d presented in Figure 3.9 and Figure 3.10 represents the individual levels of the tree. A balanced tree with n leaf nodes results to a depth of $log_2(n)$ [Harris et al., 2007].

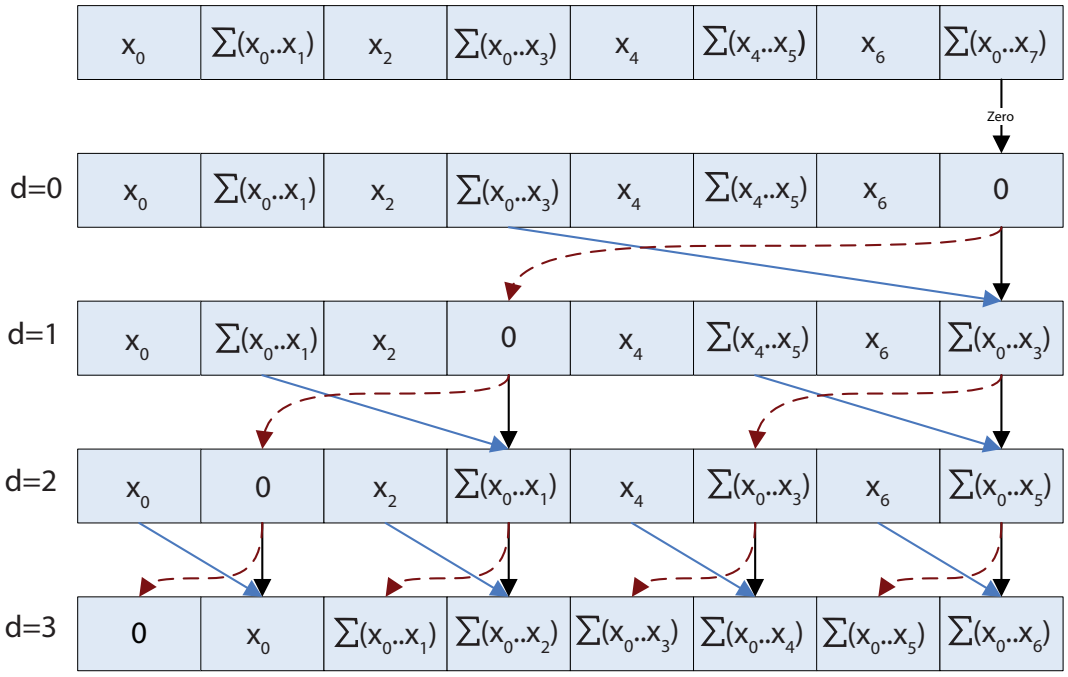


Figure 3.10: First phase of Harris parallel prefix sum algorithm. Image courtesy of Harris et al. [Harris et al., 2007].

CHAPTER 4

Annotations for Point Clouds

This chapter describes the annotation system which is able to annotate point clouds with additional information like text and images. Section 4.2 to Section 4.6 focus on the interface of the annotation system. As underlying point-cloud renderer for the annotation system, Scanopy is used.

Section 4.7 covers the details about a guidance system which uses graphs in combination with the annotations to route the user to points of interest in the point cloud.

Scanopy is a point-cloud renderer which has been developed at the *Institute of Computer Graphics and Algorithms* of the *Vienna University of Technology*. It is written in C++ using *Qt* for the user interface. For scene-graph management, *OpenSceneGraph* is used. For the low-level rendering of the point clouds, OpenGL in combination with Cg as shading language is used. The software is built in a modular fashion such that the core parts are independent of each other. Additional features not immediately necessary such as controlling the application using joysticks and 3D mice are realized with plug-ins.

Although a deep research for related work has been done, no system could be found which is able to annotate point clouds with additional information. Only one system providing a semantic annotation of point clouds has been found in the literature [Truong et al., 2012]. The system closest to the one described here has been provided by Yu et al. [Yu et al., 2011], annotating three-dimensional meshes for cultural heritage. Especially the highlighting of the annotated parts of the mesh and a brief textual hint of each annotation in the 3D space next to the model seem useful and are integrated in this system too.

4.1 Overview

Usually, there exists a large pool of information about a a scanned object. Different pieces of information are strongly correlated to various positions of the object. To present this information to the user in the context it has with respect to the object, the idea in this thesis is to place annotations next to the corresponding position in the point cloud. It is the aim of each annotation

to provide additional information about a given part of a point cloud. Therefore, each annotation can be enriched with images, PDFs, and text that can hold URLs to websites. The large benefit of this solution is that the user has the additional information right where it belongs. It is placed exactly at the object it is about and can immediately be retrieved from the annotation. Moreover, all information is well organized since it is attached to those parts of the object that it will provide with deeper information.

As already seen in Section 3.1, two concepts are important when working with 3D objects that should be annotated. First, a clear connection between the annotated part of the object and the annotation itself is required. Second, to maintain the overview within the 3D scene, not all information that is contained within an annotation can be placed in the scene. Therefore, a simplified representation of the annotation must be chosen which can be placed within the scene. When using point clouds represented in 3D, the connection can be established by placing each annotation next to the annotated part of the object it belongs to. However, the position of the annotation is not sufficient to link it with the object. Therefore, a connection between each annotation and the annotated points is established as will be described in Section 4.4. To provide a preview to the users what they can expect to find in each annotation, a meaningful title is displayed at the position of the annotation. Additionally, a 3D icon can be used to represent the annotation. Alternatively, an image attached to the annotation which copes the content of the annotation well can be used to represent the annotation in the 3D scene.

When a scene contains many annotations and those are all visible on the screen at the same time, getting an overview is difficult. To resolve this issue, several methods are used. First, each annotation can be set visible or invisible by the user. Annotations that belong together can be grouped together. This makes it possible to manually set all annotations of a group visible or invisible. The last method allows assigning a priority to each annotation. This priority determines within which distance to the viewer annotations are visible. Annotations with high priority are also visible from further away, while annotations with low priority will not be visible at the same distance. This helps to visualize only those annotations that are relevant for the current viewpoint.

In complex point clouds such as catacombs it is difficult to navigate through all the corridors to find points of interest. Such points of interest can be represented by annotations. The annotations are combined with a graph to build a guidance system. The graph is constructed such that it represents all possible paths in the point cloud (for example all corridors in a catacomb). Now it is possible to direct the user from any node of the graph to another node inside the same graph, and it can be used to guide the user to any annotation inside the scene. For this to work, the graph is constructed in such a way that each annotation is linked to its closest node in the graph. This must be considered during the construction of the graph. The graph must be constructed such that it passes each annotation, and for each annotation must exist a node in the graph that is close to it.

4.2 Rendering

Everything that is rendered in Scanopy is added into a scene graph managed by OpenSceneGraph. Therefore, to visualize the annotations in the scene, each annotation is represented as a

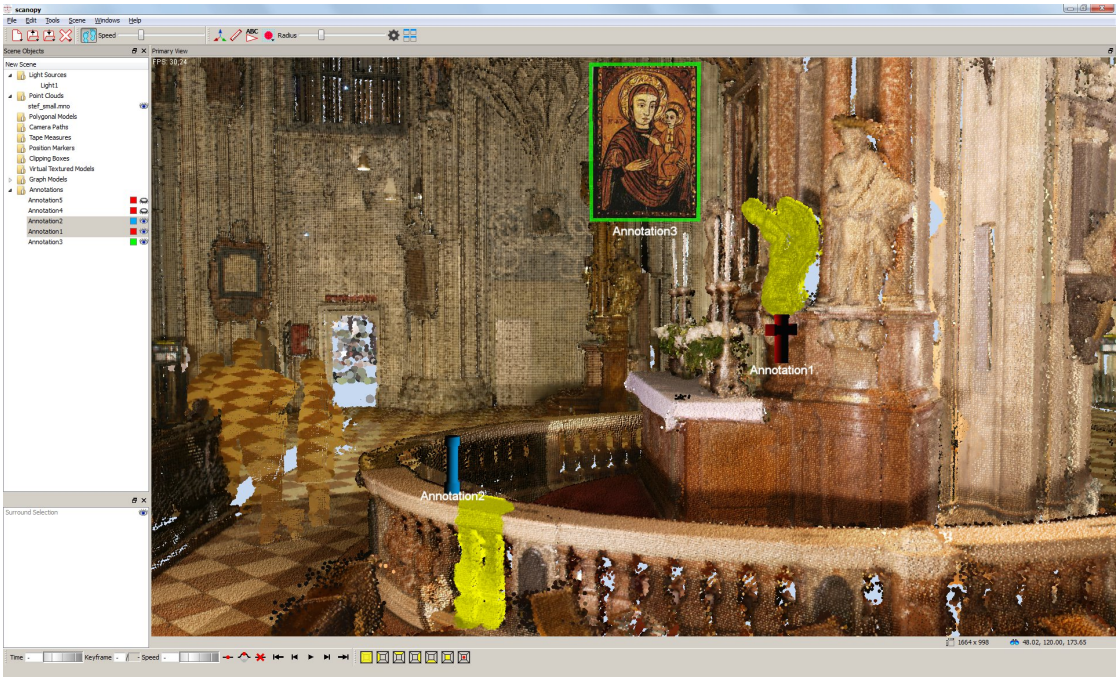


Figure 4.1: Different annotations combined with a point cloud.

child of this scene graph. To provide a hint to the user what each annotation is about, the title of the annotation is displayed at the position of the annotation in the scene. Additionally, each annotation is represented by a three-dimensional icon above the title. The icon can be freely chosen as long as it can be imported as a mesh. The following data types are currently supported: OBJ, OSG, 3DS. Considering cultural heritage as example, each group of annotations can have a different icon. For example, all inscriptions have the same icon, paintings and special architectural structures are represented with a different icon. This allows the viewer to immediately distinguish the annotations without reading the title of the annotation. The title can be used to distinguish between the elements within the same group. As an alternative, the icon can be changed with a billboard that contains one of the images attached to the annotation. If an annotation is represented in the scene with a billboard of the most meaningful image attached to the annotation, this helps the viewer guess what the annotation is about. Since the caption and the billboard are always aligned to the viewer, the representation of the annotation in the scene can be seen from any viewing position. This is also supported by the usage of 3D icons because they can represent the annotation from any viewpoint.

When working with 3D graphics, the perspective projection lets things appear bigger when the camera is moved closer and lets them appear smaller when it is moved further away. This causes two problems when working with annotations in a 3D scene. If the annotations becoming too big, more and more of the point cloud is covered by the annotation. Therefore, it is difficult to see the annotation in the context it appears to the point cloud because the point cloud is covered

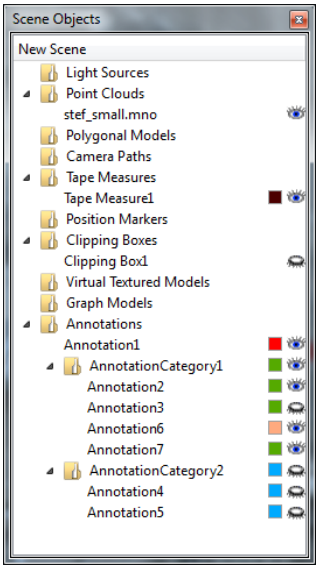


Figure 4.2: The scene explorer holds all objects of the current scene. Annotations can be grouped by using different tabs.

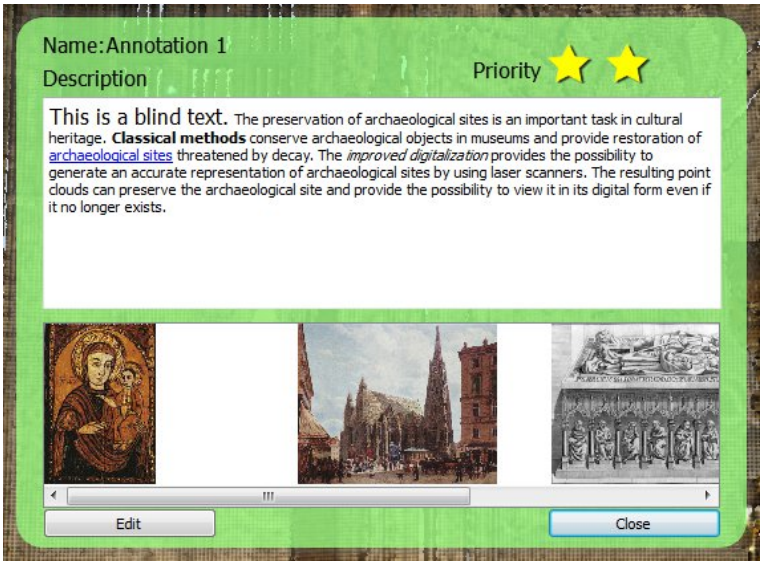


Figure 4.3: The user interface represents the information stored in an annotation. The upper half of the user interface holds titel, description and the priority of the annotation. The lower half holds all images and documents attached to the annotation.

by the annotation. The second problem results in barely visible or even invisible annotations when the viewer is too far away from the annotation. To prevent annotations from become too large or too small when the user moves closer or farther away from the annotation, the projective scaling is limited within a small range. This supports the natural behavior of objects becoming larger when moving closer but still restricting them from becoming too big or too small.

The CPU-based point-selection method uses the vertex shader to change the color of all selected points. This helps to distinguish between selected and non selected points in the point cloud. Once the selection of all points from the point cloud belonging to an annotation is completed, this method will also be reused to show the connection between point cloud and annotation. The final annotation system integrated in Scanopy can be seen in Figure 4.1.

4.3 User Interface

In Scanopy, all objects rendered in the scene are also available in a tree-view called scene explorer, represented by an entry with their names. This also holds true for all the available annotations that are inserted into the scene. Figure 4.2 shows the scene explorer with several scene elements. The data of each annotation can be visited by double-clicking either on the corresponding item in the scene explorer or by double-clicking on the annotation in the scene. The visualization of the data stored inside an annotation is done using Qt widgets.

As can be seen in Figure 4.3, each annotation can be enriched with several files. Currently, images and PDFs are supported. The attached files are visualized in a window located at the bottom of each annotation. Attached documents require a title, and images can be further annotated by adding a description. To allow further insight into the attached files, the images are represented as thumbnails. This is similar to the DECHO framework proposed by Aliaga at al [Aliaga et al., 2011], which only provides images as data type for additional files. This is a problem when too many files are attached to the annotation. As soon as there are more files attached than can fit on one page of the window, the overview of the data is no longer provided. Additionally, it would lead to long scrolling periods. Therefore, the view can be changed to a list when an annotation contains many files. This results in smaller items represented each by a small icon and the name of the attached file. To improve the overview of the attached data files, they can be sorted according to their names or their category. As can be seen in Figure 4.3, the description widget supports different text formatting types such as bold and italic plus different font sizes. This creates the possibility to emphasize certain parts of the description. This can be especially useful when the description is a long text. To link other data besides images and PDFs, the widget further enables the declaration of hyperlinks. Both PDFs and web pages can be linked.

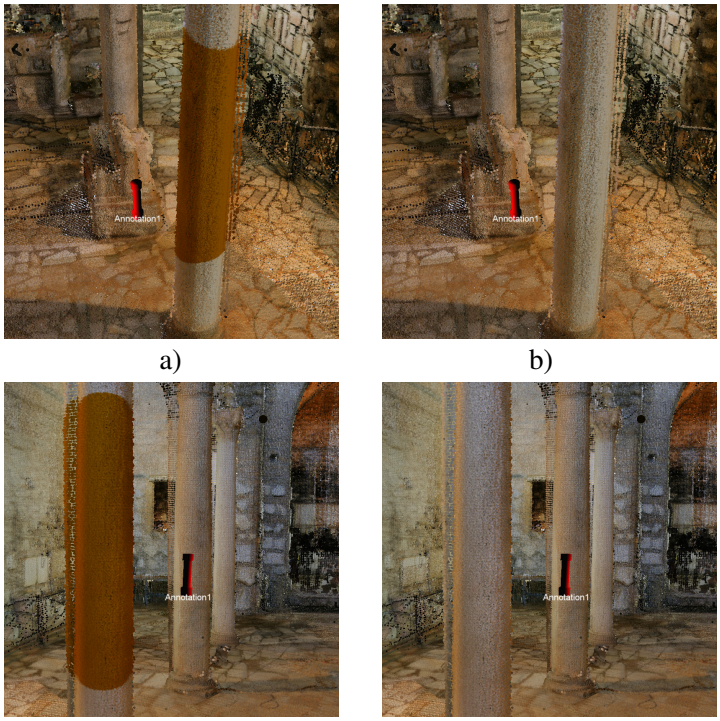
All attached images and web pages can be opened directly inside the annotation without the need of an additional application. The provided image viewer allows zooming and panning for images larger than the provided viewer window.

When an annotation is created, it is placed in the origin of the scene. To place an annotation inside the 3D scene, three options exist. First, it can be numerically defined using the corresponding fields in the properties dialog. The second option can be called by pressing the left mouse button inside the scene. By extracting the depth value of the current cursor position, the 3D position of the annotation can be calculated. The benefit of this solution is that it places the annotations immediately next to the currently visible part of the point cloud. The third option uses manipulators as they are available in 3D modeling solutions. Those manipulators can be dragged to move the annotation.

4.4 Linking Annotations to Parts of a Point Cloud

A visual hint is required when using annotations, otherwise it is not clear which part of the object is annotated. When working in three-dimensional space as it is done when working with point clouds, a first hint can be the location of the annotation. However, there are other issues to consider as well. The geometric structure of the annotated part can have different shapes. This makes it difficult to guess which parts of the object belong to the annotation and which not, by using the position only. Additionally, whether using the position alone helps to guess which parts of the object belong to the annotation heavily depends on the viewing direction from which the annotation is viewed. This can be seen in Figure 4.4, the same annotation is presented from two different views. It is difficult to determine which part of the point cloud is annotated when no hint on the point cloud is available.

Simple solutions such as polyhedrons positioned around the corresponding points of an annotation are either not fine grained enough to cover any possible case of possible shapes that



a) b)

c) d)

Figure 4.4: The point-marking method helps to detect which part of the point cloud belongs to the annotation. The left column (images a and c) shows the annotation with selected points. The right column (images b and d) shows the annotation without selected points. It is difficult to see which parts of the point cloud are annotated without a clear link between the annotation and the point cloud.

might be annotated when working with point clouds or result in complex geometry difficult to create. The most natural decision besides using a volume to confine the region of a point cloud is to highlight all individual points belonging to this region. This highlighting approach results in an accurate representation of the annotated part of the point cloud. This requires a method to select those points.

For now it is considered that a point-selection method exists to mark all points that belong to a given annotation. At this point, it is ignored that the CPU-based selection is not suited for the usage with annotations due to performance issues because the linking process conceptually works also with the CPU-based selection method. The point-selection method is used to create a selection octree for each annotation, which can be used to highlight all points within the selection to provide a visual link between point cloud and annotation.

The highlighting itself is done by changing the color of all points that belong to the selected regions of a selection octree. Exchanging the point color to a predefined color which is the same for all selected points makes it impossible to see the details of the selected part of the point cloud. An example can be seen in Figure 4.5 b), only the surroundings of the selected object are

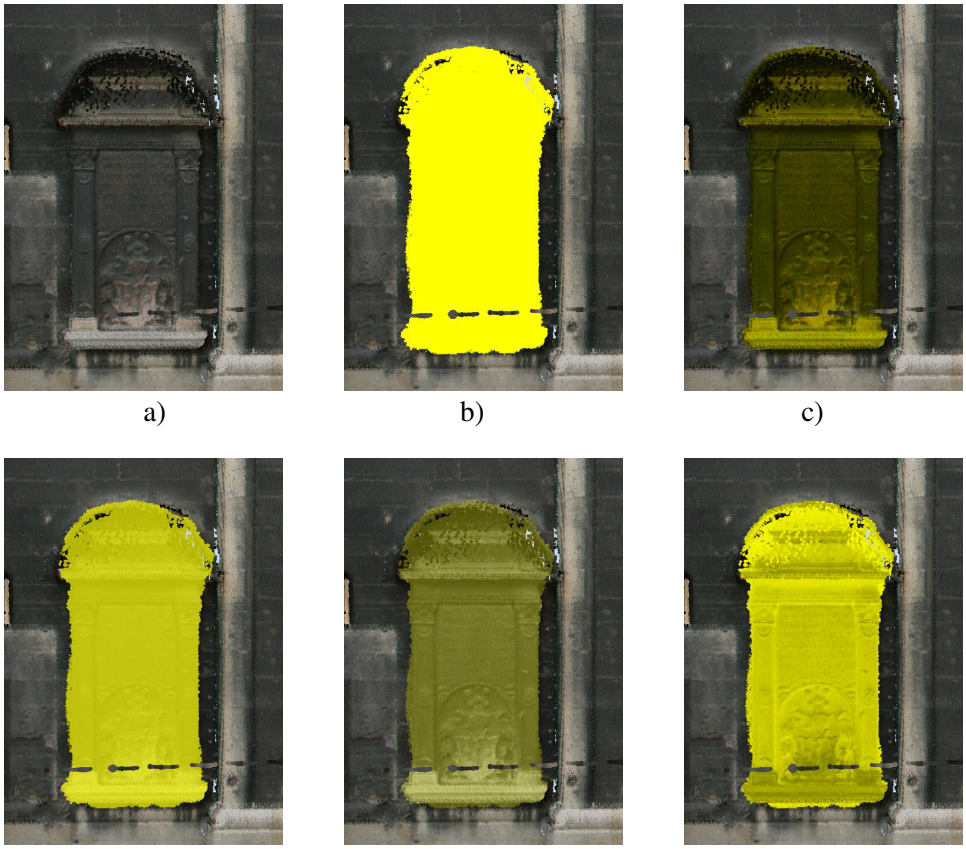


Figure 4.5: A section of a point cloud rendered with Scanopy is presented in a). As can be seen in b), a simple replacement of the color of all selected points makes it impossible to see the structure of the selected part. A multiplication of the color of each point with the selection color can resolve this issue. As can be seen in c), this does not work well with dark-colored points. An alternative blending approach is presented in d) and e). Depending on the used blending factor, the structures of the selection are not clearly visible (d)), or the selection color is not represented well (e)). The proposed solution is presented in f).

visible, all the other details of the object are lost. A first solution to change the color of all points of a selected object is to multiply the color of each point with a predefined selection color. As can be seen in Figure 4.5 c), this works well for bright colors, but it is not able to highlight dark areas of the selected object at all. Moreover, it does not work well with points whose color are saturated. The multiplication of complementary colors causes problems as well. To resolve these issues, the color of each selected point is converted to a gray value according to Formula 4.1. It computes the luma for the YCbCr color space out of the RGB color space [Poynton, 2012].

$$grayvalue = red * 0.299 + green * 0.587 + blue * 0.114 \tag{4.1}$$

```
1 grayColor = Vector4 (grayvalue, grayvalue, grayvalue, 1.0);
2
3 if (grayvalue < 0.5) {
4    finalColor = Vector4 (1-grayvalue, 1-grayvalue, 1-grayvalue, 1.0)*
         selectionColor;
5 }
6 else {
7   finalColor = grayColor* selectionColor;
8 }
```

Listing 4.1: Computation of the final color for a highlighted point.

If the resulting gray value is too dark, it is negated. This method is able to preserve the structure of the selected region for variable luminance regions. The resulting gray value is multiplied by the selection color. The complete method can be seen in Listing 4.1. This method produces for each point of the selected object a more or less saturated color. Overall, the whole selected object is colored with the same color and the details of the object itself are still visible, as can be seen in Figure 4.5 f). On the other hand, alternative approaches such as blending or weighted average are not able to provide the same result. Those results are presented in Figure 4.5 d) and e). Either visible details get lost within the selected region, or the selection color is not clearly visible.

4.5 Data Management

Before bringing annotations into Scanopy, all necessary information is stored in XML text files and proprietary binary files. For example, the objects in a scene, like light sources, tape measures, and position markers, are stored in an XML file called *XML scene file*. Also, references to the loaded point clouds and the created point selection octrees are stored in this XML scene file. The point cloud and the selection octrees are stored in a binary format.

With the introduction of annotations, the XML scene file could have been extended to adopt it to the needs required to store annotations as well. However, the annotations are rather nested with different types of galleries holding images and PDFs. Different 3D meshes are used to represent the annotations in the 3D scene. Those meshes should be available for the annotations in all scenes. The possibility to query for certain parts of the annotations would be desirable as well. This can be especially useful for advanced searching inside all the available annotations of a scene. Therefore, a global storage possibility, which is able to provide all those features, is required.

We use a database to store all the required information for each annotation. To combine all annotations belonging to a specific scene, a single entry in the XML scene file is used. The same element is also present in the database, where it groups all the annotations of a scene together. As briefly mentioned, the point selections used to manipulate the point clouds of a scene are stored on hard disk. This is no longer the case for the point selections that are used to represent the link between the point clouds and the annotations. Those selections are represented inside

the database and can be retrieved together with the corresponding annotation. The details how the selections are stored in the database are covered in Section 5.6.

As database management system (DBMS), *PostgreSQL* has been chosen for several reasons. First, it is besides *mySQL* the only established free DBMS. Compared to mySQL it has a less restrictive license. While mySQL is released under the *GNU General Public License* (GPL), which requires you to provide the developed source code again under the GPL, PostgreSQL is published under its own license similar to the BSD or MIT license.

Therefore, it is up to the developer to decide what should happen to the source code, and a release under the GPL is not necessary. There were no performance considerations when choosing the appropriate DBMS because the information retrieval from the database and storing the data to the database are no time-critical tasks in the application. Moreover, there does not exist a concurrent access to the data since the database is currently stored locally on each machine running Scanopy.

To allow a simple exchange of the used DBMS, the database access is controlled with Qt, which provides plug-ins to several DBMS such as mySQL, Oracle, SQLite, DB2, and PostgreSQL (as used in this thesis). Each object required for the annotation process is available as table in the database and also in the application. For each DBMS, the DatabaseConnection class must be extended, and the read and write methods to the database must be overridden.

Scanopy should be able to handle large amounts of annotations combined with attached images and PDFs. It is assumed that the user is only investigating a few annotations at the same time in detail compared to all annotations available in the current scene. Therefore, the data of each annotation is loaded on demand from the hard disk once it has been opened by the user. This helps to keep the memory utilization of Scanopy low and makes the number of available annotations less dependent from the available main memory.

As loading images attached to the annotation from disk is time consuming, especially if several high-resolution images are used, loading images and creating the appropriate thumbnails out of it is controlled by a separate thread. This avoids the blocking of the application. The same threaded loading is also done when an image is specified as representation of the annotation in the 3D scene and it has to be loaded from disk.

4.6 Priority-Dependent Visualization

With an increased number of annotations per scene, the system requires some filtering method such that the user is not lost within all the annotations. The most basic method is to keep the annotation minimized until the viewer claims interest about it [Aliaga et al., 2011, Sonnet et al., 2004]. This is not sufficient when the amount of annotations further increases. This is especially the case when the user is viewing the scene from further away to gain an overview of the whole point cloud. Because in this case it is difficult to get an overview when hundreds of annotations are visible at the same time. Other proposed solutions focus on combining different annotations together such that they can be set visible depending on the group they belong to [Díaz et al., 2011, Kadobayashi et al., 2005]. As the scene explorer in Scanopy gives the possibility to set the visibility of each item individually by pressing an appropriate icon, this concept is applied for the annotations as well. To include a group-like structure into the explorer, it is extended such

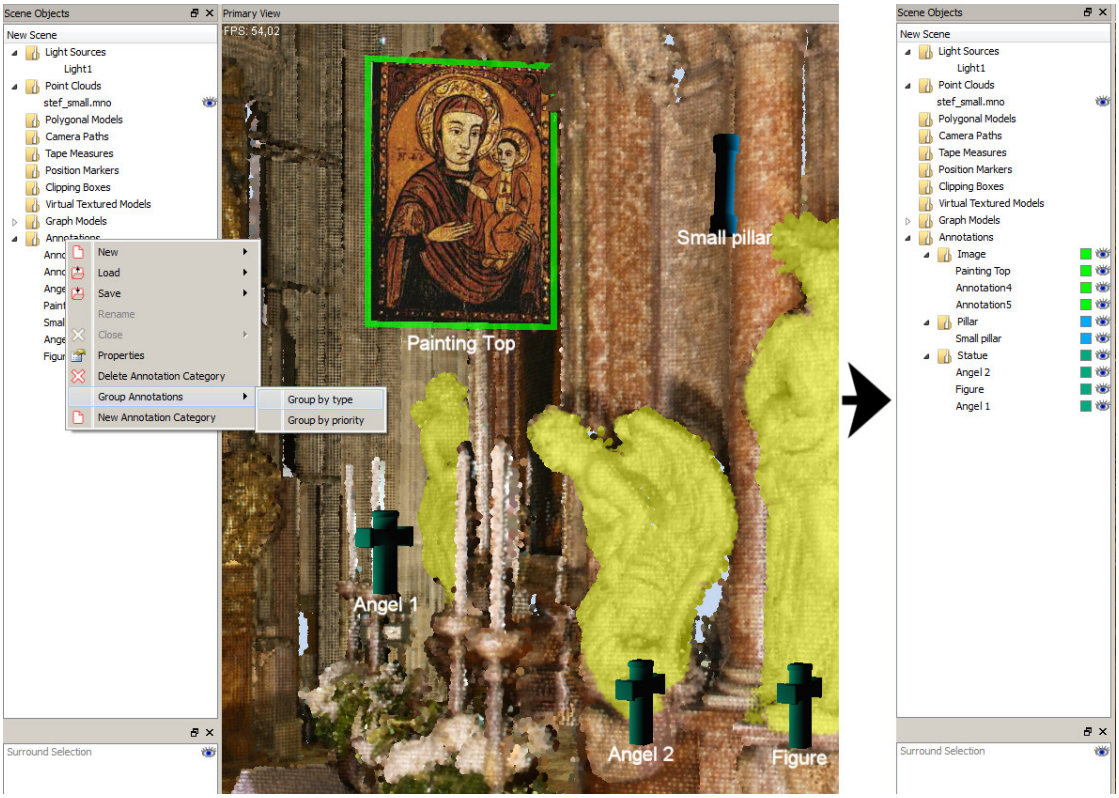


Figure 4.6: The automated annotation grouping in the explorer demonstrated on the example of the *type* property of the annotations.

that the tab holding the annotations can represent additional tabs, which group the annotations together. Several annotations grouped in different tabs can be seen on the right side of Figure 4.6. It is possible to automatically group all annotations of a scene. Annotations that share one property are grouped together. Currently, the type of a annotation or its priority can be used as grouping criterion. The type is a user-defined field which sets also the 3D icon that represents the annotation in the 3D scene. It can be seen as a way to categorize all annotations of a scene. The priority value is used for the priority-dependent visualization described below in this section. Figure 4.6 shows the automated grouping and the resulting annotation group structure. Once created, the automated grouping can be changed manually. Alternatively, the grouping can be done completely by hand. In the scene explorer, annotations can be dragged over an annotation group. As can be seen in Figure 4.7, the grouping process can also be done immediately in the 3D scene. Each annotation can be selected by right-clicking on it. Several annotations can be selected by holding the control key pressed while the annotations are selected with the mouse. The titles of all selected annotations are highlighted in red to show to the user which annotations are currently selected. Once selected in the 3D scene, annotations can be assigned to an existing group chosen from the list of available groups.

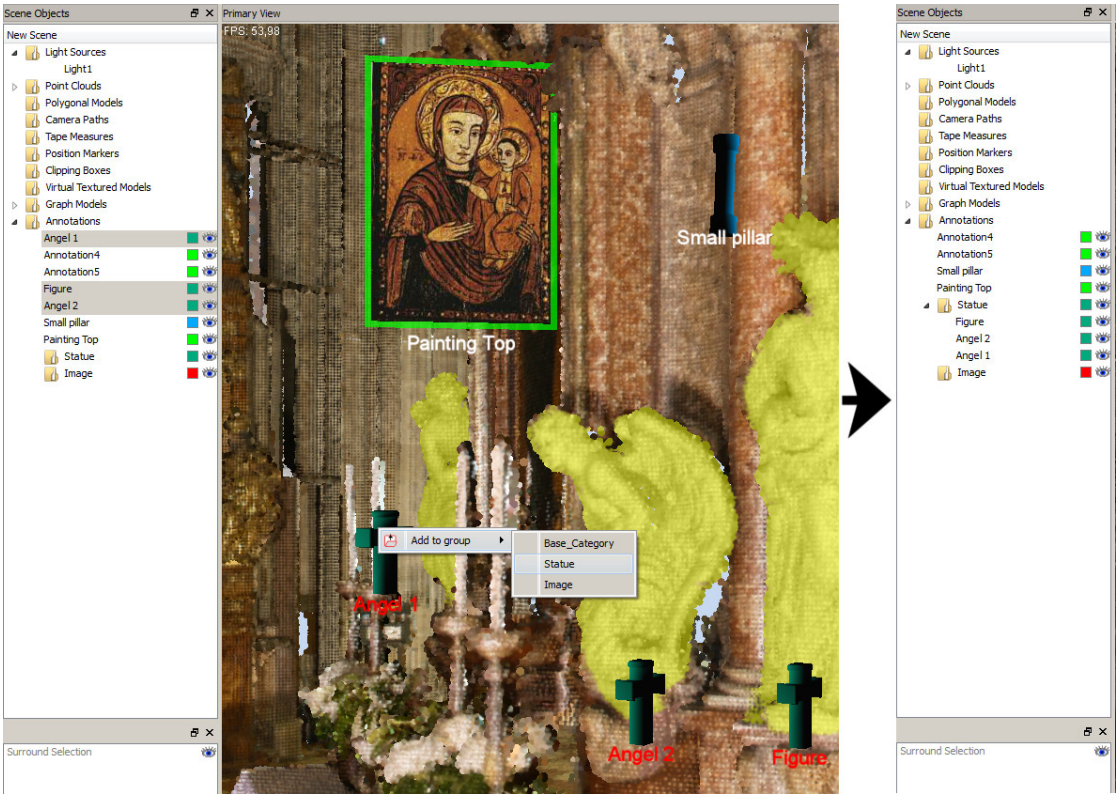


Figure 4.7: The annotation grouping in the 3D scene. All selected annotations can be added to an existing annotation group.

By toggling the visibility of a group tab, all corresponding annotations change their visibility as well. To provide a further differentiation between the different groups, a color can be assigned for each group. This color assignment can be refined by assigning annotations different colors independent of the group they belong to. Coloring the mesh icons used to represent the annotations is straight forward. To provide this feature also for annotations using an image as representation in the scene, a border with the appropriate color is drawn around the image. Of course drag and drop is supported to allow moving the annotations between the different groups easily.

The usage of groups depends only on the user defining what kind of annotations should be visible by clustering them. Another more natural approach to gain control on what is visible can be the usage of a level-of-detail system. This is a type of priority-dependent visualization, which is common in *geographic information systems* (GIS). When looking on a digital map zoomed out, not all the data available can be made visible because there is not enough space for it. Therefore, only the data considered the most important is visualized. When zooming into the map, more details become visible as there is more space available to place these details. *GoogleMaps* can be seen an example of such a system.

This idea can be used as well in Scanopy to decide between important and less important annotations. The importance and therefore also the visibility of each annotation can be controlled by the user. This is done by manually assigning a priority between 1 and 3 to each annotation, where 3 is the highest priority. Additional classes could be added in the future to allow an even more fine-grained separation. Annotations with the highest priority will always be visible in the scene. The other annotations are only visible when the distance between the camera position and the annotation does not lie above a predefined threshold. This threshold varies between the different priority classes and therefore, also the visibility of annotations with different priorities is not the same. This approach has another advantage. The visibility of each annotation can be controlled such that it is only visible when the annotated part of the point cloud itself is visible as well because the user is close enough. If the annotations would always be visible even if the relevant part of the point cloud is still far away, it would not be clear to the user what the annotation is about.

Since the system should be able to handle a large amount of annotations, testing the visibility for each annotation in each rendered frame is too expensive and also not necessary. Since an annotation can only change its visibility state when the view position or viewing direction changes, a *CullCallback* from OpenSceneGraph is used to change the visibility according to the annotation's priority. To avoid annotations from becoming immediately visible, which would result in unpleasant popping effects, they are faded in and out within a given range. The closer the user comes to the annotation the less transparent it gets, until a given distance to the annotation is reached and it becomes completely opaque.

4.7 Annotation-Supported Guidance System

Point clouds from cultural heritage sites can become rather complex. One example of such a site are catacombs. They can consist of hundreds of corridors underneath the surface of the earth, often connected over several floors. Due to their construction, they have a lot in common with a maze, and orientation can be complex especially for non-experts in the field. As one of the key purposes of annotations is to add information to important areas, it is assumed that annotations are only placed next to locations that are of special interest. Therefore, annotations can be used to guide users to interesting points in a point cloud.

To guide the user through the catacombs by using annotations, it is also required to know how the different corridors of the catacomb are connected with each other. This is done by constructing a graph through the catacomb where the edges are placed through the corridors. At the crossings of the corridors, the edges are connected. This graph can be used to guide the user between any two nodes by using a path finding algorithm such as the Dijkstra's algorithm or the A* search algorithm. The graph and the annotations can be combined to guide the user along the graph to any annotation if the catacomb with all its corridors is represented as graph. Moreover, it is required that the graph passes each annotation such that the user can be guided to it.

As the graph should be used as guidance system combined with all the available annotations, it is important to know which annotations can be reached over the connected edges of a node. Each node holds a search data structure which maps all available annotations to the connected edges of the node which will lead one step closer to the corresponding annotation.

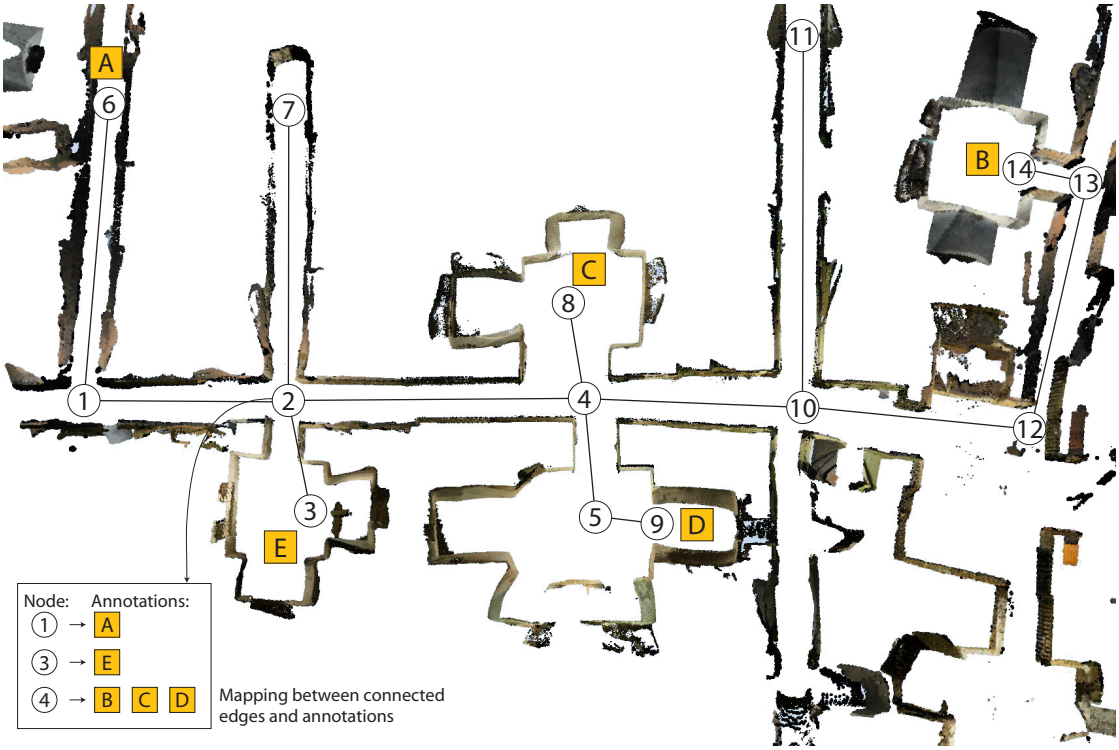


Figure 4.8: This is a sample mapping between nodes and annotations used to guide a viewer from any point along the graph to each annotation of interest. It shows which node has to be traversed next to reach a given annotation if the user is currently at node 2.

This allows the graph to be traversed from any given point to any available annotation along the shortest possible path. Before the user can be guided to an annotation, this mapping between edges and annotations has to be computed. The algorithm for this computation is divided into two parts. The entry point is Algorithm 4.1. The closest graph node is calculated for each annotation. This node will be used as target node for the annotation when traversing the graph. Next, any node directly connected over an edge to a target node is updated by linking the corresponding edge with the annotation that can be reached traversing it.

Figure 4.8 shows a sample mapping between a graph and a number of annotations. The mapping between the edges connected to node 2 and the annotations that can be reached over these edges is presented as example. All the other nodes have such a mapping as well. This allows to show the user at any node which annotations can be reached if the traversal on the graph is proceeded along the corresponding edge.

Once the mapping of the neighbor nodes is completed, the second part of the algorithm is called. This is a recursive function noted in Algorithm 4.2. It contains the principles of Dijkstra's algorithm. While Dijkstra's algorithm originally uses one target, Algorithm 4.2 can handle multiple targets. The number of targets depends on the number of annotations available in

```
   input  : A list of annotations annotations of size n
   output: A graph with all edges linked to the reachable annotations
 1 if not IsNavigatorUpToDate then
 2 |  ClearInfoPoints();
   |  /* find for each annotation a the graph node
   |     current_node[a] which is closest to it                    */
 3 |  foreach annotation a in annotations do
 4 |  |  current_node[a] ← GetClosestGraphNodeFromPosition(a);
 5 |  end
   |  /* set for each edge e connected to current_node[i] which
   |     is closest to annotation a that a can be reached over
   |     i                                                         */
 6 |  foreach node i in current_node do
 7 |  |  foreach edge e connected to i do
 8 |  |  |  list_of_nodes ← e->Annotation[i];
 9 |  |  |  attach (list_of_nodes, i ) ;
10 |  |  |  e->Annotation[i] ← list_of_nodes;
11 |  |  end
12 |  end
13 |  SetAnnotationsPerEdge ( graphRoot, current_node );
14 |  SetNavigatorUpToDate () ;
15 end
```

Algorithm 4.1: BuildUpGraphAnnotationInformation. Note that *list_of_nodes* is a map data type (such as std::map) in C++ mapping annotations to nodes.

the scene. A target is a node of the graph which is closest to the position of a specific annotation. For all nodes in the graph, it computes the adjacent node that must be traversed to reach a specific target. The execution of the Algorithm starts from the root of the graph and performs the algorithm for each node.

Similar to Algorithm 4.1 from Line 6 onwards, it checks for each node whether an adjacent node is able to reach an annotation. If this is the case, it is stored in a map that the annotation is reachable if this adjacent node is traversed. This function is called recursively for all connected edges of a node. Before the function is called for an edge, it is marked as visited to avoid visiting any node twice.

The first mapping step works only for all nodes which are traversed after a target node of a specific annotation. Therefore, a second mapping step is necessary after the recursive call returns to update all nodes before the target nodes. At the end of the algorithm, a flag marks that the mapping is up to date. The mapping has to be updated by the algorithm whenever the position of an annotation is changed or deleted. The same holds when something is changed at the graph.

The graph of the guidance system is visualized with OpenSceneGraph, nodes are represented as spheres and the edges are represented as cylinders. The components are rendered

```
1  Function SetAnnotationsPerEdge( GraphNode: node, map: current_node ) : void is
2      foreach edge e1 connected to node do
3          foreach edge e2 connected to e1 do
4              if node not part of e2 then
5                  annotations ← node->Annotation[e1] ;
6                  connected_annotations ← e1->Annotation[e2] ;
7                  foreach annotation a in annotations do
8                      attach (connected_annotations, a ) ;
9                  end
10                 node->Annotation[e1] ← connected_annotations ;
11             end
12         end
13     end
14     foreach edge e connected to node do
15         if e is not visited then
16             e ← visited ;
17             SetAnnotationsPerEdge(e, current_node) ;
18         end
19     end
20     foreach edge e1 connected to node do
21         foreach edge e2 connected to e1 do
22             if node not part of e2 then
23                 annotations ← node->Annotation[e1] ;
24                 connected_annotations ← e1->Annotation[e2] ;
25                 foreach annotation a in annotations do
26                     attach (connected_annotations, a ) ;
27                 end
28                 node->Annotation[e1] ← connected_annotations ;
29             end
30         end
31     end
32 end
```

Algorithm 4.2: SetAnnotationsPerEdge. Similar to Algorithm 4.1 *annotations* and *connected_annotations* are both data types mapping annotations to nodes.

semi-transparently as the rendered elements are only provided as a hint to the user. The semi-transparent rendering further provides a better visibility of the point cloud since the view to the point cloud is not blocked by opaque parts of the graph.

Once the construction of the graph is completed and the mapping between each node and the annotations is completed according to the previously described method, the graph can be traversed. The traversal can start from any point in the graph. When the user starts the command

Figure 4.9: Different annotations combined with a point cloud.

to traverse the graph, the camera is moved to the closest node of the graph compared to the current camera position.

While the user moves along an edge, no guidance information is displayed, as the only two possibilities to traverse the edge are forwards or backwards. This also holds true for any node connected with two edges. When a node is reached with at least three edges connecting to it, which resembles a crossing: the previously computed information can be used to display the available annotations that can be reached by following one of the edges. The possible directions are displayed by arrow icons at the corners of the screen such that the user knows in which direction it is possible to continue the traversal along the graph. In addition to each arrow, the annotations are displayed underneath each arrow to provide a hint to the user which annotations are reachable following this direction. The information displayed on a crossing can be seen in Figure 4.9. As the available screen space is limited, only a given amount of annotations can be displayed for any direction. On a screen with a resolution of 1920x1080, four annotations fit per direction. Therefore, a selection process is needed when more annotations are reachable over a given direction. The proposed selection process uses the remaining distance from the current node to the annotations and the priority of the annotations. For each of the three priority categories, the closest annotations is displayed. If the screen is large enough to provide space for additional annotations, the next closest annotation is displayed. To support the user with the decision procedure which annotation shall be visited next, relevant information is displayed.

This consists of the title of the annotation and the image chosen as icon representing the annotation in the scene. If no icon was chosen by the user, the first icon is used, or a dummy image is used if none exists. As can be seen in Figure 4.9, the priority of each annotation is displayed as stars, and the distance along the path to the annotation is visualized to get an idea how far the individual annotations are separated among each other. Also, each annotation can be opened by clicking at the area occupied by the rendered hint. This can be useful if the hint alone is not sufficient to make a decision where to go next.

CHAPTER 5

Point Selection with CUDA

This chapter gives an extended description of the point-selection approach implemented using CUDA. The reason for a CUDA-based point selection has already been mentioned in Section 3.3, and CUDA itself has been introduced in Section 2.4.1. This chapter will cover the details on designing a point-selection method. It is combined with a point-marking method to visualize the selection by highlighting the points to provide links between the annotations and the point cloud.

5.1 Overview

To highlight all points belonging to an annotation, the already existing CPU-based method is adapted to work with CUDA. Before the points can be highlighted, it is required to decide which points should be part of the highlighted area. This is done by selecting points with a three-dimensional brush such as a sphere or a cube. With each brushstroke, every point that lies within the volume of the brush is selected, and the color of the point is changed. The result of this selection procedure is a set of highlighted points.

Due to the out-of-core point-cloud rendering, only those points that are currently needed to render the point cloud are loaded from hard disk. Therefore, only those points can be selected with the brush. Whenever the user changes his viewpoint, other points are required to render the point cloud from the changed viewpoint. Those points are loaded from the hard disk. Since not all points are necessarily loaded during a selection, a selection volume is created out of the currently loaded points to solve this problem. The CPU-based point-selection method already available in Scanopy uses selection octrees (Section 3.3) as representations for the volumes. Each node of the selection octree is recursively subdivided until each node of a selection octree contains only selected or unselected nodes. When points are loaded from the hard disk, the selection octree can be used to test whether the loaded points belong to the selected regions of the selection octree and set their color according to the result of this test. Due to the good performance results for CUDA-based octrees reported in the literature [Karras, 2012], the concept of selection octrees has been chosen to be ported to CUDA and to be used for point selections.

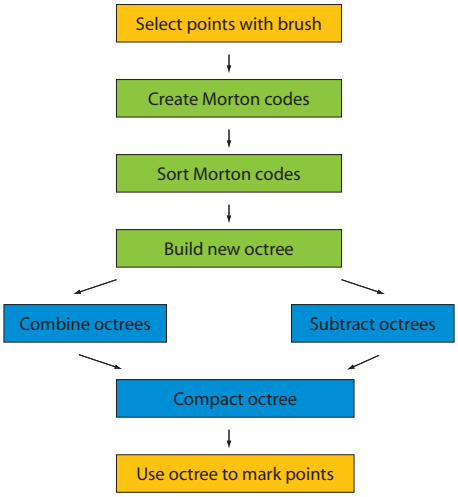


Figure 5.1: All basic construction steps required to create a selection octree with CUDA. The green parts are adapted from Karras method [Karras, 2012].

The concept is similar to the existing CPU-based selection algorithm and visualized in Figure 5.1. First, some points are marked by the user with a three-dimensional brush such as a cube or a sphere. Next, the selected points are used to create a selection octree with CUDA. It uses Karras method which first creates Morton codes out of all currently visible points. Each Morton code represents one node with the smallest available resolution depending on the length of the Morton code. A code with 30 bits can represent an octree hierarchy of 10 levels. Those codes are sorted and a binary radix tree is constructed such that a search query can be done fast. The newly created octree can then be combined with an existing octree to generate a more complex selection, or the new selection can be used to subtract a given area of an existing octree. The resulting octree has the same resolution trough all nodes. This wastes memory and reduces the access speed because more nodes than necessary must be traversed to decide whether a newly loaded point belongs to the selected part of the octree.

Therefore, a compaction step in CUDA is proposed which uses the principles introduced by Gargantini in her work about linear quadtrees [Gargantini, 1982a]. Once the octree is compacted, it can be used in CUDA to decide whether newly loaded points belong to selected or unselected regions and change the color of the selected points to provide a visual hint between selected and unselected points.

5.2 Data Retrieval

As described in Section 2.4.1.1, every piece of data that should be manipulated by an algorithm written in CUDA must be moved to the global memory of CUDA, which is located at the memory of the graphics card. When data is rendered with OpenGL, it is common that the data is stored on the GPU memory for fast access instead of sending it to the GPU for each frame.

OpenGL uses *buffer objects* to keep the data in graphics memory to allow fast access to it [Segal and Akeley, 2004]. In the case of vertex data, the buffer object is named *vertex buffer object* (VBO). The usage of VBOs is also applied to a point cloud rendered in Scanopy: each rendered node of the MNO is available on the graphics card memory as VBO [Scheiblauer, 2006]. When the point selection is created, only the currently visible points are used for the selection. Therefore, only the points available as VBOs on the graphics card memory are used. With the CPU-based selection approach, the points have to be moved back to main memory before the selection can be constructed out of them. The data must be loaded on the graphics card memory anyway such that it can be processed with CUDA. As the required visible points are already present there, they do not have to be copied back to main memory.

However, the data present in the VBOs cannot be used directly in CUDA. Nivida provided the `cudaGLMapBufferObject` function to map the VBO to a memory area located in global memory. Before mapping it, it has to be registered using `cudaGLRegisterBufferObject`. This method to gain access to VBOs has been declared deprecated since CUDA 3.0. With the latest versions of CUDA, a VBO must be registered with `cudaGraphicsGLRegisterBuffer`. This provides a resource handle which can be used to access the data of the VBO. To get a pointer to the data, `cudaGraphicsMapResources` is used to map the resource. Once the resource is mapped, the pointer to the VBO data can be retrieved by calling `cudaGraphicsResourceGetMappedPointer`. As soon as the VBO data is no longer needed by CUDA, its resource handle has to be unmapped using `cudaGraphicsUnmapResources`. Listing 5.1 provides an example to get the data from VBO *1* and make it available in CUDA over the pointer `vbo_pointer`.

```
1  GLuint vbo = 1;
2  struct cudaGraphicsResource **cudaVboResource;
3  unsigned char *vbo_pointer;
4  size_t num_bytes;
5
6  cudaGraphicsGLRegisterBuffer(cudaVboResource, vbo,
7  cudaGraphicsRegisterFlagsNone);
8
9  cudaGraphicsMapResources(1, cudaVboResource, 0);
10
11 cudaGraphicsResourceGetMappedPointer((void **)&vbo_pointer,
12 &num_bytes, *cudaVboResource);
13 /*
14 do something with the data vbo_pointer points to.
15 */
16 cudaGraphicsUnmapResources(1, cudaVboResource, 0);
```

Listing 5.1: Sample to make VBO data available in CUDA

The structure of the VBO depends on the attributes that are available for the points in the point cloud. The only required attribute is the position. Usually, a color is provided too. Other attributes are intensity, normals, or the radii of the different points. These attributes are present in different data types, see Table 5.1 for an example. For this reason, a pointer of *unsigned char* (UChar) is used to map the VBO data into CUDA. An unsigned char has the size of one byte. To

work on the point data in CUDA, the amount of bytes occupied by all attributes of a single point is passed as stride into CUDA. Only the position of the point and its color are relevant to process the points in CUDA. Therefore, additionally to the stride, the offset to the beginning of the color attribute is passed as well into the kernel. Samples of different point-attribute combinations and their strides with corresponding color offsets can be seen in Table 5.2 .

Attribute	Data Type	Size in UChar
Position(Pos)	3 Float	12 UChar
Color(Col)	1 UInt	4 UChar
Intensity(Int)	1 Float	4 UChar
Normal(Nor)	3 Float	12 UChar

Table 5.1: Different attributes and their size in unsigned char (UChar).

Combined Point Attributes	Stride	Color Offset
PosCol	16 UChar	12 UChar
PosColInt	20 UChar	12 UChar
PosColIntNor	32 UChar	12 UChar
PosColNor	28 UChar	12 UChar

Table 5.2: Examples of combined attributes that appear in different point clouds and the resulting strides and color offsets. The abbreviations for the attributes introduced in Table 5.1 are concatenated to express possible attribute combinations for the points.

This data is converted in the CUDA kernel to the appropriate CUDA data type before it can be further processed using the corresponding offset and stride.

5.3 Memory Management

The previous section described how to prepare the data from the VBOs to be used in CUDA. Then it can be used immediately inside a CUDA kernel to perform calculations with it. However, there is also other data required to calculate a selection octree. For example the size of the bounding box of the point cloud and the accuracy of th Morton code must be passed from the main memory to CUDA. The same holds true for previously computed selection octrees. This data, which comes from the host's main memory, needs to be copied to the global memory before using it. As a first step, the memory allocation for CUDA is performed with the `cudaMalloc` command. Next, it can be copied to the devices, global memory using `cudaMemcpy`. Once the copy instruction is completed, the data can be passed to the kernel and used there.

As will be described through the following sections, the process to create or change a selection octree in CUDA is a multistep approach with several different kernel launches involved. These kernels not only depend on the data computed by previous ones but also continue to calculate new data which is then combined to the final outcome. During the implementation phase

input : A list of VBOs $vbos$ of size n
input : The previously computed octree old_octree
output: A linear octree

```
1  morton_codes ← GenerateMortonCodes(vbos);
2  sorted_morton_codes ← RadixSort ( morton_codes );
3  new_octree ← BuildBinaryRadixTree ( sorted_morton_codes) ;
4  if selected then          // selection:add more points to the octree
5  |  list_of_octree_nodes ← CombineOctrees (new_octree, old_octree) ;
6  else          // deselection:points are removed from the octree
7  |  list_of_octree_nodes ← SubtractOctrees (new_octree, old_octree) ;
8  end
9  sorted_list ← RadixSort ( list_of_octree_nodes );
10 sorted_list ← CompactData ( sorted_list );
11 new_octree ← BuildBinaryRadixTree ( sorted_list) ;
```

Algorithm 5.1: Octree creation

of the CUDA-based selection octree, the problem was discovered that memory needed to be allocated several times for the different kernel launches. Such memory allocations are expensive. Those allocations consumed more time than the computation of the selection octree itself. It turned out that the size of the required memory blocks which hold the data is similar among all required kernel launches. Therefore, a memory pool is used, which is created at the beginning of the construction algorithm. Depending on the required data size, the corresponding memory block which fits the data size is used. A continuous reallocation of the memory is no longer necessary because the allocated memory can be reused from the memory pool for the different CUDA kernels.

5.4 Octree Construction

The construction of the octree consists of the following steps marked in Algorithm 5.1. The first part until Line 3 is equal to the method proposed by Karras [Karras, 2012]. It has been extended by introducing a compaction method to get an octree which has a reduced number of points compared to the original. As the selection should be editable, it is not enough to create each octree once from a set of points. It is further required to change the selection. Therefore, a restructuring method for the octree is necessary as well. Section 5.4.3 describes the used approach to fulfill this task.

5.4.1 Morton Code Generation

When working with point clouds, the points are usually represented as floating point numbers. To generate the Morton code, the usage of integers is required. Therefore, the points are converted from floating point numbers to integers. Scanopy provides a bounding box in form of a cube for each point cloud that contains all points of the given point cloud. Using the bounding

box, it is simple to map all points into the unit cube. Now, all points have coordinates between 0 and 1. These coordinates are mapped to integers to generate the Morton codes from the coordinates.

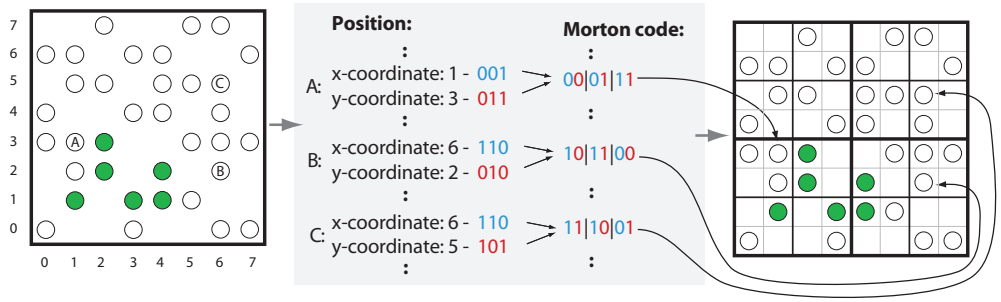


Figure 5.2: Sample construction of 6-bit Morton codes. The bits of the x and y coordinate are combined to get the Morton code. Note how each pair of two bits of a Morton code represents one level of the quadtree.

Following the method described in Section 2.2, the three coordinates are packed together into a single variable. Therefore, the precision depends on the used data type. The construction of the Morton codes and their relation to octrees are expressed in 5.2 which shows the construction of a quadtree with a 6-bit Morton code. To store three coordinates into a 32-bit integer, each coordinate can use 10 bits. All coordinates are mapped from the range of 0 to 1 in the domain of floating points to integers between 0 and 1023. When this precision is used to generate an octree out of the Morton codes, the resulting depth of the octree is 10. This means that the octree cannot be subdivided more than 10 times. Although this precision can be used to represent the selection octree, it is by far not fine grained enough. Therefore, a Morton code with a length of 30 bits can only be used for point clouds with a reduced expansion. For point clouds that cover a larger area, this resolution provides unpleasant outcomes. To express it in numbers: when the bounding box has an edge length of 64, the smallest octree node has an edge length of 1/16. When high-resolution point clouds with a bounding box edge length of 1024 have to be selected, it would result in an edge length of 1 for the smallest possible octree node. When those units represent meters, this would mean for the last case that it is not possible to provide a selection octree with a finer resolution than 1 meter. This is not detailed enough to represent parts of the object in an octree. As can be seen in Figure 5.3, the selection is not correctly represented due to the limited precision of 30-bit Morton codes. Increasing the precision to 60 bits can resolve this issue, as can be seen in Figure 5.4.

Variable Morton code length To compute the Morton code, more than one 32-bit integer is used. The current implementation supports up to 3 32-bit integers that are used to represent one single Morton code. To provide a general solution, only the last 30 bits per integer are used, although it would be possible to use more than 30 bits. This is the case, for example, when the Morton code is represented by 2 32-bit integers, only one bit cannot be used when one level of the octree is split between the 2 integers. As this would raise the complexity of the point-

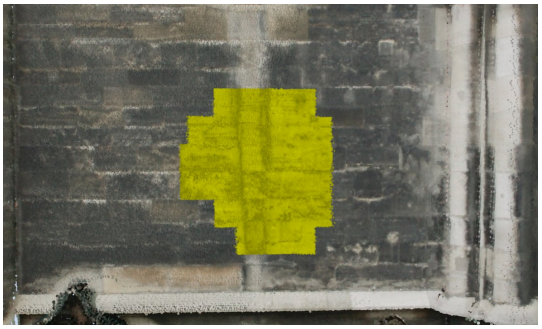


Figure 5.3: A circular selection using 30-bit Morton codes to represent the octee. Due to the limited precision of the octree representation, visible artifacts appear.

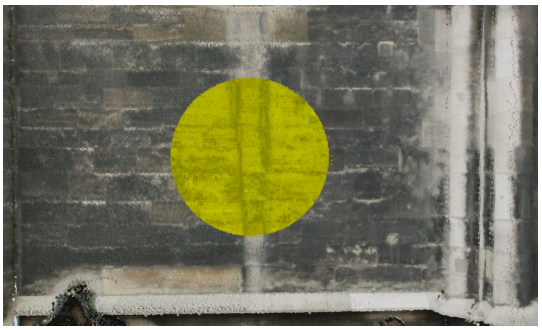


Figure 5.4: The same circular selection presented in Figure 5.3 uses 60-bit Morton codes. The higher precision reduces the visible artifacts.

selection method, the first two bits of each 32-bit integer are always omitted. Table 5.3 shows the available configurations.

Precision	Required integers	Morton code length	Octree depth
Low	1	30	10
Medium	2	60	20
High	3	90	30

Table 5.3: Different precisions and the required number of 32-bit integers to represent the corresponding Morton codes in memory.

The computation of the Morton code is done completely in CUDA. The data of the VBOs is made available for CUDA as described in Section 5.2. The parallelization is straight forward; each thread converts one point of the VBOs into three integers, one for each coordinate of the point. The range of the integers depends on the used precision. These three integers are combined into one Morton code using the defined amount of integers depending on the desired precision. The computation of the Morton code itself can be done with a few bitwise *and* and *or* and shift operations.

Similar to the CPU-based point selection presented in Section 3.3, the alpha value for each point is checked because it is used to distinguish between selected and unselected points. Therefore, the function `GenerateMortonCodes` not only provides the Morton code for each point. It also provides a flag which defines whether a point is selected or unselected, for each point. This makes it possible that the methods used in the following steps do not have to rely on the color information, which is only available in the VBO data.

5.4.2 Build-up and Compaction

When only one 32-bit integer is used for each Morton code, the construction of the octree can be done as described by Karras [Karras, 2012]. He uses radix sort to sort the Morton codes and creates a binary radix tree out of the sorted list of Morton cods. As described in the section before, using one integer is often not precise enough to create an octree. Therefore, up to three 32-bit integers are used per Morton code. But when more than one 32-bit integer is used, the sorting algorithm and the radix tree construction require small changes to handle those Morton codes. Internally, the Morton codes are stored as an array of 32-bit integers. When a Morton code consists of more than one integer, all integers belonging to the same code follow each other before the next code starts. The radix sort starts sorting with the least significant integer for each Morton code and moves forward until all integers that belong to the same Morton code have been processed. The changes required for the radix tree construction are straight forward. The prefixes must be computed over the whole Morton code using all integers that represent the corresponding code.

Compaction The final octree represents all points of the MNO nodes that contain at least one selected point. Each point is represented as a node in the octree. As there can be several million points visible, this can result in an octree with millions of nodes. This is not only a complex structure when it comes to processing it, but it also covers too much information in many regions of the octree. To reduce the number of nodes in the octree, a principle is used that is the same as introduced by Gargantini in her work about linear quadtrees [Gargantini, 1982a]. Only at the border, which splits selected and unselected regions, a large amount of points is required to represent this border precisely in the octree. Therefore, an algorithm is required which is capable to reduce the number of nodes within the octree on regions that do not provide additional information. This is true for all nodes whose children only consist of one type of nodes, either selected or unselected nodes.

To remove nodes from the octree, all the siblings of one node have to be retrieved. If the siblings have all the same selection state, all except one can be removed. The not removed node can then be used to represent the merged octree node. Since the used octree structure has no pointers, all siblings of a node cannot be retrieved by accessing the parent and use the pointers to its children. However, when the Morton codes are sorted, codes close to each other are also grouped together in the octree. For the Morton code representation of the octree, this means that each level of the octree is represented by 3 bits. Consider the lowest level of the octree, if two Morton codes only differ at the last 3 bits they have the same parent. Figure 5.5 shows this concept for quadtrees. Therefore, it is possible to access any random entry in the list of Morton codes, compare the left and right entries in the list as long as they do not differ by more than the last 3 bits they must have the same parent. As soon as the next tested Morton code does not hold this criterion, the search procedure can be stopped in this direction. When all siblings of a node have been found, they can be dropped if they all share one selection state, and the parent node is used to represent this area of the selection octree.

Considerations for MNOs As already discussed in Section 3.3, it is not guaranteed that all relevant points are loaded during a selection step. This has to be considered also in the com-

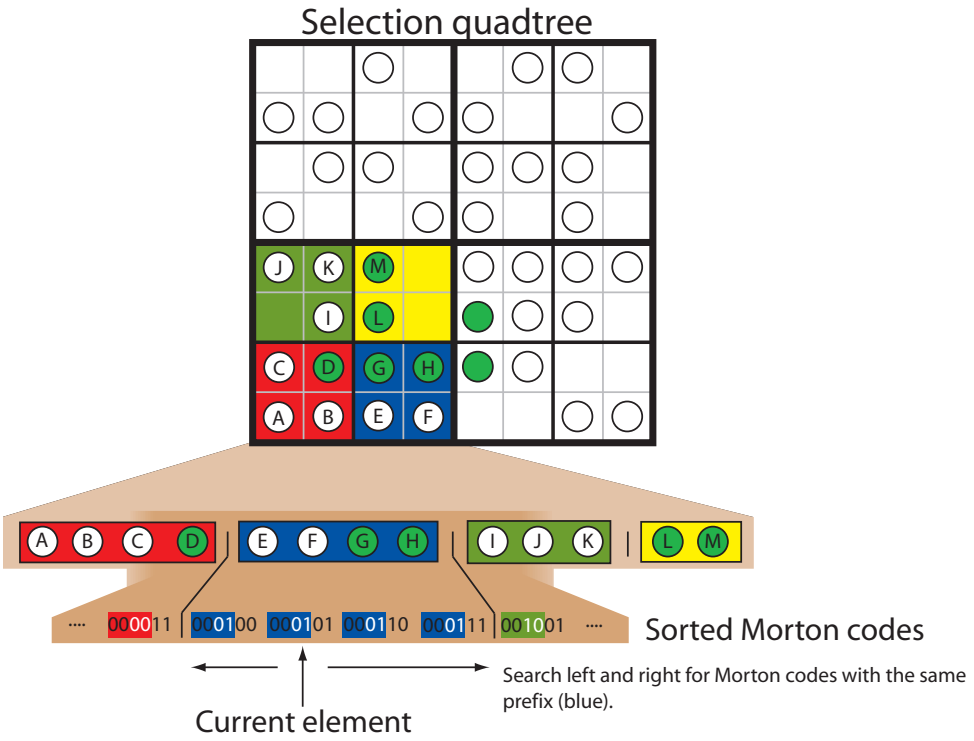


Figure 5.5: Sample of a 6-bit Morton code used to represent a quadtree with level 3. All siblings that share the same parent share all bits except the last two as well.

paction step. Not only must the selection state be the same for all siblings. It is also required that all 8 siblings are available to make sure no information is omitted due to the out-of-core rendering algorithm. Otherwise no compaction is allowed. If this is not considered, it might be the case that nodes are grouped together although there exist other nodes that are currently not displayed by the point-cloud renderer and are not available for a selection.

There is one exception to this rule. The maximum depth of the selection octree depends on the length of the Morton code. It might be the case that the maximum depth of the MNO is smaller than the maximum depth of the selection octree. The side length $minNodeSideLength$ of the smallest MNO node is retrieved using the side length $maxNodeSideLength$ of the point cloud's bounding box (which is a cube) and the maximum depth (counting levels [1..n]) of the MNO $maxMNODepth$: $minNodeSideLength = maxNodeSideLength/2^{maxMNODepth-1}$. As long as the side length of the nodes of the currently compacted octree level is smaller than the side length of the smallest MNO nodes, the nodes in this octree level can be compacted even if not all 8 siblings are available, but all available siblings have the same selection state. This can be done because in such a case the resolution of the Morton code (when used for creating selection octree nodes) is higher than the resolution of the MNO nodes, and therefore several Morton code nodes cover the volume of a single MNO node.

input : The input octree $octree$
output: A compacted octree

```
1 foreach octree level l in octree do
2 |   octree ← MarkUnnecessaryNodes ( octree, l );
3 |   octree ← ParallelCompaction ( octree ) ;
4 end
```

Algorithm 5.2: Overall octree compaction

CUDA compaction algorithm With this approach, it is possible to parallelize the compaction of the octree with CUDA. The process uses one kernel launch for each level of the octree. This is because after each step the unnecessary nodes can be dropped out of the octree. Nodes that are deleted from the octree change the structure of the octree. To make sure that all threads work on the same octree, it is required that all threads are in the same state before and after the deletion of the individual nodes. This is denoted as synchronization. The only possibility to synchronize between all threads is to wait until one kernel has finished its execution. Therefore, each level requires its own kernel launch. Whether a node is removed or not can be checked individually using one CUDA thread per node. Algorithm 5.2 shows the overall structure of the compaction solution for CUDA. The core part is Algorithm 5.3, which searches for unnecessary nodes and marks them.

Algorithm 5.3 runs in parallel on CUDA for each octree node individually. The octree is stored in CUDA's global memory. As shortly mentioned, each Octree node has to check its siblings by traversing the list of Morton codes left and right. Therefore, each node is potentially accessed several times by several threads. Accessing the same element multiple times is not efficient because global memory access is slow compared to registers and shared memory. Although the amount of shared memory is limited per block, it can still be used. It can be assumed that each thread traverses only a small amount of indexes along the list of Morton codes because each node has at most 7 siblings, and the traversal can be stopped as soon as an octree node from a different parent is discovered. Therefore, in Line 2 each thread stores its node to shared memory.

Finding reducible nodes The list of Morton codes to the left of the node is traversed to check for siblings. As a part of the left side of the list is stored in shared memory this is checked first. Due to the limited amount of the shared memory it can be necessary to check a small part of the octree which is only available in global memory as well.

`CheckNeighbour` checks for each potential sibling, whether it is a sibling and if it has the same selection state then the current node. Only if it is a sibling and it holds the same selection state as the current node, the traversal of the list continues. When the traversal of the list in the left direction is completed because all left siblings have been visited, the same operations are applied to the right side of the list. The current node is marked at the end of the algorithm. If all siblings have the same selection state as the current node, it can be dropped if it is not the leftmost sibling. The variable *compaction index* is used to remember if a node can be deleted. It

input : The input octree $octree$
output: A marked octree $octree$
output: A compaction index $comp\_index$

```
1  current_node ← octree[global_id] ;
2  shared_memory[thread_id] ← current_node;
3  valid_node ← current_node ->is_reducible ;
4  is_reducible ← true ;
5  if current_node is further reducible then
       // search for nodes to the left of the current node
6      while next element available in shared_memory AND valid_node == true do
7          valid_node ← CheckNeighbour ( shared_memory, is_reducible );
8      end
9      while next element available in global memory AND valid_node == true do
10         valid_node ← CheckNeighbour ( octree, is_reducible);
11     end
       // search for nodes to the right of the current node
12     while next element available in shared_memory AND valid_node == true do
13         valid_node ← CheckNeighbour ( shared_memory, is_reducible );
14     end
15     while next element available in global memory AND valid_node == true do
16         valid_node ← CheckNeighbour ( octree, is_reducible );
17     end
18     if is_reducible then
19         comp_index[global_id] ← 0;
20     else
21         comp_index[global_id] ← 1;
22         if has other siblings then
23             current_node->is_reducible ← false;
24         end
25     end
26     octree[global_id] ←current_node ;
27 end
```

Algorithm 5.3: Overall octree compaction

is set 0 if the node can be deleted. Otherwise the compaction index is set to 1 and it is marked that it is not further reducible if it has siblings with different selection states.

The outcome of `CheckNeighbour` can be seen in Figure 5.6, where a quadtree is used for better visibility. The compaction index is only visualized for the bottom left quadrant of the input quadtree surrounded with a red border. The concept works for the other quadrants in the same way. Moreover, the reducible flag is set for each node to memorize whether it can

be further compacted during the next steps. The circles inside the quadtree represent all visible points during a selection step. They are converted into Morton codes to get the presented input selection octree by using the steps previously presented in this Section. The yellow regions of the quadtree represent the selected areas of the point cloud.

Figure 5.6: The compaction algorithm applied to a sample quadtree. The bottom left quadrant of the input quadtree is used to show all the required steps to compact the quadtree. The final compacted quadtree is presented at the bottom right corner of the Figure. Note that the number of nodes used to represent the quadtree decreased from 35 to 14. This is only a theoretical example to explain the different steps of the algorithm. Compaction results on real data can be seen in Section 6.2.

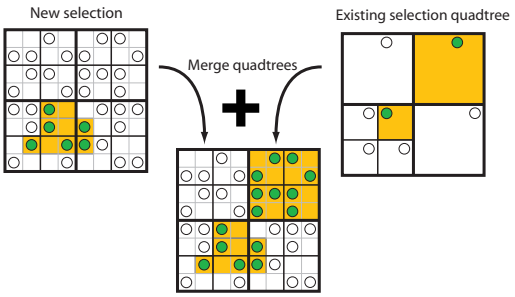


Figure 5.7: Add a new selection to an existing selection quadtree.

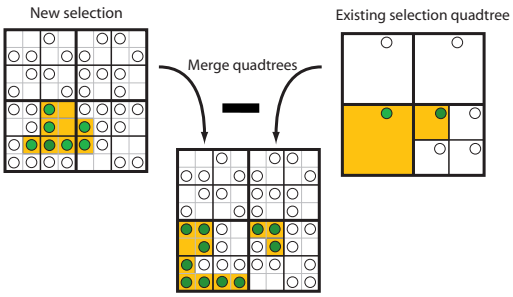


Figure 5.8: Subtract a new selection from an existing selection quadtree.

Parallel compaction with CUDA After the compaction index has been computed for all octree nodes and the CUDA kernel returns, Algorithm 5.2 continues with Line 3. Now, *parallel compaction* is applied to the octree data to remove all unnecessary nodes. The parallel prefix sum introduced in Section 3.6 is used to sum up all the 0's and 1's which result into a list that can be used as an index for the octree nodes. This index can be used to write the remaining octree nodes in parallel at the correct position. The points A to I in Figure 5.6 are compacted resulting in a smaller quadtree that holds only as many points as are necessary to represent the exact quadtree. No precision is lost during this step.

Note that if a compaction of siblings is possible, always the rightmost sibling stays in the octree although the leftmost sibling is the only one with a 1 in the compaction index. This is due to the construction of the parallel prefix sum which overrides the previous siblings when the right siblings compaction index originally was 0. However, it does not matter which of the siblings is kept when the nodes are reduces. Because the bits used to represent the currently reduced level of the octree are not necessary any more. This can be seen in Figure 5.6 as well. The first 4 digits of the Morton codes, which are responsible to define the parent of the nodes E, F and G, are the same. Since all three nodes have the same selection state, node emphE and F can be dropped and the first 4 bits of G are enough to represent the parent node.

Figure 5.6 shows the different steps of the compaction. The first reduction step assumes that the node size of the lowest octree level is smaller than the size of the smallest MNO node. This allows to compact nodes even if not all siblings are present. The result of this algorithm is a compacted octree visible in Figure 5.6 as *selection quadtree after step 2*. It helps to perform a faster point marking due to its reduced size. The details about the point-marking method are covered in Section 5.5. The possible compaction ratio achieved with this method and the performance benefits for point marking are covered in Section 6.2.

5.4.3 Merging

It should be possible to edit selections once they have been created. As the underlying structure of the point-selection method is a selection octree, a method is provided which is able to change

input : The current octree $current\_octree$
input : The selection as octree $selection\_octree$
output: A unsorted list of octree nodes represented in Morton code $node\_list$
output: A list of values whether a node is selected or unselected $value\_index$

```
1  if global_id < selection_octree->num_nodes then
2      if current_octree->num_nodes > 0 then
3          octree_node ← getOctreeNode ( current_octree,
           selection_octree[global_id] ) ;
4          value_index[global_id] ← (octree_node->is_selected OR
           selection_octree[global_id]->is_selected ) ;
5      else
6          value_index[global_id] ← selection_octree[global_id]->is_selected ;
7      end
8      node_list[global_id] ← selection_octree[global_id] ;
9  else
10     octree_node ← getOctreeNode ( selection_octree, current_octree[global_id
       - current_octree->num_nodes] ) ;
11     value_index[global_id] ← (selection_octree->is_selected OR
       octree_node[global_id - current_octree->num_nodes]->is_selected ) ;
12     node_list[global_id] ← octree_node[global_id - current_octree->num_nodes];
13 end
```

Algorithm 5.4: Octree combination

an already existing selection octree. In the overall octree creation Algorithm 5.1, this is done between Line 4 and Line 8. The rough idea of this solution is to build a second octree out of the newly selected nodes. This octree is denoted as `new_octree` in Algorithm 5.1. The octree created during the previous selection steps (here denoted as `old_octree`) can then be combined with the new octree. The result is a merged octree that includes the newly selected points into the octree created during the previous selection steps. Figure 5.7 shows the combination on the example of two quadtrees while Figure 5.8 shows how a new selection can be used to remove selected parts from an existing selection.

Merging without two sorting steps The disadvantage of this approach is that the Morton codes must be sorted twice before the final output is generated. This could be avoided if the newly selected points could be integrated into the existing octree without creating an individual octree for the new selection. This works as long as the new selection process has the same or more points available during the selection process. As long as the same points are visible during multiple selection steps, the points that are already present in the octree can be overwritten by the newly selected points if they are outside of the selected regions of the octree. This works due to the structure of the MNO because all points that are available looking from a far distance are also rendered when the user moves closer to the point cloud. As long as the user does not move further away from the selection, it is guaranteed that all unselected points in the octree can be

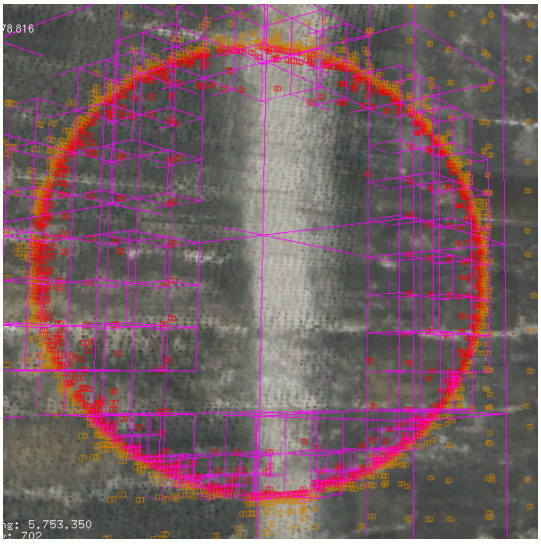


Figure 5.9: A compacted selection octree. The orange and red structures represent the points that belong to the selection octree. The selected points are red and the unselected points are orange. Note that the number of points is much higher at the border of the selection to represent its exact structure. The selection octree, which is rendered in pink, can be constructed out of the points in the octree.

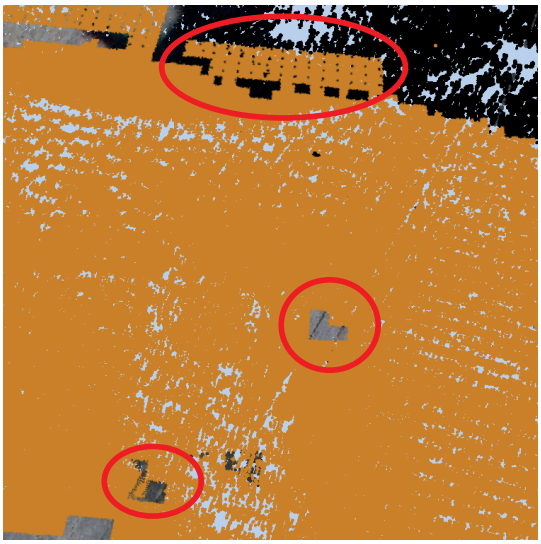


Figure 5.10: The artifacts, surrounded in red, result from insufficient number of points during two selection steps.

overwritten by newly selected points. All nodes that are not selected during the new selection, but fall into the existing selection octree, can be relabeled from unselected to selected. After the relabeling step has been completed, the construction of the new merged octree can be done by building it up according to Algorithm 5.1. The code from Line 3 to Line 9 becomes obsolete when this approach is used.

Limitations As soon as the new selection has fewer points available due to a different viewpoint, this solution does no longer work. The problem arises at the boundary of the selection octree. As can be seen in Figure 5.9, the compacted octree holds the most detail there. Overwriting the unselected points in the octree by the selected points of the new selection fails because the new selection has fewer points available that are present in the boundary region of the octree. When the octree is built with this method, it results in artifacts visualized in Figure 5.10. This kind of error could be avoided by modifying the compaction algorithm presented in Section 5.4.2. Points contained previously in the octree could be labeled differently than newly added points. This allows a specialized compaction, considering also the difference between old and new points. Incorrect holes as they appear in Figure 5.10 can be removed by compacting the

input : The current octree $current\_octree$
input : The selection as octree $selection\_octree$
output: A unsorted list of octree nodes represented in Morton code $node\_list$
output: A list of values whether a node is selected or unselected $value\_index$

```
1  if global_id < selection_octree->num_nodes then
2      if current_octree->num_nodes > 0 then
3          octree_node ← getOctreeNode ( current_octree,
           selection_octree[global_id] ) ;
4          if (NOT selection_octree[global_id]->is_selected) AND
           (octree_node->is_selected) then
5              octree_selected ← true;
6          else
7              octree_selected ← false;
8          end
9      else
10         octree_selected ← false;
11     end
12     node_list[global_id] ← selection_octree[global_id] ;
13 else
14     current_id ← global_id - selection_octree->num_nodes ;
15     octree_node ← getOctreeNode ( selection_octree,
       current_octree[current_id] ) ;
16     if octree_node->is_selected AND current_octree[current_id]->is_selected
       then
17         octree_selected ← false;
18     else
19         octree_selected ← current_octree[current_id]->is_selected;
20     end
21     node_list[global_id] ← current_octree[current_id] ;
22 end
23 value_index[global_id] ← octree_selected ;
```

Algorithm 5.5: Octree substraction

nodes although they consist of children with old unselected points and newly selected points. However, this algorithm destroys the structure of the selection octree by removing some details at the boundary between selected and unselected octree nodes. It is also slower than the here proposed solution which merges the new selection into the existing selection octree.

Final merging solution with CUDA The new selection is transformed into a temporal selection octree which can be combined with the existing one. It has to be distinguished whether the selected points should be added to the existing octree, or they should be used to subtract

the newly selected region from the existing selection octree. Both methods, the combination and the subtraction, run completely in CUDA. For each node of both octrees one thread is used. The two CUDA kernels are presented in Algorithm 5.4 and Algorithm 5.5 respectively. Conceptually, they are similar and use two selection octrees as input. The output is a list which combines all nodes of both octrees together. Additionally, for each node the values list identifies whether the node is selected or not. This is important because both algorithms might change the selection state of each octree node. For each octree node, it is checked in which octree node of the other octree it fits in. Depending on the merge operation and the states of the two nodes, the selection state of the checked octree node might change. In the case of the combination operation the state change is simple. As can be seen in Figure 5.7, each unselected node of one octree must be marked as selected if it falls into a selected node of the other octree. Figure 5.8 shows the subtraction case, which is not symmetric as it was the case for the combination. The `selection_octree` is used to subtract all points from the `current_octree` which intersect with the `selection_octree`. The performed intersection test, which is done per node, can be seen in Algorithm 5.5. Note that in the case of the `selection_octree`, all points have to be remarked as not selected except those that fall in the `current_octree` and are not already selected. This is because during the deselection process, all points used for deselection are marked first to make a distinction between deselected points and all other points. Once the deselection from `current_octree` is performed, they have to be set back to non selected.

The output of the merging methods is a list of octree nodes represented in Morton code and the corresponding correct selection states for each node. The remaining steps to complete the octree construction can be seen in Algorithm 5.1 from Line 9 to Line 11. The list has to be sorted and can then be compacted using the method described in Section 5.4.2. After the creation of the binary radix tree, the octree is merged and its construction is complete.

5.5 Point Marking

As already described in Section 3.3, due to the out-of-core rendering method of Scanopy, not all points are available during the selection process of a point cloud. Once the selection octree has been created, points that become visible when a new frame is rendered must be tested against the selection octree. This is required to mark each point to visualize whether it is selected or not. This is done by assigning all selected points a specific color. This method was proposed by Scheiblauer et al. for the CPU-based point selection [Scheiblauer and Wimmer, 2011]. As the selection octree is created in CUDA, a method is proposed to perform the point-marking test of each selection octree also on the GPU using CUDA.

Generally, the marking process has to be applied only when a new MNO node is loaded to the GPU to render it, or when the selection state changes. A selection can be active, inactive, or invisible. Therefore, it is required to distinguish selected points from not selected ones. If the state of a selection changes, it is required to set all points which belong to the selection to the new state.

The case of a node put on the GPU is considered first. This requires a brief introduction to the out-of-core algorithm of Scanopy. As described in Section 3.2, MNO nodes are loaded on demand from the hard disk. However, the number of MNO nodes loaded from disk to main

memory is limited per rendered frame. In the case of the CPU-based selection, all points can be tested immediately against the selection octree before the VBO for the node is created and moved to the GPU.

This method is not effective when a point-marking method is created in CUDA for two reasons. First, the data must be loaded to the graphics card memory anyway before it can be rendered. Therefore, loading it first into CUDA's global memory, coloring the selected points and copying it back to main memory such that it can finally be loaded on the VBO to render it is not effective. The second reason is that the CPU-based method works on the points of each MNO node individually. Doing the same in CUDA would mean to launch for each MNO node its own CUDA kernel and perform the required marking operation there. Not only does each kernel launch take some time, but also the memory management to make required octree information available in CUDA is not cheap.

Therefore, all MNO nodes that become available during one frame put their data on the respective VBOs instead of marking them before the creation of the VBO, as it is done during the CPU-based approach. As soon as all nodes are available as VBOs in graphics card memory, the data of the points can be accessed and modified by CUDA as already introduced in Section 5.2. This way, copy operations between CUDAs global memory and the main memory of the CPU can be avoided. Combining the point-marking operation of all newly available MNO nodes into one kernel call improves the performance as well due to the reduction of kernel calls and memory instructions.

The second case is concerned with the state changes of the point selections. In case of the CPU-based selection, the marking method is performed in a top down fashion. This means that the root node of the MNO is tested against the selection octree first. The children of the root node are recursively tested against the selection octree. They only have to be tested if they are actually visible, which is only the case when they are available as VBO. Therefore the testing process on the CPU slightly differs from the first case because in this case, the data is only available in the VBO and has to be copied back to main memory to test it against the CPU-based selection octree.

The fact that the data is already on the graphics card in form of VBOs supports the CUDA-based marking process, as the data does not have to be copied back to main memory. While the CPU-based method tests also in this case against all VBOs separately, the CUDA-based approach tests all VBOs at once against the selection octree for the same reason as in the first case. Before the point-marking method is executed in CUDA, all MNO nodes that are available as VBOs are collected in a list to pass it once into the CUDA kernel responsible for the point marking.

The CUDA kernel to mark each point is simple. Each point is processed by one thread. The point is checked against the selection octree and its color is set depending on whether it falls within a selected or unselected node.

The performance difference between the CPU-based point marking and the CUDA-based point marking will be discussed in Section 6.4.

5.6 Persistent Storage

As the selection octrees represent the links between point clouds and annotations, they are also stored in the database such that anything related to annotations is stored in the same location. The data size of each selection octree should not become too big, as it is assumed that only small parts of the whole point cloud are selected for the different annotations.

Both the linear octree, represented as Morton codes, and the attribute for each node, which states whether it is selected or not, are stored in the database as binary object. To successfully convert it back from binary format into 32-bit integers, the number of nodes as well as the used precision for the Morton code must be stored tool.

CHAPTER 6

Results

The developed point-selection and point-marking solution is analyzed in this chapter. The point-selection method creates the selection octree from a set of selected points. The point-marking solution uses the selection octree to test whether newly loaded points from a point cloud belong to the selection octree and have to be marked. Section 6.1 describes the benchmarking setup used to provide the results presented throughout this Chapter. The benefits of the compaction method are presented in Section 6.2. Section 6.3 shows the achieved performance with different graphics cards and with different CUDA compute capabilities. It also compares the existing CPU-based point-selection method with the CUDA-based method introduced in this thesis. Section 6.4 presents the results of the point-marking method, comparing different graphics cards and analyzing the performance difference between CUDA-based marking and CPU-based marking.

6.1 Benchmarking Setup

There exists a broad variety of CUDA-capable devices starting from the first series of Nvidia's *GeForce 8800* graphics cards to today's *GeForce 700* series. Not only the number of parallel processing units and their clock rate has changed, but also the overall architecture has evolved. Section 2.4.1.1 covers the details of this development. Due to this evolution of CUDA-capable devices, a variety of different hardware settings is used to analyze the performance of the developed point-selection and point-marking method, and they are compared with each other.

The four systems presented in Tables 6.1, 6.2, 6.3 and 6.4 have been used to compare the developed algorithms and they will be referred to as benchmarking system 1 to benchmarking system 4. The point clouds used for the tests are listed in Table 6.5.

6.2 Octree Compaction

The benefits of a compacted octree used during point marking are analyzed in this section. For the compaction to be useful, the selection octree should be reduced by a reasonable amount.

Component	Specification
CPU	AMD X2 6000+ / 3.0 GHz
Main memory	4 GB
Graphics card	Nvidia GeForce GTS 250 128 CUDA cores 512 MB of memory Compute capability 1.1

Table 6.1: Desktop PC with compute capability 1.1

Component	Specification
CPU	Intel i7-2600K / 3.4 GHz
Main memory	16 GB
Graphics card	Nvidia GeForce GTX 570 480 CUDA cores 1280 MB of memory Compute capability 2.0

Table 6.2: Desktop PC with compute capability 2.0

Component	Specification
CPU	AMD X2 6000+ / 3.0 GHz
Main memory	4 GB
Graphics card	Nvidia GeForce GTX 660 960 CUDA cores 2 GB of memory Compute capability 3.0

Table 6.3: Desktop PC with compute capability 3.0

Component	Specification
CPU	Intel i7-3612QM / 3.1 GHz
Main memory	4 GB
Graphics card	Nvidia GeForce GT 630M 96 CUDA cores 2 GB of memory Compute capability 2.1

Table 6.4: Notebook with compute capability 2.1

Otherwise there would be no performance improvement for the point-marking step. As has been discussed in detail in Section 5.5, point marking is required when a new MNO node becomes visible or a selection changes its visibility. The compaction results are measured using the benchmarking system from Table 6.3.

The *Building* and *Catacombs* point clouds denoted in Table 6.5 are used for the compaction test.

Name	Points	Scan Positions	Size	Point Attributes
Building	7,358,930	1	112 MB	position, color
Catacombs	1,921,537,902	1826	29384 MB	position, color

Table 6.5: Point clouds used for the compaction test

As can be seen in Table 6.7, the possible number of nodes that can be compacted can vary depending on the structure of the selected part of the point cloud. Selections with the highest potential to result in a small selection octree are those which have only a limited amount of not selected neighbour points. Pillars can be seen as a good example for this case. Once a selection is completely surrounded by neighbour points, its possibility for compaction depends on the complexity of the border of the selection, which seperates selected from unselected points. Table 6.6 shows examples of selections with different border complexities and the resulting

compacted selection octree. Morton codes with a length of 60 bits are used for this test. In all cases the possible amount of memory that can be saved due to the reduction of nodes is at least 96,5 %, as can be seen in Table 6.6 and 6.7. The selections of the *Catacombs* data set are presented in Figure 6.1. The amount of remaining nodes compared to the original selection octree are presented in the last column of Table 6.6 and 6.7.

	Octree			
Selected Points	Construction Time [ms]	Compaction Time[ms]	Final Nodes	Percent
235,052	88.9	24.6	2,151	0.91 %
206,812	82.6	14.1	860	0.42 %
275,887	90.9	23.7	3,688	1.34 %
316,559	106.1	25.6	4,771	1.51 %
390,081	117.2	27	8,292	2.13 %
613,854	150.5	34.9	21,775	3.55 %

Table 6.6: Marked *Building* data set with compaction

	Octree			
Selected Points	Construction Time [ms]	Compaction Time[ms]	Final Nodes	Percent
504,199	124.5	27.2	396	0.08 %
1,201,430	223.6	40.5	12,615	1.05 %
1,296,256	283.1	60.4	2,078	0.16 %
817,474	162.6	30.8	9,770	1.20 %

Table 6.7: Marked *Catacombs* data set with compaction

The direct comparison between compacted and not compacted selection octrees can be seen in Figures 6.2 and 6.3. There is a clear connection between the time required to test the points against the corresponding selection octree and whether the corresponding selection octree is compacted or not. As expected, it is the case that compacted selection octrees speed up the marking step of the point clouds.

It can be noted that the achievable speed up for compacted selection octrees during the point-marking step is not as high as the potential memory savings. As can be seen in Table 6.6 and 6.7, between 96,5% and 99,9% of all nodes can be dropped during the octree compaction. On the other hand, when compacted selection octrees are used, the computation time required for the point-marking step is at most reduced by half compared to the not compacted octrees. Still, the marking step benefits from a compacted octree, especially if it was originally created by a large set of selected points.

Figure 6.1: The selections created on the *Catacombs* data set presented in Table 6.7.

6.3 Point Selection

The overall construction process for a selection octree in CUDA has been presented in Section 5.4, and Algorithm 5.1 shows the general structure of the construction algorithm. To get an overview of the time required by each subpart of the algorithm, the benchmarking system 6.2 is used, and with it several selections are performed using the *Building* data set. The result of this test can be seen in Figure 6.4. During the creation of the selection octree, most of the time is spent for the two sorting steps. When the number of points involved into the creation of the selection octree increases, the time required for the Morton code generation and the compaction step, which is used to reduce the number of points in the selection octree, increase notably as well.

To test the selection step on the different hardware setups, the selection scene composed of one point cloud and different selections was created and stored as a scene file on hard disk. This allows reopening the scene on the other systems such that the camera is already placed at the correct position, as the position is also stored on the scene file. The selection step can then be repeated on the other systems. This can result in a selection which contains a set of slightly different points because the selection is done manually with the mouse. However, measurements have shown that the required time to complete the construction of the selection octree is the same

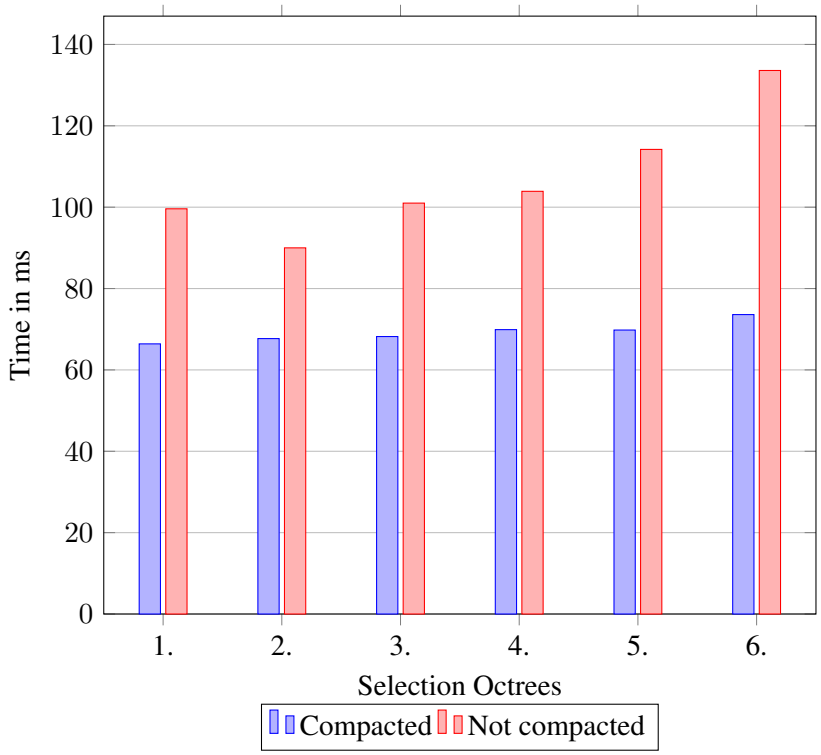


Figure 6.2: Difference in time required to mark compacted and not compacted selection octrees of the *Building* data set

as long as the selected number of points does not vary too much and the selections have a similar form.

When the different hardware setups are compared, the comparison between benchmarking systems 1 and 4 is of special interest. Both GPUs provide a similar amount of CUDA cores. However, between the two graphics cards the GPUs of the 400 and 500 series lay in between. Also, the compute capability has been increased from 1.1 to 2.1. Despite these gaps, the selection process does not benefit from the newer architecture of the GPU. The selection process even performs marginally faster on the GTS 250 compared to the GT 630M. If the system that contains the GTX 570 is compared with the system containing the GTX 660, the selection results show that the GTX 570 performs the selection task on average in two thirds up to half the time the GTX 660 requires. This is interesting as the GTX 660 has a higher compute capability compared to the GTX 570 and its CUDA cores have been doubled from 480 to 960. Still the GTX 570 can perform the selection task faster.The reason can be a more complex scheduler used by the Fermi architecture compared to the Kepler architecture, which handles data dependencies with special hardware during runtime, whereas the simpler scheduler of the Kepler architecture relies on the compiler to handle these dependencies [Nvidia, 2012]. Therefore, the Fermi scheduler can deal with the random accesses to the linear octree, which are not predictable at compile time, more efficiently. Moreover, the number of streaming multiprocessors (SM) decreased from

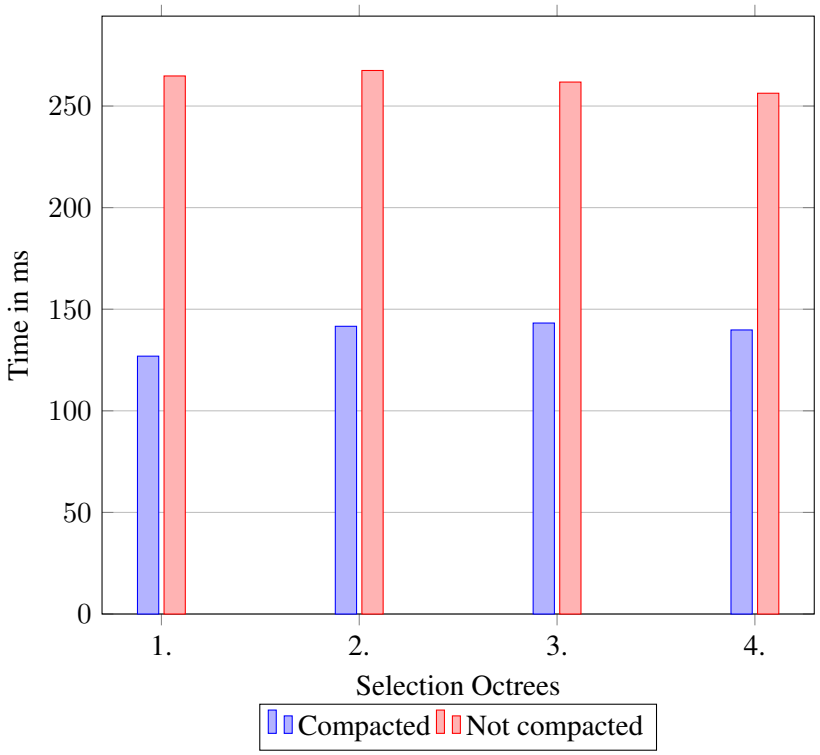


Figure 6.3: Difference in time required to mark compacted and not compacted selection octrees of the *Catacombs* data set

15 on the GTX 570 to 5 on the GTX 660. This reduction in SMs requires to triple the parallelism in one streaming multiprocessor of the GTX 660 to provide the same performance compared to the GTX 570 [Nvidia 2013]. This is not possible because of the register limit on each streaming multiprocessor. Although the number of registers per SM is doubled from the GTX 570 to the GTX 660, it is not enough to cover the required increase of parallelism, which must be tripled to provide the same speed on the GTX 660.

As already discussed in Section 5.4.1, when using Morton codes as a representation of an octree, the precision of the octree is limited by the number of bits used per Morton code. Therefore, the desired precision per selection octree can be changed. It is assumed that the length of the Morton code has an impact on the performance of the point-selection method. Figure 6.5 confirms this assumption. It shows the time required to build the selection octree using three different Morton code lengths. As test selections, the first four selections from Table 6.8 are chosen. The results come from benchmarking system 4 using an Nvidia GT 630M. Although the absolute time required performing the selections varies using other GPUs, the relative time required for the different parts of the selection process stays the same. Looking at the previous results, which present the required computation time per subpart of the selection process, the sorting steps are the most time consuming. Radix sort is used as the sorting algorithm. Since the running time of this sorting algorithm depends on the length of the keys, the required time to

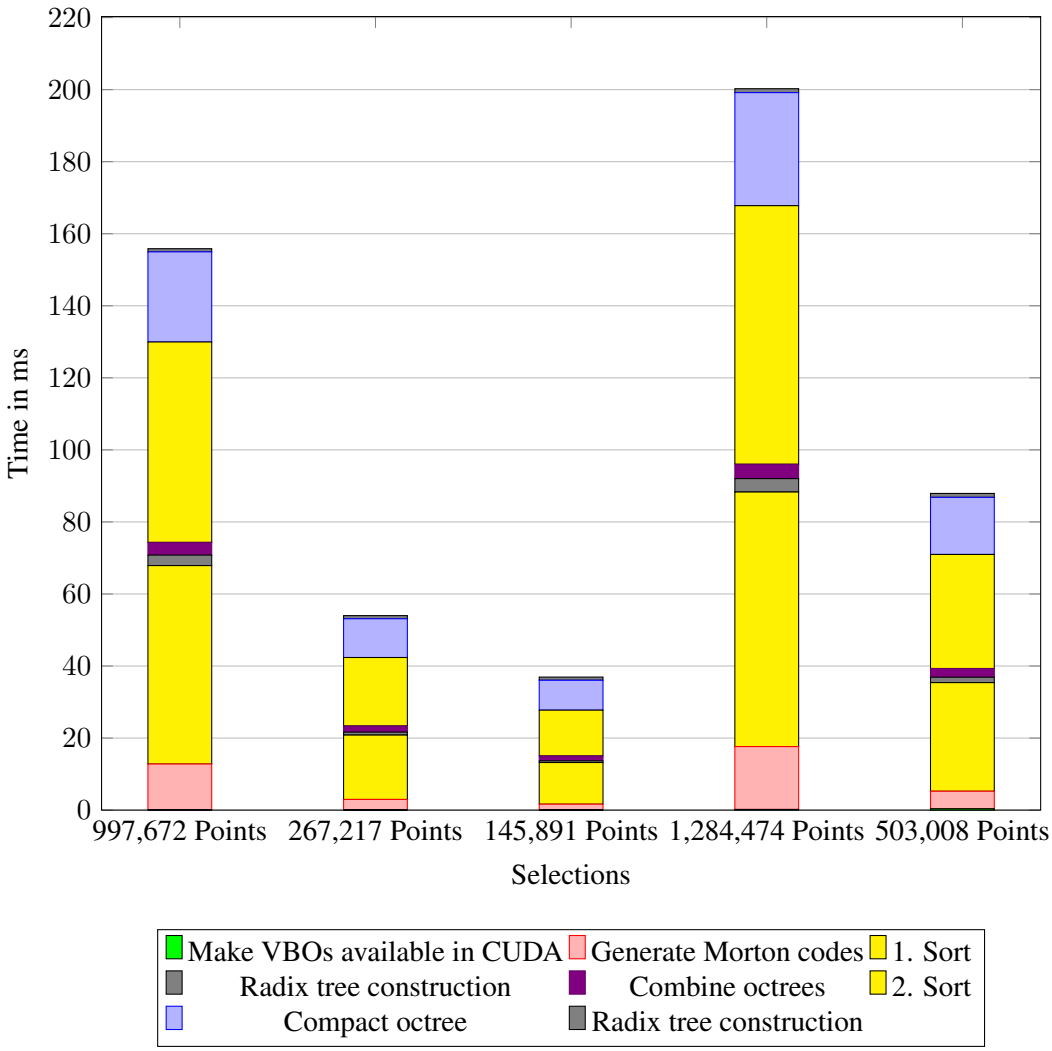


Figure 6.4: Point selections of the *Building* data set and the time required for each step

complete the construction of the selection octree should also increase when longer Morton codes are used. The compaction step depends as well on the length of the Morton code. Therefore, also the compaction time almost doubles when the length of the Morton codes is increased from 30-bit to 60-bit values. Those two steps have already been confirmed by the measurements in Figure 6.4 to be the most time consuming during the construction process. This shows that the length of the Morton code has immediate influence on the construction of the selection octree as can be seen in Figure 6.5.

During the individual selection steps, each brushstroke must be tested against the point cloud on which the selection is performed. This is done by testing each point of the point cloud if it is inside the brush's volume. First, this was considered as a good possibility to optimize the selection process with CUDA by testing each point with one CUDA thread. Unfortunately,

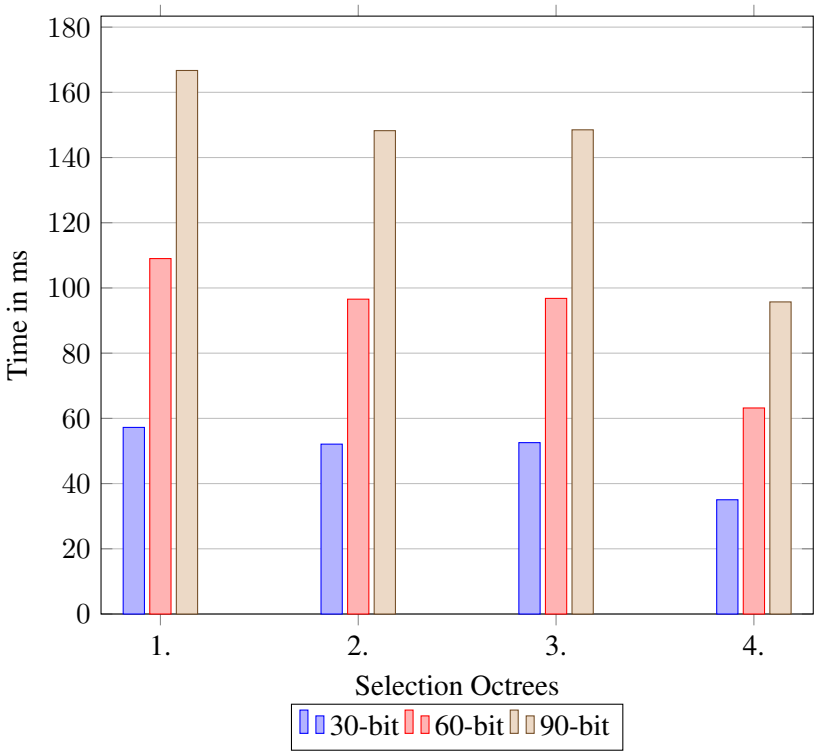


Figure 6.5: Comparison of the required construction time using different lengths of the Morton code. The first four selections from Table 6.8 are presented, which use the *Building* data set as model.

performance analysis and comparison with the existing CPU test showed that the usage of CUDA was not faster than the CPU-based containment test. The outcome was surprising as the idea looked promising due to the perfect mapping to parallel systems. The problem with the CUDA-based testing approach was that there is one communication step required after the testing of the points. Once each point of an MNO node has been tested against the brush shape, it is required to return the amount of points which are inside the brush. Only when the MNO node contains selected points, it is relevant for the octree construction. Therefore, this number must be passed back from CUDA's global memory to the main memory where all required MNO nodes are collected in a list. This list ensures that the octree calculation is only performed on MNO nodes that contain selected points. To test whether a point of an MNO node is inside the brush is faster in CUDA compared to the CPU-based. However, the time required to copy the number of selected points back to main memory is slow. The overall performance measurement makes the CUDA-based approach as slow as the CPU-based method. Since both methods need the same time to complete this task, it does not matter whether CUDA or the original approach is used to select the points with the brush.

6.3.1 Point Selection CPU vs. CUDA

When the time required by the CPU to create a point selection is compared with the CUDA-based solution, both the GTX 570 and the GTX 660 outperform the existing CPU based selection by far. Especially the GTX 570 is able to complete the selection task more than 10 times faster than the CPU.

Selected points	CUDA build-up time [ms]	CPU build-up time [ms]	Speedup
104,812	21.8	45,1	2,1
92,484	20.3	48,6	2,4
92,484	20.2	32,6	1,6
56,369	15.1	16,4	1,1
234,560	36.9	63,3	1,7
199,830	33.0	72,2	2,2
467,461	68.8	289,7	4,2
522,373	75.5	215,1	2,8
244,013	44.0	243	5,5
255,770	44.2	133,2	3,0

Table 6.8: Point selections on the *Building* data set using the system described in Table 6.2 containing a GTX 570

Selected points	CUDA build-up time [ms]	CPU build-up time [ms]	Speedup
1,003,475	128.9	281.5	2.2
1,329,920	165.5	540.0	3.3
529,689	77.8	91.7	1.2
682,923	94.1	213.9	2.3
613,433	87.9	216.6	2.5
710,219	98.7	236.4	2.4
1,775,522	214.4	1045.7	4.9
369,553	59.4	900.2	15.2
1,474,452	181.9	859.0	4.7
565,527	79.6	211.0	2.7

Table 6.9: Point selections on the *Catacombs* data set using the system described in Table 6.2 containing a GTX 570

The CUDA-based approach is especially faster compared to the CPU under two conditions. A large amount of selected points provides a better improvement. When building a selection out of 100,000 points or less, the speedup of the CUDA-based approach is at most tripled compared to the CPU-based selection. Larger selections with millions of selected points benefit from the CUDA-based selection by improving the selection speed 5 times and more. See Figures 6.6 and 6.7 for details. The second factor which determines the speed improvement of the CUDA-based selection method to the CPU-based method lies in the structure of the selection.

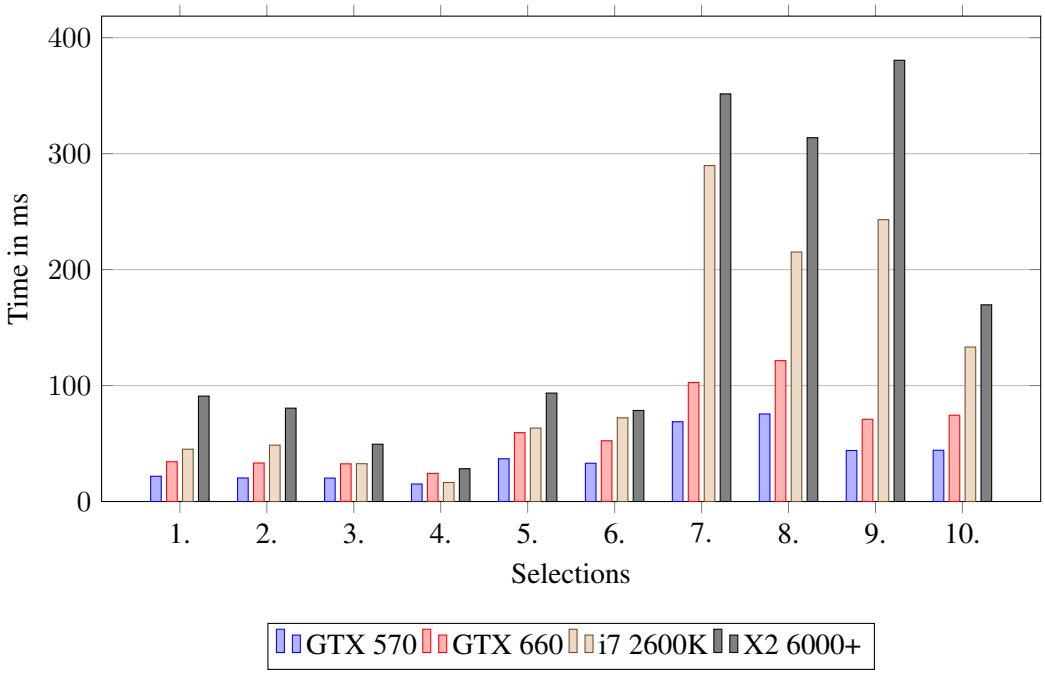


Figure 6.6: Point selections using the *Building* data set. Details about the selections are presented in Table 6.8.

The CUDA-based method only depends of the number of involved points to create the selection. The running time of the CPU-based method depends on the structure of the selection. This comes from the difference of the build-up process. The existing CPU-based method builds the selection octree in a top-down fashion. This means that it divides the initial cube surrounding the point cloud until each node contains either only selected or only unselected points. This process can potentially be stopped early for some nodes during the construction process. This results in a faster construction of the selection octree. If such an early stopping is possible due to a selection which does fit well into octree nodes of a higher level, the construction takes less time on the CPU.

Such an example can be seen in Figure 6.7, the third selection is not faster using CUDA because an isolated object is selected. The splitting process can be stopped early when using the CPU-based method because there are no unselected points close to the selected object.

6.4 Point Marking

To analyze the difference between the Fermi architecture and the older CUDA devices, the benchmarking systems 1 and 4 are used, which use a GTS 250 and a GT 630M respectively. Both have a similar amount of CUDA cores as can be seen in Table 6.1 and 6.4. Although the GTS 250 has 32 CUDA cores more than the GT 630M, the marking step is slower on the GTS 250 as can be seen in Figure 6.8. The reason for this is the access pattern to the selection

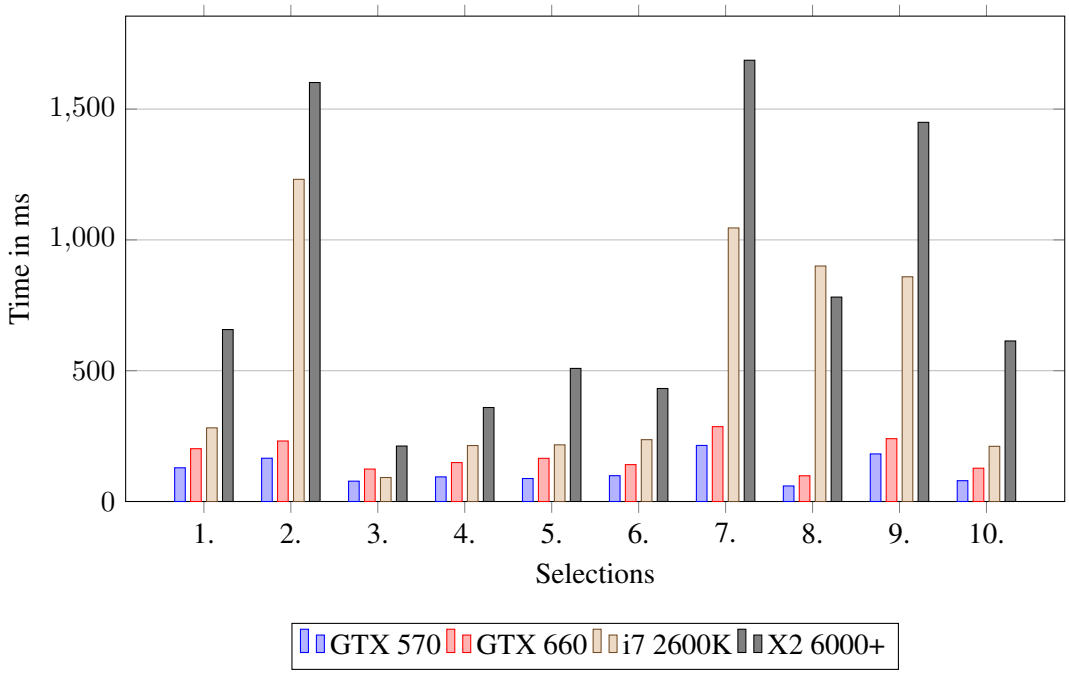


Figure 6.7: Point selections using the *Catacombs* data set. Details about the selections are presented in Table 6.9.

octree. It cannot be structured such that the memory can be accessed in coalesced way. The L1 cache, which has been introduced with the Fermi architecture, supports the access of the irregular access pattern to the selection octree. Therefore the GT 630M outperforms the older GTS 250.

6.4.1 Point Marking CPU vs. CUDA

The improved speed of the CUDA-based point-selection method compared to the CPU-based method alone does not provide the required speedup to use a large amount of annotations. To work with several point selections such that they can be used as a link between point cloud and annotations, a fast point-marking step is more important. The selection octree, once created for each annotation, remains constant. Therefore, the construction time is not as crucial as the time required to mark the points that belong to the selection. Whenever a new MNO node becomes visible, it has to be tested against the currently available selections because it has to be decided for each point individually whether it belongs to a selection or not. If it does belong to a selected node of an octree, its color must be changed such that the user knows that this point belongs to the selection as well. As the amount of annotations increases, also the number of point selections increases because each annotation holds its own point selection to create a link between point cloud and annotation. To compare the existing CPU-based point-marking approach with the CUDA-based approach implemented in this thesis, a test setup has been created.

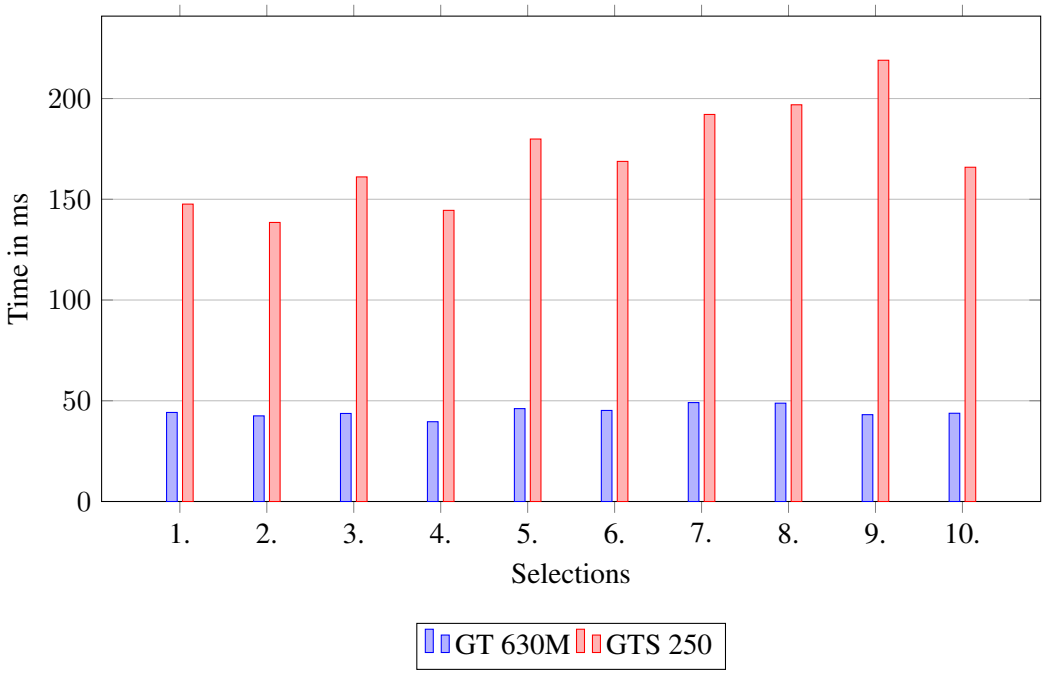


Figure 6.8: Point marking using the *Building* data set.

To compare the existing CPU-based point-marking approach with the CUDA-based approach implemented in this thesis, two different test setups have been created. The first method compares how fast all currently visible MNO nodes can be tested against existing selection octrees. This is done be switching the selection from not active to active. When such a switch is performed all currently visible points have to be tested against the selection octree. If a point falls within the selected part of the selection octree it is marked as selected. The comparison is done using the *Building* and the *Catacombs* presented in Table 6.5.

The results of this test can be seen in Figure 6.9. The GTX 570, which is one of Nvidias high-end graphics cards, outperforms all the other GPUs used in this test. Moreover, it shows the best performance results compared to the CPU-based marking approach. The results are similar to those presented in Section 6.3 which covers the results of the selection methods. Large selection octrees benefit much more from the CUDA-based method than selection octrees resulting from smaller selections. However, the performance advantage for the CUDA-based method to compare newly loaded points against existing selection octrees is even higher compared to the CPU-based method than it was the case when the point-selection methods have been compared with each other. It is able to mark the points which have to be tested against large selection octrees almost 20 times faster than the CPU-based method. The results show as well that the GTX 660 is not able to compete with the older GTX 570. Although the GTX 660 has twice as many CUDA cores as the GTX 570, it is three times slower than the GTX 570. The reason for this have been discussed in Section 6.3. The reduced number of streaming multiprocessors and the simpler scheduler used on the GTX 660 seams to have a even greater impact. This comes

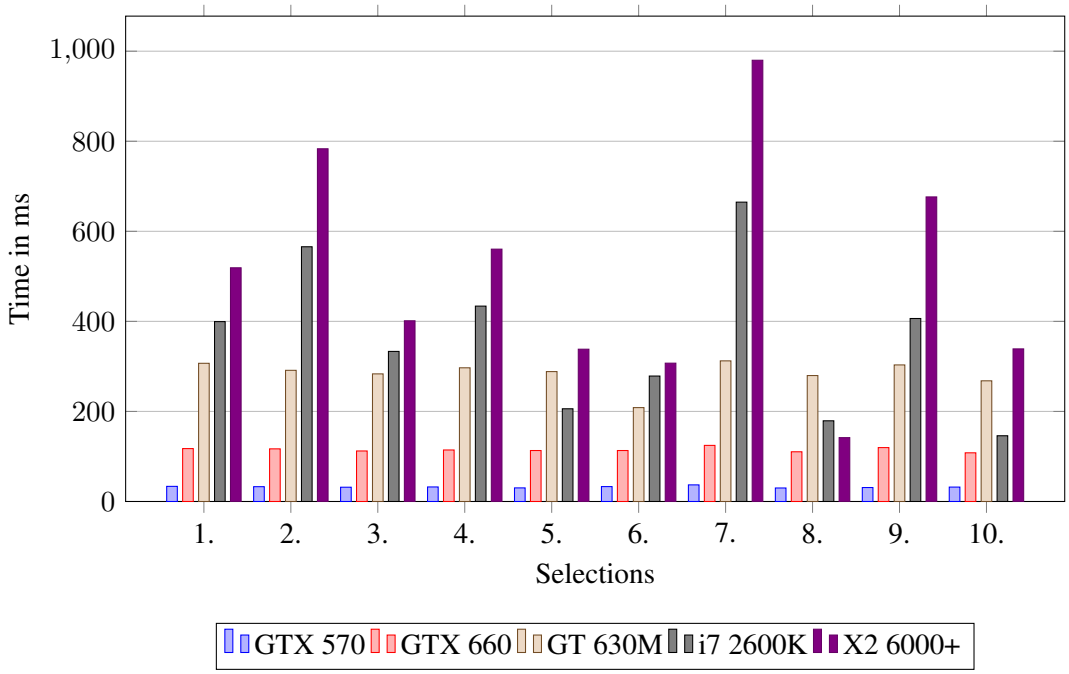


Figure 6.9: Point marking using the *Catacombs* data set. Details about the selections are presented in Table 6.9.

from the random access to the selection octree, which is used to test whether a point belongs to the selection.

The size of the selection octree has only limited influence on the computation time of the CUDA-based marking method. This is not true for the CPU-based method. Selection octrees with limited depth can be performed much faster. Such selection octrees result from selections which have only a limited number of unselected points close to the selected points. As already discussed in Section 6.3, example cases are completely selected objects such as pillars or selections with small boundaries between selected and unselected objects as they appear when circular or rectangular areas of the point cloud are selected. See the selection octree 3 in Figure 6.9 which represents the marking of a selected pillar.

The second test setup simulates a real movement of the user through the scene. During this traversal, other parts of the point cloud become visible. This makes it necessary to load new MNO nodes from the hard disk to the graphics card. The points within the newly loaded MNO nodes must be tested against all active point selections. This test is required to decide whether a point must change its color to represent it as selected or it can keep its original color if it does not belong to any of the selected regions of all active selection octrees. Which MNO nodes are loaded from disk depends on the path the user moves though the point cloud. To make sure this traversal is equal for both, the CPU-based method and the CUDA-based method, a automated camera path is created. This path moves the camera on a fixed track through the scene and makes sure to load the same MNO nodes for all tests.

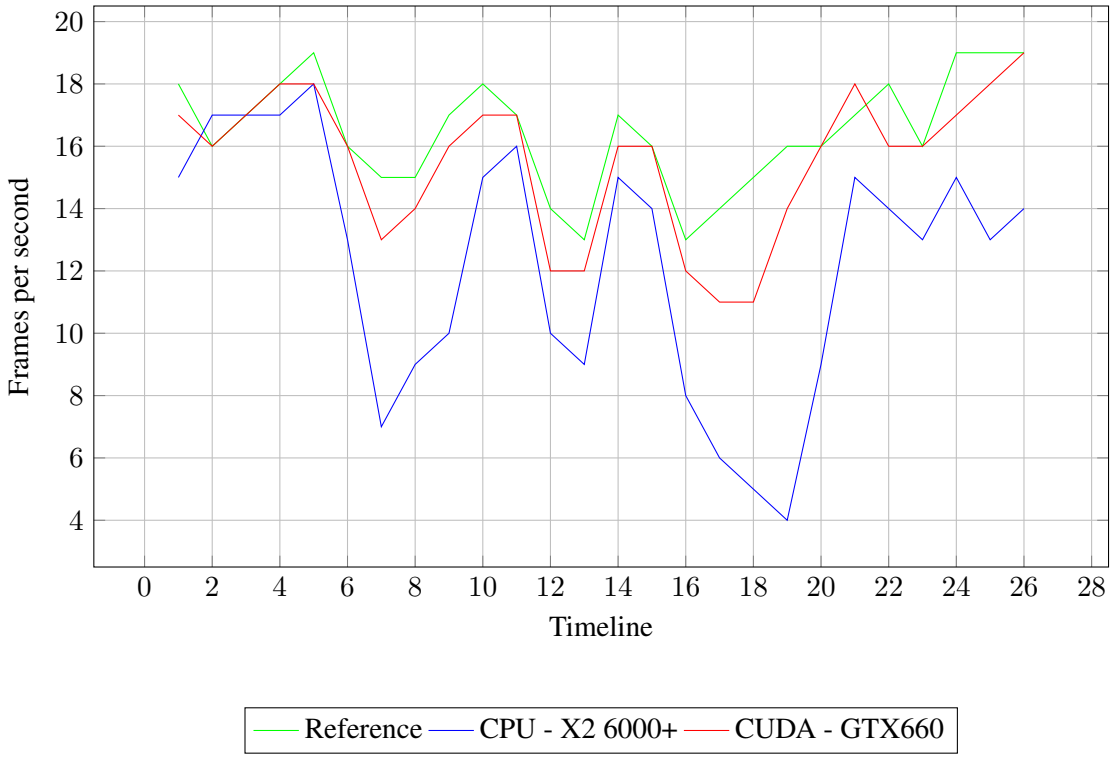


Figure 6.10: Performance achieved while moving through the *Catacombs* data set with 10 selection octrees. Reference traverses the scene without any selection test.

To show the maximum frame rate that is possible to achieve with Scanopy during the traversal of the camera path, a traversal through the scene without any selection octree tests is done. This reference is presented as green line in both Figure 6.10 and Figure 6.11.

The second test continues to use the selection octrees presented in Table 6.9 and uses the benchmarking system from Table 6.3. Figure 6.10 shows the achieved frame rate while the user moves through the scene along the predefined path. As can be seen, the frame rate of the CUDA-based method falls off only a little compared to the reference camera traversal which does no octree test at all. It is nor possible for the CPU-based method to keep the frame rate all the time at the same level than the CUDA-based method. It shows also some heavy performance drops. Those do not occur with the CUDA-based method. To extend this test, 10 more selection octrees have been added. The camera path through the scene remained the same. Now, the newly loaded points are tested against 20 selection octrees and the results are presented in Figure 6.11. The result for the CUDA-based method are similar to the previous test with 10 selections. However, the CPU-based method shows over all a lower frame rate with similar performance drops at the same place during the traversal of the scene like before.

As can be seen in Figure 6.10 and 6.11, the CUDA and the CPU version are sometimes faster than the reference solution. This can occur when it is not necessary to execute the point marking. This is the case when newly loaded MNO nodes do not belong to any selected region

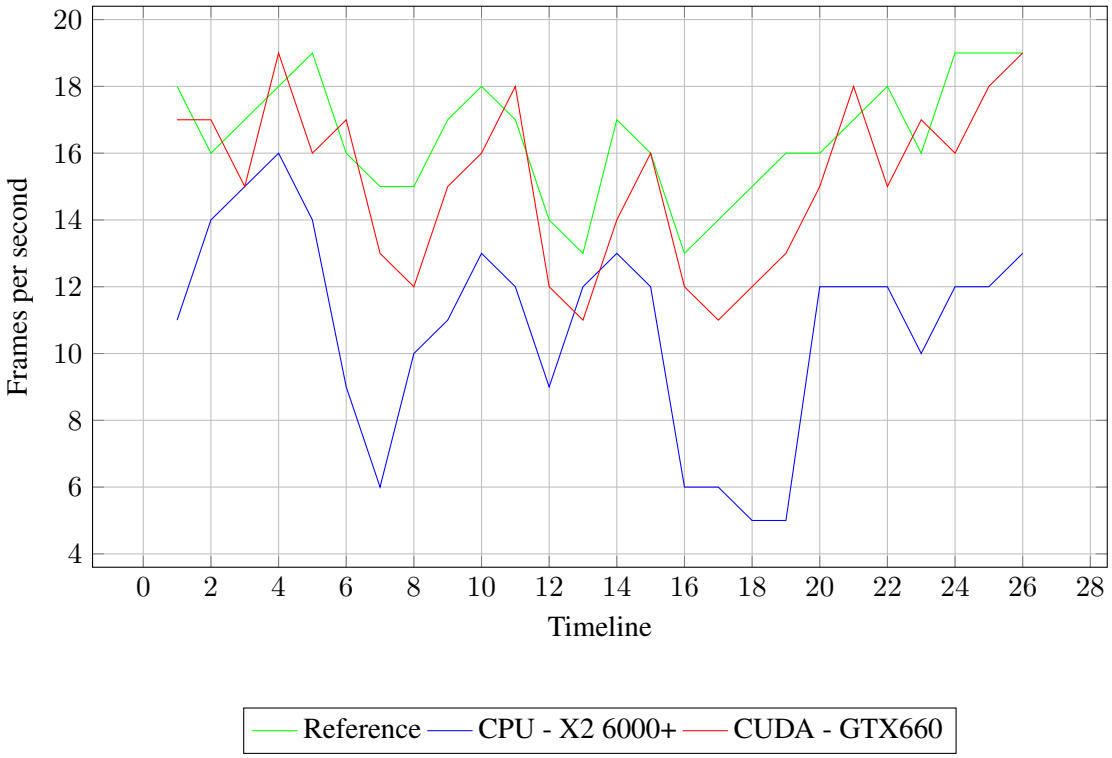


Figure 6.11: Performance achieved while moving through the *Catacombs* data set with 20 selection octrees. Reference traverses the scene without any selection test.

of all available selection octrees. It can be tested within 0.01 milliseconds whether the marking step can be skipped by testing if the volume of the MNO node itself falls into any selected octree node. Only if this is the case, the single points must be tested against the available selection octrees to distinguish between selected and unselected points. When the marking is not executed for any newly loaded MNO nodes of a given frame, only the time required to load the points from the hard disk to the graphics card memory comes into account. This means that the traversal with selection test and the reference traversal perform the same tasks when no newly loaded MNO nodes must be tested against the selection octrees. The camera path ensures that the same MNO nodes are loaded at a given point in time during the traversal of the path. However, the required time to load a specific MNO nodes can slightly vary between consecutive camera path traversals. This difference in time required to load the MNOs from the hard disk can change the overall frame rate between 1-2 frames per second. Therefore, the marking methods can be slightly faster than the reference solution at some points in time when no point marking needs to be done during this time.

As can be seen in both Figure 6.10 and Figure 6.11, the CUDA-based marking method is able to test points against 10 or even 20 different selection octrees at the same time without a severe drop in performance compared to the reference traversal. On the other hand, when the CPU-based method is used, the frame rate collapses several times. When 20 different selection

octrees are used, this problem is even higher.

CHAPTER 7

Conclusion

The aim of this thesis was the integration of an annotation system into the point cloud renderer Scanopy.

To link the annotations with the point cloud, a point-highlighting method has been proposed. The existing point-selection method running on the CPU was not able to work with a reasonable amount of selections such that more than 10 annotations could be used. Therefore, a different approach had to be created. It was assumed that an improved selection solution on the CPU would not provide the required performance to work with a large amount of annotations. The GPU with their capabilities of massive parallelism seemed to be an alternative to the existing CPU-based method.

A point-selection and marking method completely running on the GPU has been developed. As shown in Chapter 6, the performance was improved for both tasks. The point-selection method is at least 5 times faster than the corresponding CPU-based method, while the marking-method, which was responsible for low frame rates when several selections must be marked at the same time, runs up to 20 times faster using the GPU-based method. The simulated movement through a scene has show, that the CUDA-based method is able to handle 20 point selections with a small loss in performance compared to a traversal that does not have to process any point selection at all. The CPU-based method is not able to work with several annotations with attached selection octrees. It already shows some severe performance problems when 10 point selections are used. This drop in performance is also present when the amount of point selection increase to 20. On the other hand, the new CUDA-based approach is able to handle at least 20 active annotations with attached CUDA-based selection octree. This is considered to be enough for the annotation system in Scanopy because the selection octree for an annotation has only to be evaluated if the annotation is active. Additionally, to preserve the overview on the scene there should not be more than this amount of annotations visible at the same time.

7.1 Future Work

The work of this thesis consists of two parts which lead to different improvements. The main focus of the annotation system integrated into Scanopy is to visualize the additional information such that the user gains deeper knowledge regarding the visualized object represented as point cloud. Although a guidance system was integrated to lead the viewer to points of interest, other possibilities to search for specific information are not available.

Such a search function can be useful especially when the number of annotations in a scene increases. Due to the possibility to attach text documents and images to each annotation such a search function could also allow to retrieve specific data out of an annotation or get all information related to a given topic from all annotations. As it is already possible to add descriptions to each image, such a search function could also provide the possibility to search for the content of images if the descriptions cover the content of the images well.

On the technical part of this thesis, which covers the point-selection and point-marking methods responsible to establish a link between the point cloud and the annotations, a few improvements could be useful.

As shown in the results, most time during the construction of the selection octree is spent sorting the different Morton codes. It is not only the most time consuming step during the construction process, it is also used twice in a selection step. A faster sorting method could provide a reasonable performance improvement. Currently radix sort is used, which is assumed to be the fastest sorting algorithm in CUDA. However, the usage of Morton codes makes it necessary to use at least 60 bits of information per Morton code. As the complexity of radix sort depends not only on the number of elements but also on their length, another sorting algorithm could provide a better performance. The merge sort proposed by Satish et al. [Satish et al., 2009] does not lose as much performance as it is the case for the radix sort when the keys that have to be sorted get longer. The merge sort itself is slower than the radix sort when sorting 30-bit keys and there is no comparison available for keys that are use 60 bits or more. Therefore, it is not clear if a switch to merge sort improves the performance when longer keys have to be sorted.

The second step that can be covered is the issue that Scanopy can only perform a selection on point clouds with a color attribute because it currently uses the alpha channel of the color to distinguish between points that have been selected and those that have not been selected by the brush. Generally, this is not a problem since most point clouds have color as an attribute of their points. If the color attribute does not exist, an additional attribute, which is used to distinguish between selected and unselected points, can be useful to provide the selection method for colorless point clouds too.

The last aspect that can be improved is the performance issue on the latest CUDA capable devices from Nvidias 600 series onwards. As shown in Chapter 6, both the selection and the marking of points with the newer GTX 660 requires at least twice as long than the older GTX 570. Although the number of CUDA cores doubled between both graphics cards and Nvidia claims a higher performance for the newer card, the results could not verify the improved technical specifications. A possible adoption of the selection and marking algorithm can be useful, such that it better fits the newer architecture. However, it is not clear if this is possible at all since other CUDA-based algorithm face this performance issue as well.

Bibliography

Prekshu Ajmera, Rhushabh Goradia, Sharat Chandran, and Srinivas Aluru. Fast, parallel, gpu-based construction of space filling curves and octrees. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, page 10. ACM, 2008.

Daniel G. Aliaga, Elisa Bertino, and Stefano Valtolina. Decho - a framework for the digital exploration of cultural heritage objects. *J. Comput. Cult. Herit.*, 3(3):12:1–12:26, February 2011. ISSN 1556-4673. doi: 10.1145/1921614.1921619. URL http://doi.acm.org/10.1145/1921614.1921619.

Nancy Amato, Ravishankar Iyer, Sharad Sundaresan, and Yan Wu. A comparison of parallel sorting algorithms on different architectures. Technical report, College Station, TX, USA, 1998.

K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM. doi: 10.1145/1468075.1468121. URL http://doi.acm.org/10.1145/1468075.1468121.

Sean Bechhofer, Leslie Carr, Carole Goble, Simon Kampa, and Timothy Miles-Board. The semantics of semantic annotation. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 1152–1167. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-00106-5. doi: 10.1007/3-540-36124-3_73. URL http://dx.doi.org/10.1007/3-540-36124-3_73.

Jeroen BéDorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the gpu processor. *J. Comput. Phys.*, 231(7):2825–2839, April 2012. ISSN 0021-9991. doi: 10.1016/j.jcp.2011.12.024. URL http://dx.doi.org/10.1016/j.jcp.2011.12.024.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975. ISSN 0001-0782. doi: 10.1145/361002.361007. URL http://doi.acm.org/10.1145/361002.361007.

Guy E Blelloch. Prefix sums and their applications. 1990.

Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, March 1996. ISSN 0001-0782. doi: 10.1145/227234.227246. URL http://doi.acm.org/10.1145/227234.227246.

Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, SPAA '91, pages 3–16, New York, NY, USA, 1991. ACM. ISBN 0-89791-438-4. doi: 10.1145/113379.113380. URL http://doi.acm.org/10.1145/113379.113380.

I. Boada, I. Navazo, and R.l Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 35(2):185–197, April 2011.

Ian Buck and Tim Purcell. A toolkit for computation on gpus. *GPU Gems*, pages 621–636, 2004.

Shi-Kuo Chang and Arding Hsu. Image information systems: where do we go from here? *Knowledge and Data Engineering, IEEE Transactions on*, 4(5):431–442, 1992. ISSN 1041-4347. doi: 10.1109/69.166986.

Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann, 2013.

Lily Díaz, Markku Reunanen, Blanca Acuña, and Atte Timonen. Imanote: A web-based multi-user image map viewing and annotation tool. *J. Comput. Cult. Herit.*, 3(4):13:1–13:11, April 2011. ISSN 1556-4673. doi: 10.1145/1957825.1957826. URL http://doi.acm.org/10.1145/1957825.1957826.

Oxford English Dictionary. Oxford english dictionary online. *Mount Royal College Lib., Calgary*, 14, 2004.

Jianbin Fang, A.L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225, 2011. doi: 10.1109/ICPP.2011.45.

M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph.*, 14(3):124–133, July 1980. ISSN 0097-8930. doi: 10.1145/965105.807481. URL http://doi.acm.org/10.1145/965105.807481.

Irene Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, December 1982a. ISSN 0001-0782. doi: 10.1145/358728.358741. URL http://doi.acm.org/10.1145/358728.358741.

Irene Gargantini. Linear octtrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20(4):365–374, December 1982b. ISSN 0001-0782.

M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, and V. Volkov. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27, 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.57.

Enrico Gobbetti and Fabio Marton. Layered point clouds. In *Proceedings of the First Eurographics conference on Point-Based Graphics*, SPBG'04, pages 113–120, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. ISBN 3-905673-09-6. doi: 10.2312/SPBG/SPBG04/113-120. URL http://dx.doi.org/10.2312/SPBG/SPBG04/113-120.

Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

David Hilbert. Ueber die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web: Grundlagen*. Springer London, Limited, 2008.

Qihang Huang, Zhiyi Huang, P. Werstein, and M. Purvis. Gpu as a general purpose computing resource. In *Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on*, pages 151–158, 2008. doi: 10.1109/PDCAT.2008.38.

G.M. Hunter. *Efficient Computation and Data Structures for Graphics*. Princeton University, 1978. URL http://books.google.at/books?id=Vd5EPwAACAAJ.

R. Kadobayashi, J. Lombardi, M.P. McCahill, H. Stearns, K. Tanaka, and A. Kay. 3d model annotation from multiple viewpoints for croquet. In *Creating, Connecting and Collaborating through Computing, 2006. C5 '06. The Fourth International Conference on*, pages 10–15, 2006. doi: 10.1109/C5.2006.3.

Rieko Kadobayashi, Julian Lombardi, Mark P. McCahill, Howard Stearns, Katsumi Tanaka, and Alan Kay. Annotation authoring in collaborative 3d virtual environments. In *Proceedings of the 2005 international conference on Augmented tele-existence*, ICAT '05, pages 255–256, New York, NY, USA, 2005. ACM. ISBN 0-473-10657-4. doi: 10.1145/1152399.1152452. URL http://doi.acm.org/10.1145/1152399.1152452.

Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12, pages 33–37, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association. ISBN 978-3-905674-41-5. doi: 10.2312/EGGH/HPG12/033-037. URL http://dx.doi.org/10.2312/EGGH/HPG12/033-037.

David Koller, Bernard Frischer, and Greg Humphreys. Research challenges for digital archives of 3d cultural heritage models. *J. Comput. Cult. Herit.*, 2(3):7:1–7:17, January 2010. ISSN 1556-4673. doi: 10.1145/1658346.1658347. URL http://doi.acm.org/10.1145/1658346.1658347.

Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980. ISSN 0004-5411. doi: 10.1145/322217.322232. URL http://doi.acm.org/10.1145/322217.322232.

Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. Gpu gems 2. chapter 37: Octree textures on the gpu, 2005.

Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to algorithms*. The MIT press, 2009.

Thomas Lewiner, Vinícius Mello, Adelailson Peixoto, Sinésio Pesco, and Hélio Lopes. Fast generation of pointerless octree duals. *Computer Graphics Forum*, 29(5):1661–1669, 2010. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2010.01775.x. URL http://dx.doi.org/10.1111/j.1467-8659.2010.01775.x.

David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM. doi: 10.1145/1103900.1103933. URL http://doi.acm.org/10.1145/1103900.1103933.

Wei Ma, Yizhou Wang, Ying-Qing Xu, Qiong Li, Xin Ma, and Wen Gao. Annotating traditional chinese paintings for immersive virtual exhibition. *J. Comput. Cult. Herit.*, 5(2):6:1–6:12, August 2012. ISSN 1556-4673. doi: 10.1145/2307723.2307725. URL http://doi.acm.org/10.1145/2307723.2307725.

Donald Meagher. Geometric Modeling Using Octree Encoding. *Computer Graphics an Image Processing*, 19:129–147, 1982.

Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

Nvidia. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Nvidia, 2009.

Nvidia. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. Nvidia, 2012.

Nvidia 2013. Tuning cuda applications for kepler. URL https://www.clear.rice.edu/comp422/resources/cuda/html/kepler-tuning-guide/index.html. Accessed: 2013-09-19.

J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '84, pages 181–190, New York, NY, USA, 1984. ACM. ISBN 0-89791-128-8. doi: 10.1145/588011.588037. URL http://doi.acm.org/10.1145/588011.588037.

Charles Poynton. *Digital video and HD: Algorithms and Interfaces*. Access Online via Elsevier, 2012.

Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352. ACM Press/Addison-Wesley Publishing Co., 2000.

Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-50255-0.

Hanan Samet. Spatial data structures. In *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addison-Wesley, 1995.

Hanan Samet and Robert E. Webber. Hierarchical data structures and algorithms for computer graphics. part i. *IEEE Computer Graphics and Applications*, 8(3):48–68, May 1988a. ISSN 0272-1716. doi: 10.1109/38.513. URL http://dx.doi.org/10.1109/38.513.

Hanan Samet and Robert E. Webber. Hierarchical data structures and algorithms for computer graphics part ii. *IEEE Computer Graphics and Applications*, 8(4):59–65, 67–75, July 1988b. ISSN 0272-1716. doi: 10.1109/38.7750. URL http://dx.doi.org/10.1109/38.7750.

Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. doi: 10.1109/IPDPS.2009.5161005. URL http://dx.doi.org/10.1109/IPDPS.2009.5161005.

Claus Scheiblauer. Hardware-Accelerated Rendering of Unprocessed Point Clouds. Master's thesis, Vienna University of Technology, 2006.

Claus Scheiblauer and Michael Wimmer. Cultural heritage: Out-of-core selection and editing of huge point clouds. *Comput. Graph.*, 35(2):342–351, April 2011. ISSN 0097-8493. doi: 10.1016/j.cag.2011.01.004. URL http://dx.doi.org/10.1016/j.cag.2011.01.004.

Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004)*. 2004.

Eun-Joo Sin, Yoon-Chul Choy, and Soon-Bum Lim. Study on 3d annotation in virtual space for collaboration. In *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 01*, CSE '09, pages 382–387, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3823-5. doi: 10.1109/CSE.2009.259. URL http://dx.doi.org/10.1109/CSE.2009.259.

Henry Sonnet, Sheelagh Carpendale, and Thomas Strothotte. Integrating expanding annotations with a 3d explosion probe. In *Proceedings of the working conference on Advanced visual*

interfaces, AVI '04, pages 63–70, New York, NY, USA, 2004. ACM. ISBN 1-58113-867-9. doi: 10.1145/989863.989871. URL http://doi.acm.org/10.1145/989863.989871.

Ching-Lung Su, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang, and Kuo-Hsuan Wu. Overview and comparison of opencl and cuda technology for gpgpu. In *Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on*, pages 448–451, 2012. doi: 10.1109/APCCAS.2012.6419068.

Hideyuki Tamura and Naokazu Yokoya. Image database systems: A survey. *Pattern Recognition*, 17(1):29 – 43, 1984. ISSN 0031-3203. doi: http://dx.doi.org/10.1016/0031-3203(84)90033-5. URL http://www.sciencedirect.com/science/article/pii/0031320384900335. <ce:title>Knowledge Based Image Analysis</ce:title>.

Hung Truong, F. Boochs, A. Habed, and Y. Voisin. A knowledge-based approach to the automatic algorithm selection for 3d scene annotation. In *Information Science, Signal Processing and their Applications (ISSPA), 2012 11th International Conference on*, pages 225–230, 2012. doi: 10.1109/ISSPA.2012.6310550.

P. Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 372–381, 2003. doi: 10.1109/EMPDP.2003.1183613.

Michael S Warren and John K Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21. ACM, 1993.

Chih-Hao Yu, Tudor Groza, and Jane Hunter. High speed capture, retrieval and rendering of segment-based annotations on 3d museum objects. In *Proceedings of the 13th international conference on Asia-pacific digital libraries: for cultural heritage, knowledge dissemination, and future creation*, ICADL'11, pages 5–15, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24825-2. URL http://dl.acm.org/citation.cfm?id=2075271.2075278.

Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 712–721, New York, NY, USA, 1991. ACM. ISBN 0-89791-459-7. doi: 10.1145/125826.126164. URL http://doi.acm.org/10.1145/125826.126164.

Dengsheng Zhang, Md. Monirul Islam, and Guojun Lu. A review on automatic image annotation techniques. *Pattern Recognition*, 45(1):346 – 362, 2012. ISSN 0031-3203. doi: http://dx.doi.org/10.1016/j.patcog.2011.05.013. URL http://www.sciencedirect.com/science/article/pii/S0031320311002391.