Interactive Rendering of 3D Geovirtual Environments using Image-Based Services

Lars Schneider

Computer Graphics Systems Group

Hasso-Plattner-Institut

A thesis submitted for the degree of

Master of Science (MSc) in IT-Systems Engineering

under the supervision of

Prof. Dr. rer. nat. habil. Jürgen Döllner
Dipl. Inform. Dieter Hildebrandt

2011 April

Abstract

A 3D Geovirtual Environment (3DGeoVE) is a powerful instrument for visualizing, exploring, analyzing, and managing complex multi-dimensional geospatial data in a clear and comprehensive way. Due to the continuing shift towards ubiquitous computing, there is a strong desire to employ this instrument within a mobile environment.

However, innovation is inhibited by lower processing power and memory capacity on mobile devices compared to desktop machines, making it difficult to store and process 3DGeoVE on them.

The approach of this thesis transcends current limitations by employing a network-based rendering service which consumes camera and viewport parameters to generate appropriate G-Buffers for use by the mobile client. These G-Buffers are continuously requested by the mobile client and transformed into a point cloud which is stored and rendered locally on the mobile client using a dynamic out-of-core multi-resolution data structure.

G-Buffers are a compact format for transmitting point data with varying levels of detail over the network. The mobile client adaptively controls the level of detail, enabling interactive rendering at 15 frames per second. On the client side, the rendering algorithm employs a compact point data structure consuming approximately 6.4 byte per point including position, color, and surface normal. This structure is transferred without modifications from mass storage to the main memory, and on to GPU memory, saving 50% of the memory bandwidth compared to conventional methods.

As a consequence, applications can leverage 3DGeoVE to visualize geospatial data even on devices with limited hardware capabilities.

Zusammenfassung

3D-geovirtuelle Umgebungen (3DGeoVE) sind ein leistungsfähiges Instrument um komplexe, multidimensionale räumliche Daten klar und verständlich darzustellen, zu erkunden, zu analysieren und zu verwalten. Mit dem aktuellen Wandel zur allgegenwärtigen Informationsverarbeitung entsteht ein starkes Interesse dieses Instrument auch auf mobilen Geräten zu benutzen.

Mobile Geräte haben allerdings, verglichen mit durchschnittlichen Arbeitsplatzrechnern, eine geringere Verarbeitungsgeschwindigkeit und Speichergröße, was die Visualisierung von 3DGeoVE erschwert.

Die Arbeit nähert sich diesem Problem mit Hilfe eines netzwerkbasierten Rendering-Dienstes, welcher Kameraparameter und Sichtbereich als Eingabe nimmt und entsprechende G-Buffer der Szene generiert. Diese G-Buffer werden fortlaufend vom mobilen Klienten angefragt und in eine Punktwolke transformiert, welche dann mit Hilfe einer dynamischen out-of-core multiresolution Datenstruktur lokal gespeichert und gerendert wird.

G-Buffer erweisen sich als effizientes Format um Punktdaten mit unterschiedlichem Detailgrad auszutauschen. Weiterhin kann der mobile Klient den Detailgrad dynamisch anpassen und so eine Renderinggeschwindigkeit von 15 Bildern pro Sekunde garantieren. Der Algorithmus nutzt eine kompakte Datenstruktur, welche, inklusive Position, Farbe und Normale, circa $6,4$ Byte pro Punkt benötigt. Diese Datenstruktur wird ohne Modifikation vom Massenspeicher zum Hauptspeicher zum Grafikspeicher transferiert. Folglich muss nur 50% der Datenmenge konventioneller Methoden über den Datenbus gesendet werden.

Im Ergebnis können 3D-geovirtuelle Umgebungen auch auf Geräten mit technisch limitierter Hardware visualisiert werden.

To my family.

Acknowledgements

I would like to thank my supervisors Prof. Dr. Jürgen Döllner and Dipl. Inform. Dieter Hildebrandt for their support and mentoring. Our conversations clarified my thinking and inspired me to new ideas on the topic.

Furthermore, I want to thank Keith Klemba, Prof. Dr. Mark Billinghurst, Prof. Dr. Patrick Baudisch, and Dipl. Inform. Matthias Trapp for giving me the opportunity to work with state-of-the-art mobile devices and explore their 3D capabilities.

I am also grateful to my family and friends for their support. In particular I want to thank Birte, Martin, Ronny, Sebastian and Torsten for social support and Bernd, Kirsten, Rico and Robert for proof-reading.

Contents

List of Figures **vii**

List of Tables **ix**

Glossary **xi**

1 Introduction **1**

1.1 Motivation 2

1.2 Document Structure 4

2 Fundamentals and Related Work **5**

2.1 Point-Based Rendering 5

2.1.1 Point-Based Rendering for Static Models 6

2.1.2 Point-Based Rendering for Dynamic Models 9

2.2 Distributed Visualization 10

2.3 Mobile Hardware and Software 13

2.3.1 Memory Constraints 13

2.3.2 CPU Attributes 14

2.3.3 GPU Attributes 14

3 Requirements and Concept **17**

3.1 Term Definitions 17

3.2 Requirements 18

3.3 Out-Of-Scope / Non-Requirements 19

3.4 System Overview 20

4 Design and Implementation **23**

4.1 General Optimizations . . . 23
4.2 Cross-Platform Compatibility . . . 24
4.2.1 Programming Language . . . 24
4.2.2 Abstraction of OS Functions . . . 25
4.2.3 Computer Architecture . . . 25
4.2.4 Graphics Library . . . 25
4.3 WVS Data Acquisition . . . 26
4.3.1 WVS Request . . . 26
4.3.2 Image Layer to Point Cloud . . . 29
4.3.3 Request Trigger . . . 31
4.4 Memory Allocation . . . 31
4.4.1 Malloc . . . 31
4.4.2 Memory Pool . . . 32
4.5 Spatial Data Structure . . . 34
4.5.1 Two Layer Octree Structure . . . 35
4.5.2 Insertion . . . 40
4.5.2.1 `QuantPoint` Generation . . . 40
4.5.2.2 `QuantPoint` LOD Generation and Insertion . . . 42
4.5.3 Out-Of-Core Representation . . . 45
4.5.3.1 Working Set . . . 46
4.5.3.2 Backing Store . . . 46
4.6 Rendering . . . 46
4.6.1 Input-Mode vs. Quality-Mode . . . 47
4.6.2 Visibility Culling . . . 48
4.6.2.1 Hierarchical View Frustum Culling . . . 50
4.6.2.2 Occlusion Culling . . . 51
4.6.3 LOD Selection . . . 52
4.6.3.1 Node Selection . . . 52
4.6.3.2 Quality Factor . . . 54
4.6.4 Data Processing for GPU . . . 55
4.6.4.1 Preparation . . . 55
4.6.4.2 Transmission . . . 57

4.6.4.3 Shader Processing 59

5 Evaluation **63**

5.1 Performance 63

5.1.1 Memory Allocation 63

5.1.2 Image Decoding 65

5.1.3 3D Point Acquisition 65

5.1.4 Vertex Submission 69

5.1.5 Rendering Speed 69

5.2 Visual Limitations 72

5.2.1 Insufficient Precision 72

5.2.2 Imprecise Quantization 72

5.2.3 Incomplete LODs 74

5.2.4 Surface Cracks 74

5.2.5 Incomplete Visualization 76

5.3 Requirement Fulfillment 76

5.3.1 Data and Data Structure Requirements 76

5.3.2 Rendering Requirements 77

5.3.3 Platform Requirements 77

5.3.4 Hardware Requirements 77

6 Conclusions **79**

6.1 Contributions 79

6.2 Discussion 80

6.3 Future Work 81

References **83**

List of Figures

1.1 Visualization of a 3DGeoVE on a mobile device. 1

2.1 Visualization of a point cloud [Lab10]. 5
2.2 Two levels of the LDC tree (in 2D) [PZvBG00]. 7
2.3 Hierarchical binary voxel grid (in 2D) [BWK02]. 8
2.4 Hierarchical structure of the 2D example from Figure 2.3 [BWK02]. . . 8

3.1 FMC Petri Net Diagram of the high level program flow. 20
3.2 FMC Block Diagram of the software architecture. 22

4.1 UML Class Diagram of the `WVSAdapter`. 27
4.2 UML Sequence Diagram of the WVS Data Acquisition. 28
4.3 3D point reconstruction using a depth image value. 29
4.4 UML Class Diagram of the Memory Pool. 33
4.5 Memory layout of a `MemoryBin`. 33
4.6 Voxels in two consecutive levels in an octree with 8^3 grid nodes. 36
4.7 UML Class Diagram of the octree data structure. 37
4.8 Memory layout of a `QuantPoint` 38
4.9 FMC Petri Net Diagram of the point insertion process. 43
4.10 FMC Petri Net of the rendering loop. 47
4.11 Flowchart of the octree traversal. 49
4.12 Illustration of view frustum and occlusion culling (in 2D). 51
4.13 Illustration of octree regions (in 2D). 56
4.14 Memory layout of an octree node position for GPU submission. 57
4.15 Approximate calculation of the splat size. 61

LIST OF FIGURES

5.1 Performance comparison between `malloc` and my memory pool implementation on iPad and iPhone 3GS. 64
5.2 Performance comparison between JPEG and PNG image decoding on iPad and iPhone 3GS. 66
5.3 Time to request, download, and process WVS G-Buffers on iPad and iPhone 3GS. 68
5.4 Submission and rendering of 1,000,000 points on iPad and iPhone 3GS. 69
5.5 Exploration of an urban city scene on the iPhone 3GS. 70
5.6 Exploration of an urban city scene on the iPad. 71
5.7 Screenshots depicting artifacts due to grid quantization. 73
5.8 Screenshots depicting imprecise quantization. 73
5.9 Screenshots depicting different surface cracks. 75
5.10 Schematic visualization of surface cracks due to altered camera parameter. 75

List of Tables

2.1 Unofficial specifications of Apple iPhone 3GS and iPad. 13

5.1 Image file size and decoding time using the image depicted in Figure 5.2 on iPad and iPhone 3GS. 65

5.2 G-Buffer resolution, number of pixels, download size, number of allocated octree nodes, number of allocated `QuantPoints`, and the percentage of client side processing of the urban city scene depicted in Figure 5.3 on iPad and iPhone 3GS. 67

Glossary

2D	Two Dimensions
3D	Three Dimensions
CPU	Central Processing Unit
CRS	Coordinate Reference System
FMC	Fundamental Modeling Concepts
FOV	Field of View
GB	Gigabyte
GeoVE	Geo Virtual Environment
GPS	Global Positing System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IP	Internet Protocol
JPEG	Joint Photographic Experts Group
KB	Kilobyte
LDC	Layered Depth Cube
LDI	Layered Depth Image
LIFO	Last In, First Out
LOD	Level of Detail
LRU	Least Recently Used
MB	Megabyte
MPEG	Moving Picture Experts Group
NP	Non-Deterministic Polynomial
OGC	Open Open Geospatial Consortium
OOP	Object-Oriented Programming
OS	Operating System
OSG	Open Scene Graph
PBR	Point-Based-Rendering
PDA	Personal Digital Assistant
PNG	Portable Network Graphics
RGB	Red Green Blue
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
STL	Standard Template Library
TBDR	Tile Based Deferred Rendering
TCP	Transmission Control Protocol
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
VBO	Vertex Buffer Object
VFP	Vector Floating Point
VRML	Virtual Reality Modeling Language
W3DS	Web 3D Service
WVS	Web View Service

1

Introduction

In 1963, Ivan Sutherland presented one of the first computer graphics applications called *Sketchpad* [Sut63]. It was state-of-the-art technology and capable of rendering simple 2D models based on point and line primitives. Graphics technology, from it's modest beginnings to current day, has evolved to the point to which it is possible to render entire computer generated worlds, as well as details as fine as a blade of grass.

Today's graphics technology can be used to visualize virtual 3D city models as a specialized type of 3D geovirtual environments (3DGeoVEs). 3DGeoVEs serve to visualize, explore, analyze, and manage geo-referenced data and information. 3DGeoVEs are increasingly used as general purpose medium for communicating spatial information in mobile applications such as car navigation, city marketing, tourism, and gaming [TSL+11].

Figure 1.1: Visualization of a 3DGeoVE on a mobile device.

This thesis discusses an approach to visualize massive 3DGeoVE on mobile devices (Figure 1.1). The result is an adaptive rendering system that delivers the visual representation of 3DGeoVEs at interactive frames on mobile devices, in spite of their limited processing power and memory capacity.

1.1 Motivation

According to MacEachren et al. [MEH+99] and based on Heim [Hei98, pp. 162-167, 171] 3DGeoVEs are characterized by four factors that contribute to their *virtuality*:

Immersion: The sensation of "being in" the environment.

Interactivity: The change of the viewpoint on the environment.

Information intensity: The level of detail (LOD) used to present the environment.

Intelligence of objects: Context sensitive "behavior" of the environment.

These characteristics are hard to implement due to the inherent problem that 3DGeoVE data is generally complex. This complex data is massive, multidimensional, heterogeneous, distributed, imprecise, and incomplete. Consequently, a device that visualizes 3DGeoVEs needs to cope with all these difficulties.

Even though mobile hardware improves constantly, today's devices can neither store nor process complex data. Their memory capacity and processing power is considerably lower than the latest desktop hardware due to the fact of their mobile form. Furthermore, because of their reliance on battery power, mobile devices face the additional challenge of needing to be energy efficient, which is not necessary relevant to desktop hardware. Neglecting the hardware constraints, storing entire 3DGeoVEs locally on mobile devices might not be desired by the industry either. The capturing and preparation of the data is laborious. Consequently, the data is valuable and must be protected against misuse or theft. Furthermore, a frequent requirement is up-to-date data and maintenance might be complicated with locally installed data.

The Open Geospatial Consortium (OGC) [Ope10], leading the development of standards for geospatial web services, tries to address this problem by means of the *Web 3D Service* (W3DS) draft [QK05]. This web service provides a 3DGeoVE piece by piece over the network in 3D formats such as X3D [Web11] or KML [Wil08]. The 3D models are generally represented as triangle meshes and an inherent problem is to identify the pieces relevant for the client in a LOD that is suitable for the client. The LOD computation for triangle meshes turns out to be strongly $\mathcal{NP}$-hard [HMHW97], thus is not applicable for interactive processing.

Even if the geometry LOD can be computed, the W3DS approach remains difficult because of texturing. Every mesh requires an appropriately sized texture depending on the view settings to avoid unnecessary data transmission and aliasing artifacts due to minification [AMHH08, p. 161].

Hagedorn et al. [HHD10a] and Hildebrandt [Hil10] propose a different approach and address these difficulties with an image-based solution using a web service called *Web View Service* (WVS) [Hag10]. The WVS consumes camera, view port, and various style parameters and returns rasterized images representing, e.g., the color, depth, and surface normal data layer of the 3DGeoVE. These layers are called G-Buffers [ST90]. In the scenario proposed by Hagedorn et al., a client requests the color layer of a view and displays the returned image. Consequently, the client is not in need of processing any 3D data which considerably lowers the hardware requirements. However, the WVS request and response requires time due to transmission bandwidth and network latencies. These latencies can prevent the former mentioned interactivity and immersion requirement of 3DGeoVEs. This is particularly the case in the context of mobile devices because they have to cope with low bandwidth and high latencies due their wireless network connections.

This thesis aims to improve the interactivity and immersion characteristics in a WVS based scenario continuing the ideas of Hildebrandt et al. [HHD10b]. The following approach is evaluated: Instead of color-only images the client requests a set of G-Buffers, containing color, depth, and normal information, from the WVS. This data is used to incrementally build a point cloud based on this data locally on the client. This point cloud is rendered on the device using *point-based rendering* (PBR) techniques [GP07].

PBR has a number of advantages as already realized by Levoy and Whitted [LW85] in 1985. They proposed the notion of points as the universal primitive instead of triangles for two reasons. First, points as individual *samples* of a surface of a 3D geometry have no connectivity between each other and therefore refinement is essentially resampling. Second, every screen has a fixed screen size and screen resolution. Thus the maximum amount of information, in other words the number of pixels n, that can be displayed is also fixed. That means with a perfect resampling and selection strategy maximal n points need to be processed in every frame. This is especially beneficial for mobile devices, because they have usually a small n due to their small screens.

The resampling and selection of points is complicated. This thesis will show a reasonable approximation which enables state-of-the-art mobile devices to render scenes of high geometric complexity at interactive frame rates.

1.2 Document Structure

The document in structured as follows. Chapter 2 discusses briefly PBR and distributed rendering techniques found in literature. Furthermore, the characteristics of mobile hardware are outlined using Apple iPhone and iPad devices as examples. Chapter 3 introduces terms used in the thesis and lists the requirements for the proposed solution. Based on these requirements the design of the system is introduced at a high level. All core components and their interaction with each other are explained. Chapter 4 discusses the design and the implementation of the system components in depth. The WVS data acquisition, the client's point data structure, and the client's rendering algorithm, amongst others, are also discussed. Chapter 5 evaluates the solution with respect to performance, visual artifacts, and requirement fulfillment. Chapter 6 summarizes the contributions and outlines future work based on this thesis that are potentially worth pursuing.

2

Fundamentals and Related Work

In this section, related work and fundamentals are presented. Relevant research is introduced and classed into *point-based rendering* and *distributed visualization*. Moreover, the characteristics of mobile hardware are discussed.

2.1 Point-Based Rendering

PBR requires two main steps [GP07, p. 313]. First, out of all points representing the entire geometry of a scene, a small set of surface sample points is selected to approximate the geometry in the desired view. The density of the selected points aims to match the resolution of the resulting image as close as possible to provide (preferably) one sample point for each pixel. Second, based on the sample points the final image is reconstructed (Figure 2.1). This step might require special considerations to avoid artifacts (e.g., aliasing or holes) due to the rasterization process.

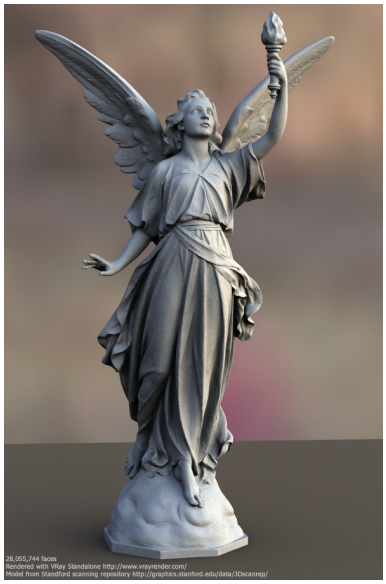

Figure 2.1: Visualization of a point cloud [Lab10].

A comprehensive survey of real-time PBR is given in [GP07]. The research can be divided into PBR for static models and PBR for dynamic models since this distinction has significant implications on the data structure and the rendering performance.

2.1.1 Point-Based Rendering for Static Models

Static models have an immutable data structure at rendering time that is generated in a preprocessing step. Modifications on such a data structure at run-time are considered computational heavy and therefore are not applicable for interactive applications [WBB+07]. Nevertheless the research on PBR started with static models and has therefore many important insights as discussed below.

Rusinkiewicz and Levoy [RL00] present QSplat, a multi resolution point rendering system for large meshes. QSplat performs a preprocessing step that transforms the mesh into a kD-tree [Ben75] forming a hierarchical point data structure based on a bounding sphere hierarchy. During rendering, this data structure is used for visibility culling and LOD control.

Each node of the tree represents a sphere. This sphere contains a center position, a radius, a normal cone, and a color. Position encoding relative to parent nodes, quantization and look-up tables are used to reduce the memory footprint of a node to 48 bits.

A recursive rendering algorithm traverses the tree in breadth-first order. In the case that a node is not visible, tested via view frustum and backface culling, all child nodes of that node are skipped. If a node is visible, all child nodes are recursively traversed. If a node is a leaf node or an inner node with sufficient visual precision, the node is rendered and all child nodes of that node are skipped. A node is rendered according to the projected diameter of it's sphere.

The paper by Rusinkiewicz and Levoy [RL00] as well as the paper by Pfister et al. [PZvBG00] are especially influential to this thesis since they started the modern research of PBR [GP07, p. 16].

Pfister et al. [PZvBG00] present rendering based on surface elements called *surfels*. A surfel is a zero-dimensional n-tuple with shape and shade attributes that locally approximate an object surface.

In a preprocessing step, geometric objects (including textures) are sampled as surfels by using Layered Depth Images (LDIs) as proposed by Shade et al. [SGHS98]. Lischinski and Rappaport [LR98] define three orthogonal LDIs as a Layered Depth

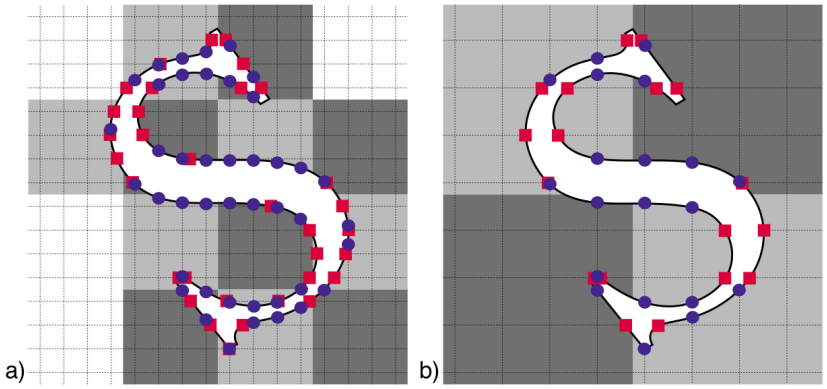

Figure 2.2: Two levels of the LDC tree (in 2D) [PZvBG00]. - Every LDI contains k^2 layered depth pixels (k = 4 in this LDC tree). Neighboring LDCs are differently shaded and empty LDCs are white.

Cube (LDC). Pfister et al. store these LDCs in a hierarchical space-partitioning octree structure and call it *LDC tree*.

LDCs on higher levels of the octree are constructed by subsampling their children by a factor of two (Figure 2.2). To avoid texture aliasing artifacts, each surfel contains three pre-filtered texture samples (surfel mipmap). A surfel containing these mipmap levels, a normal, a depth value, and a material has a memory footprint of 160 bits.

The tree is traversed with a recursive rendering algorithm that performs view frustum and backface culling on the LDCs. The remaining LDCs are selected depending on their projected screen space size. All rendering is done in software.

The two layer data structure composed of a coarse octree with fine-grained LDC nodes is similar to the proposed data structure in this thesis. Moreover I adopted the idea to quantize all samples - even on the leaf level of the octree.

Botsch et al. [BWK02] present an octree based data structure that defines every node as regular 2x2x2 binary voxel grid. A bit value indicates if a voxel is present or not. Thus every node is a sampled representation of it's bounding cube similar to [PZvBG00; WBB+07].

Noteworthy is that only occupied voxels are stored using a byte code to represent the occupied nodes in the child (Figure 2.3 and 2.4). 2x2x2 voxels allow 256 permutations that can be encoded with 2^3 bit = 1 byte. As a result 2.67 bits per sample are required

to encode the position. Given the fact that certain byte codes appear more often than others, entropy encoding [Sal00] is used to lower this value to 2 bits.

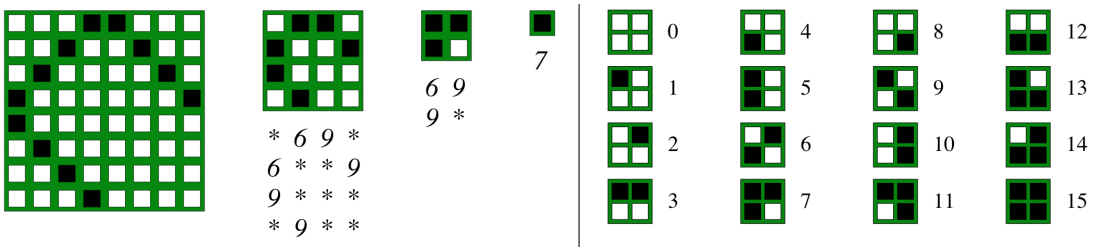

Figure 2.3: Hierarchical binary voxel grid (in 2D) [BWK02]. - An 8x8 sample grid is hierarchically encoded (left) by using 4-bit codes (right). Empty samples "*" are encoded implicitly by following the zero-bits on the next coarser level. In this example, the 64-bit sample grid is encoded with 40-bit. The 3D case will have many more empty samples and therefore a higher compression.

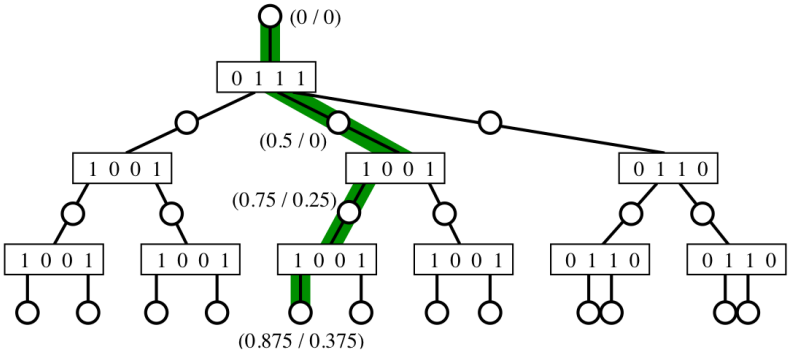

Figure 2.4: Hierarchical structure of the 2D example from Figure 2.3 [BWK02]. - Every step in the path from the root to a leaf node adds another precision bit.

At rendering time the final sample positions are incrementally reconstructed during octree traversal. The authors evaluated depth-first and breath-first traversal. Although depth-first traversal is more memory efficient, breath-first traversal enables progressive reconstruction. Furthermore, the nodes are traversed front to back based on their distance to the viewpoint. This lowers the probability that a later sample overwrites an earlier one, thus the image reconstruction appears faster.

Although the position encoding proposed in the paper by Botsch et al. has the lowest memory footprint known to literature, it is not applicable to this thesis. The efficiency of the data structure inherently depends on the static and pointer-less memory

layout. In spite of that, I adopted the idea to traverse the octree nodes based on their distance to the viewpoint.

Duguet and Drettakis [DD04] introduce a flexible PBR scheme suitable for a range of devices including mobile devices. They use regular grids as a data structure and evaluate different subdivisions per dimension. The data is preprocessed and limited to the main memory of the rendering device. All rendering is done in software.

The paper especially discusses the application on mobile devices as this thesis does. Therefore they faced the same difficulties with respect to processing power and memory consumption.

2.1.2 Point-Based Rendering for Dynamic Models

Dynamic models have a mutable data structure at run-time. Insert, delete, and modify operations can be performed on the data at any time. This implies that the entire data structure is not known upfront and therefore preprocessing can not be applied.

Zwicker et al. [ZPKG02] present Pointshop 3D, the first paper regarding PBR for dynamic models. Although their focus is on editing concepts rather than large point clouds and efficient rendering.

Pauly et al. [PKKG03] use kD-trees [AMHH08, p. 651] to render mutable point clouds (up to 1,000,000 points) efficiently. However, kD-trees require a costly rebalancing operation after modification. To mask this delay, a cache is employed during editing. At the end of an editing session, this cache is integrated into the kD-tree data structure.

Wand et al. [WBB+07] address interactive editing of very large data sets that exceed the limits of main memory. Their implementation is able to perform real-time editing mostly independent of data set complexity (up to 63 GB in their evaluation) by leveraging a point-based multi-resolution data structure and out-of-core algorithms. In fact, the complexity of a dynamic update operation is proportional to the amount of geometry affected as long as there is some spatial coherence (e.g., actual data blocks accessed within a local time window in main memory).

In order to efficiently utilize current graphics hardware Wand et al. employ a two layer data structure similar to [PZvBG00]. The first layer creates a coarse spatial subdivision by employing an octree data structure. The second layer is created within a node of the octree. All inner nodes establish a fixed sample spacing k^3 grid to store a quantized, down sampled point cloud of the bounding cube of the corresponding node. The points of these grids are managed in a hash data structure to access individual points fast. Only leaf nodes store original data points.

Required nodes only are kept in main memory during rendering or editing by using a least recently used (LRU) queue. Nodes that are required but not yet in main memory are scheduled to be fetched while a representation with a lower resolution is used in the mean time. Fetch operations are done in a separate thread to hide I/O latencies.

A recursive rendering algorithm performs a depth-first traversal of the hierarchy until the minimum projected point spacing is below a fixed threshold. View frustum culling is performed on the octree nodes although it is less accurate due to the coarse structure. The image is reconstructed by scaling point primitives according to the point sample spacing similar to [RL00].

I adapted the idea of the two layer data structure composed of an octree and a quantization grid within the nodes. Moreover, I also used a LRU queue to organize in-core nodes, multi threading to hide I/O latencies, and depth-first octree traversal. This thesis differs from Wand et al.'s paper in that the leaf node data is also quantized in my implementation.

2.2 Distributed Visualization

3DGeoVEs data sets exceed the memory capacity of the latest mobile devices and consequently can neither be stored nor processed on these devices. A client/server architecture is a possible solution to this problem. The server is expected to have enough resources to store and process the entire data set. The (mobile) client requests only relevant parts of the data from the server and is expected to have enough resources to store and process this subset. In the following, the advantages and disadvantages of client/server solutions are outlined and important related work is discussed.

Advantages

- Data is permanently stored on the server only. Thus clients have to update their required subset and therefore are expected to have always the latest data.
- Data maintenance is centralized on the server.
- Data can be protected against misuse by centralized access and usage restrictions on the server.
- Expensive data can be sold with different business models (e.g., sold piece by piece or monthly usage fee).
- Data processing can be balanced between server and client.

Disadvantages

- Data transmission is limited to the available network bandwidth and reliability.
- Intelligent data caching/prediction is necessary in order to provide smooth and free interaction in a 3DGeoVE.

Rusinkiewicz and Levoy [RL01] incorporated view-dependent progressive transmission into their QSplat system that permits the viewer to look at and interact with partially-downloaded models. In case parts of the model are not present on the client, a present low-resolution model is rendered and the missing parts are requested from the server. Data is added until the entire model is stored on the client.

A coarse availability mask is used to determine which parts of the data are present, desired, or already requested from the server. Nodes that are not present are requested via HTTP [FGM+99]. The requests are managed in a priority queue determined by the projected screen size and position of the node. Based on experimental results the authors do not employ data prefetching.

Color coding is used to provide visual cues to users for content that is still being downloaded. A tool called "magnifying glass" can be used to increase the request priority for a certain region of the screen and consequently visualize details in that area faster.

The solution presented in the paper transmits points and their position explicitly and independently. In contrast to that, a WVS G-Buffer as used in the implementation presented in this thesis encodes the position implicitly. The former approach has a

higher memory footprint but can employ an availability mask which is not possible with the latter. Another disadvantage of WVS G-Buffers is that their samples have to be reprocessed before inserted into the client data structure.

Chang and Ger [CG02] propose a system in that a server renders traditional polygon based 3D models and transmits the resulting color image, an according depth image, and the viewing matrix that was used for rendering to the mobile client. With the help of an image-based rendering technique called "3D Warping" [McM97], the mobile client is able to render different views based on this data.

Only minor view changes can be rendered without noticeable errors due to occlusion and limited 3D information offered by the original image data. The authors argue that this technique can hide network latency and can offer interactive frame rates on mobile clients with weak processing power and small network bandwidth.

The image-based geometry transmission presented in the paper by Chang and Ger is similar to the WVS based approach discussed in this thesis. However, they employ warping techniques to accommodate for different views whereas my implementation performs PBR.

Lamberti and Sanna [LS07] propose a system where a cluster of computer render a 3D scene that is afterwards transmitted as MPEG [LG91] video stream to an arbitrary number of mobile clients. The video stream is tailored in terms of resolution and quality to match the client's screen size and available network bandwidth.

The client in turn transmits commands to change the camera settings on the server and thus the image that is received from the server.

In comparison to independent WVS G-Buffers used in this thesis, a MPEG video stream can exploit similarities between consecutive frames and therefore lower the data to be transmitted. However, in the case that the network connection breaks or degrades, the proposed solution can no longer function, as the client replays only the video stream and does not store any data.

Google [Dan09] offers a browser based service called "Street View with Smart Navigation" to explore urban road systems. The user is able to navigate using forward and backward arrows along the roads on a predefined path within cities. Given that the

Vendor and Name	Apple iPhone 3GS	Apple iPad
Available since	2009	2010
Operating System	iOS (based on UNIX)	iOS (based on UNIX)
Graphics Library	OpenGL ES 2.0	OpenGL ES 2.0
CPU	ARM Cortex-A8 at 600 MHz	ARM Cortex-A8 at 1000 MHz
Level 1 Cache	32/32 KB Instruction/Data Cache	32/32 KB Instruction/Data Cache
Level 2 Cache	256 KB	640 KB
GPU	Power VR SGX 535	Power VR SGX 535
Memory	256 MB DRAM	256 MB DRAM
Storage Capacity	16/32 GB flash memory	16/32/64 GB flash memory
Display	480x320 px at 163 ppi	1024x768 px at 132 ppi

Table 2.1: Unofficial specifications of Apple iPhone 3GS and iPad.

scenery is visualized as cube map, rotation and zoom can be rendered with interactive frame rates on the client (web browser with Adobe Flash support [Ado10] or native Apple iPhone/iPad maps app). On movement, a new cube map is downloaded and rendered. The transition between two cube maps is smoothed using 3D warping.

Due to the predefined cube maps, the viewpoint placement is also predefined and therefore limited. However, the transmission of cube maps is similar to the transmission of view dependent G-Buffers provided by the WVS.

2.3 Mobile Hardware and Software

The implementation is supposed to run on desktop and mobile hardware. Since mobile hardware has typically only a fraction of the resources of desktop hardware (Table 2.1), all performance improvements are made towards mobile hardware. The Apple iPhone 3GS and the Apple iPad running iOS are exemplary mobile target devices and used for the implementation. Significant differences to desktop machines in hardware, operating system (OS) and available graphics library are discussed in the following.

2.3.1 Memory Constraints

On mobile devices, memory is typically a scarce good. In the case of the iPhone or the iPad, the iOS operating system ensures that at any point in time enough resources are available to keep the device responsive. To achieve this, iOS oversees the memory

consumption of every running app. In case an app uses more than a specific memory threshold, iOS sends a *memory warning* message to the app [CH10, p. 229]. The app is supposed to free memory in that event. If the app continues to claim memory, a second memory warning is sent and the app is shut down by iOS shortly after. This is necessary because iOS does not support swapping main memory to disk. Thus the physical main memory is a hard constraint [All10, p. 50].

2.3.2 CPU Attributes

The ARM Cortex-A8 CPU used in iPhone 3GS and iPad is a single core, 32-bit reduced instruction set computer (RISC) with Harvard memory architecture and two modes of operation called *ARM-Mode* and *Thumb2-Mode*. Thumb2-Mode operates with an even more reduced instruction set leveraging 16-bit instructions to increase code density. This mode is the default and leads to smaller executables because of the reduced memory footprint for certain operations [Rid10, p. 372]. However, ARM-Mode is considered faster for math-intense applications and enables an extended instruction set to leverage the NEON pipeline for executing SIMD (Single Instruction Multiple Data) and VFP (Vector Floating Point) instructions [All10, p. 389].

2.3.3 GPU Attributes

The Power VR SGX GPU [POW09] used in iPhone 3GS and iPad supports vertex and fragment shaders using a unified shader architecture. That implies both kinds of shaders are executed on the same hardware, called *Universal Scalable Shader Engine* (USSE), to maximize core utilization independent of the vertex/fragment processing ratio. The USSE supports multiple precisions (`lowp`, `medium`, and `highp`) defined as hints in the shader program to perform calculations only with necessary precision.

In contrast to desktop GPU's, the Power VR SGX employs a method called *Tile Based Deferred Rendering* (TBDR) [AMHH08, p. 870ff.]. The viewport is divided into tiles, rectangular regions, which are rendered individually. The fragment processing is deferred until all geometry data and state required for a frame has been completely submitted and the visibility of every surface defined by this geometry has been calculated. Consequently, only those parts of surfaces that are visible in the final image will be computed by the fragment pipeline.

The Power VR SGX is designed for low power consumption. The power consumption is more than two orders of magnitude below that of high-end desktop graphics cards [POW09, p. 5]. As a negative side effect, these power saving techniques lower the performance of the GPU.

The GPU is accessed via OpenGL ES. OpenGL ES is an application programming interface (API) for advanced 3D graphics targeted at handheld and embedded devices such as cell phones, personal digital assistants (PDAs), consoles, appliances, vehicles, and avionics [MGS09]. OpenGL ES 2.0 is derived from the OpenGL 2.0 [SWND07] specification and implements a programmable graphics pipeline. The most signifiant difference between OpenGL ES and OpenGL is a lower memory footprint due to the removal of redundant functions.

The shaders running within the GPU are implemented using OpenGL ES Shading Language (GLSL ES). GLSL ES is based on the OpenGL Shading Language (GLSL) version 1.20 [Sim09] and augmented with precision qualifiers to be used by the USSE as mentioned above [MGS09, p. 96].

3

Requirements and Concept

This chapter defines important terms that are used in the thesis, lists all requirements for the proposed system, and presents a high level overview of the general program flow as well as all relevant software components. In addition, certain possible requirements in particular are marked out and called "non-requirements".

3.1 Term Definitions

A number of terms in PBR are not used consistently in literature. Therefore, all terms are defined as they are used in this thesis.

Definition 1 (3D Point) *"A 3D point is a location in space, in a 3D coordinate system." [PVM+07, p. 29]*

Definition 2 (Point Cloud) *A point cloud is an unstructured collection of three-dimensional point data [GP07, p. 436]. The points have no inherent connection between each other [AMHH08, p. 533].*

Definition 3 (Voxel) *"'Voxel' is short for 'volumetric pixel', and each voxel represents a regular volume of space." [AMHH08, p. 502]*

Definition 4 (Sphere) *"A three-dimensional surface, all points of which are equidistant from a fixed [center] point is called sphere." [Kar03, p. 21]*

Definition 5 (Surfel) *"A surfel is a zero-dimensional n-tuple with shape and shade attributes that locally approximate an object surface." [PZvBG00]*

3.2 Requirements

This section lists all requirements for the proposed software system.

Data and Data Structure Requirements

Req01 **WVS Data Source** - The WVS is the only data source for all geometry rendered by the system. Originating from color, depth, and normal images provided by the WVS as well as their camera parameter, the client must consecutively generate 3D points.

Req02 **Dynamic Data Structure** - The point cloud data structure on the client must be dynamic to allow a consecutive insertion of new 3D points originating from WVS at run-time.

Req03 **Main Memory Independent Data Structure** - The point cloud data structure, hierarchy as well as 3D point data, must be able to grow beyond main memory capacity.

Req04 **Spatial Extend** - The point cloud data structure must be able to cover a spatial extend of 1000 km^2 (e.g., area of the city of Berlin) with a resolution of less than 15 cm.

Req05 **Minimal Storage Cost** - The client must not waste memory due to padding or alignment. The memory footprint of a single 3D point with color must be lower than the 16 byte per point required in [RD10].

Rendering Requirements

Req06 **Visual Quality** - The client must render the WVS based point cloud hole free as long as data for the geometry is present.

Req07 **Interactive Rendering** - While interactively exploring the scene, the client must render with more than 15 frames per second as stated in [AMHH08, p. 1].

Req08 **Response Time** - The client must respond to user input at least after 0.2 seconds.

Platform Requirements

Req09 **Cross-Platform Compatibility** - The client must support the iOS and Mac OS X platform.

Req10 **Respect Memory Restricted Environments** - The client must respect platforms with hard memory restrictions such as iOS. In these cases the application must not crash as a consequence of allocating too much memory.

Hardware Requirements

Req11 **Hardware Exploitation** - The client must exploit all available hardware such as a Graphical Processing Unit (GPU) or a floating point coprocessor to speed up rendering.

Req12 **Energy Consumption** - The client must not perform unnecessary rendering in order to use as less energy as possible.

3.3 Out-Of-Scope / Non-Requirements

In the following "non-requirements" are listed. These requirements are explicitly out-of-scope and must not be fulfilled.

NReq01 **Animated Geometry** - The client is not required to support animated geometry.

NReq02 **Transparency** - The client is not required to support transparent shading.

3.4 System Overview

This section presents a high level overview of the general program flow as well as all relevant software components and the connections between them.

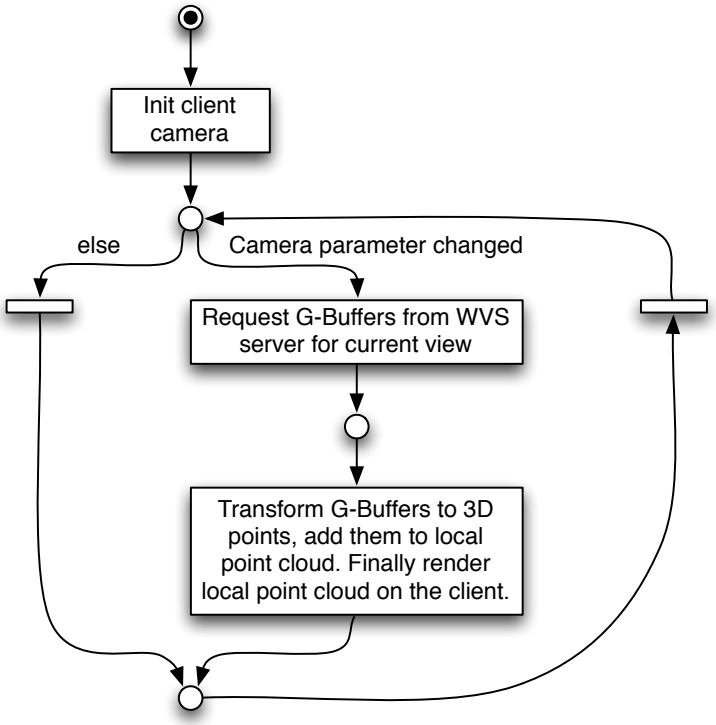

Figure 3.1: FMC Petri Net Diagram of the high level program flow. - The client initializes the local camera. Afterwards the request and render loop starts. The client requests a new G-Buffer as soon as the camera parameter have changed. The G-Buffer is transformed into 3D points. These points are added to the local point cloud. Finally, the local point cloud is rendered for the current view.

The `AppCore` (Figure 3.2) is the central component of the system and responsible for the main loop (Figure 3.1) that performs WVS G-Buffer acquisition, G-Buffer to 3D point transformation, and point rendering. The `WVSAdapter` executes G-Buffer acquisition by issuing `WVSRequests`. A `WVSRequest` contains camera and viewport settings (camera position, view direction, field of view, viewport size), as well as a set of strings that identify the requested G-Buffer layer(s) and their encoding. The `WVSRequest` is transformed into a URL [BLMM94] and executed as HTTP GET request [FGM+99] on top of the TCP/IP [Ste94] network protocol. The server responds with

an HTTP-Multipart message in which every part contains a JPEG [Int92] or PNG [Bou97] encoded image according to the request. The `Response Decoder` splits the multipart message, decodes the images using `libJPEG` [Ind10] and `libPNG` [Gle10], and generates new 3D points based on the camera settings, the color G-Buffer, and the depth G-Buffer.

The `Octree` manages the client's persistent data structure. This covers the integration of new 3D points into the data structure as well as the means to traverse the data structure for rendering. The data structure is built upon nodes that are linked with each other. Every node retains a number of 3D points. The memory for these persistent data structures is provided by the `MemoryPool`. As stated in Section 2.3.1, certain clients, in particular mobile clients based on iOS, only have a limited amount of memory. Thus, if the `MemoryPool` can not issue any more memory, the `Octree` has to release memory by swapping unused nodes (including their associated 3D points) from memory to a swap file with the help of the `BackingStore`.

All utilized platform specific functions with the exception of the native platform graphics/user interface are accessed via the `APIFacade` to hide the platform specific implementation. The reason for the exception are significant differences in UI metaphors, human computer interaction techniques, and OpenGL context creation between mobile devices and desktop computers.

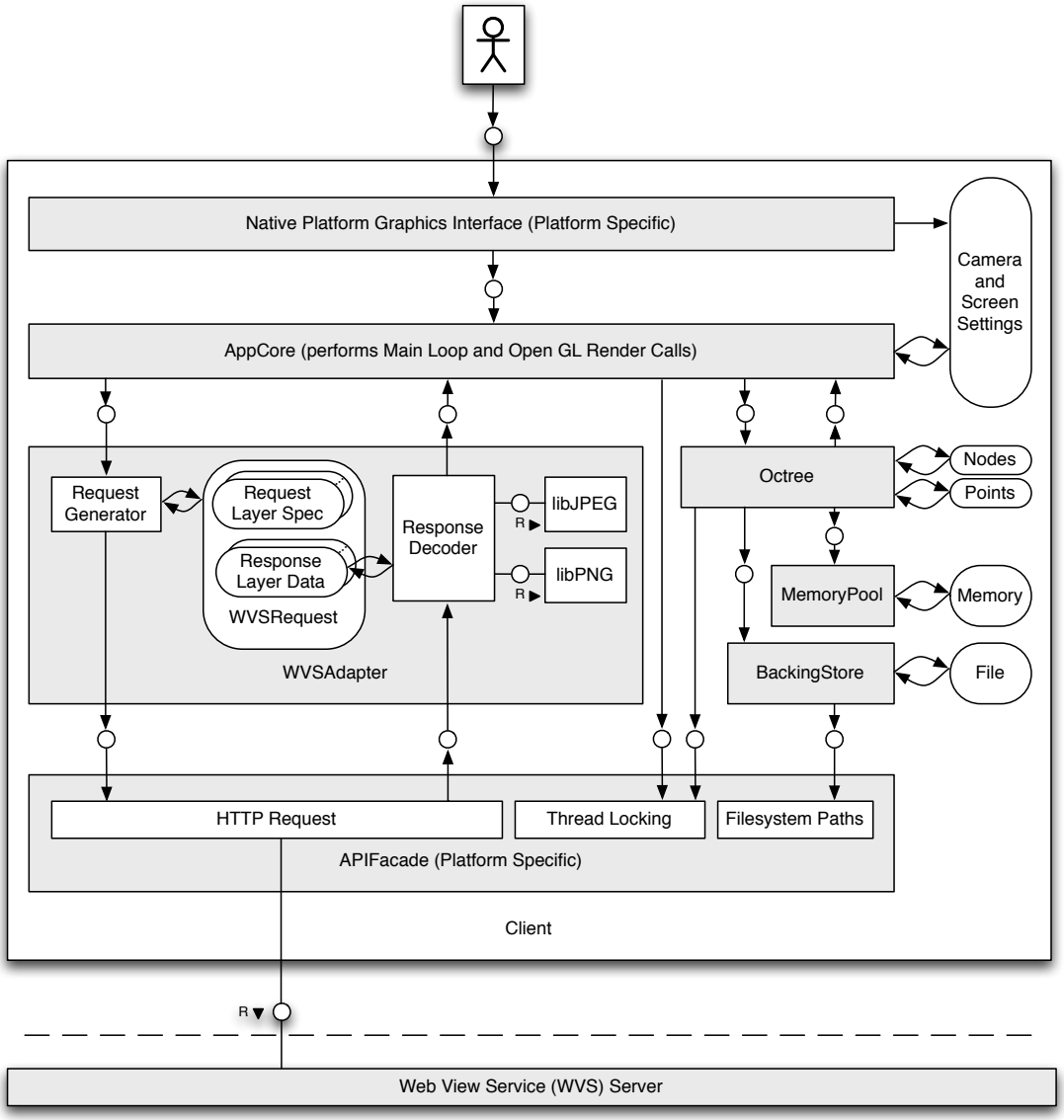

Figure 3.2: FMC Block Diagram of the software architecture. - The client acquires G-Buffer data from the server via `WVSAdapter`. The consequential 3D points are stored on the client using the `Octree` and `BackingStore` implementation. In addition, the `Octree` implementation is able to traverse the data structure and copy view relevant 3D points to a buffer. This buffer is transferred to the GPU and rendered on the screen via OpenGL calls in `AppCore`.

4

Design and Implementation

In the following, the design and implementation of all components introduced in the previous chapter are explained in detail. First, general optimizations are discussed which are applicable to the implementation of all components. Second, the means to implement a cross-platform solution are discussed. Third, data acquisition via WVS is explained in detail. Forth, memory allocation problems and a possible solution using a memory pool is presented. Fifth, the spatial data structure managing the 3D points is examined. Last, rendering of 3D points is described.

4.1 General Optimizations

As stated in Section 2.3, mobile hardware is very limited in terms of CPU/GPU performance, memory bandwidth, and memory capacity. Consequently, these resources must be used as efficiently as possible.

All performance-critical data structures are implemented as plain C types instead of convenient C++ STL (Standard Template Library) [MS95] types. The default STL memory manager is time and memory inefficient [Mey01, p. 54]. These drawbacks were recognized by the video game industry, specifically by the company Electronic Arts, and led to the development of an extended and partially redesigned replacement for the STL called *Electronic Arts Standard Template Library* (EASTL) [Ped07]. Unfortunately, the open source release of the EASTL in October 2010 was too late to be considered for this project. Consequently, plain C data structures and custom memory allocation were implemented (Section 4.4).

Memory within data structures usually cannot be accessed one bit at a time. It's rather accessed in buckets of the CPU's *machine-word* [Fra03, p. 26]. To improve the memory access efficiency, the compiler aligns data structures to the machine-word boundaries in the physical memory. This implies that if the size of a data structure is not a fraction of the machine-word then extra (otherwise unused) memory is added by the compiler. This procedure is called *padding*. Since memory is precious on mobile devices, all data structures are designed in a way that no padding is necessary. Furthermore, all data structures are designed to align with cache lines of the iPhone/iPad hardware as well as possible to minimize cache misses. On the Cortex-A8 CPU, a level 1 cache line is 32 bytes, and a level 2 cache line is 64 bytes.

All floating point operations on the CPU are executed with single-precision for two reasons. First, they require half the memory of double-precision variables and less cycles per instruction are necessary. Second, single-precision floating point operations can take best advantage of the NEON pipeline [ARM05, p. 6].

For GPU vertex submission Apple recommends 4 byte alignment [KR10, 24:42 min] [Rid10, p.367] and batching [Rid10, p.371] to enable efficient vertex shader processing. Moreover, arranging operations as *multiply and add* operations is beneficial because they can be executed in a single cycle [WSYN09, p. 89].

In addition, all GPU and CPU code is annotated with compiler optimization hints such as `const` [Mey97, p. 91], `inline` [Mey97, p. 137], and precision qualifiers [KR10, 33:06 min] where appropriate.

4.2 Cross-Platform Compatibility

According to requirement *Req09*, the client has to run on different platforms. The means to implement this requirement are discussed in the following.

4.2.1 Programming Language

C++ [Str00] was chosen as the programming language for three reasons: First, mature compilers exist for all relevant platforms (including iOS). Second, the language supports high-level features such as classes and inheritance as well as low-level features such as direct memory access. Third, code is directly executed on the hardware with no intermediate layer and therefore considered fast.

4.2.2 Abstraction of OS Functions

Network communication, thread spawning, thread synchronization, and file system access are, amongst others, low level operating system functions with a platform specific interface and implementation. To work with these functions in a platform-independent manner, they are grouped in a unified, platform agnostic interface called `APIFacade` using the *Façade* [GHJV95] design pattern. Every supported platform requires a custom `APIFacade` implementation.

4.2.3 Computer Architecture

Both 32-bit and 64-bit architectures are supported via preprocessor directives. 64-bit systems are capable of addressing more than 4 GB (= 2^{32} byte) main memory that certainly can be used for massive point clouds. However, since every pointer occupies 64-bit (= 8 byte) memory instead of 32-bit (= 4 byte) on these architectures, the amount of memory used by data structures coping with a large amount of pointers will increase rapidly.

To avoid problems due to different architectures, every integer was explicitly declared with its memory width using the `cstdint` header file of the C standard library [PL87].

4.2.4 Graphics Library

Although OpenGL ES is derived from OpenGL it is not a strict subset [MGS09, p. 3]. Thus a distinction between the two has to be made via preprocessor directives in code (Listing 4.1).

Listing 4.1: Preprocessor directives to determine OpenGL ES on the iOS platform.

```
1 #if (TARGET_IPHONE_SIMULATOR || TARGET_OS_IPHONE)
2   #import <OpenGLES/ES2/gl.h>
3   #import <OpenGLES/ES2/glext.h>
4   #define OPENGL_ES
5 #endif
6
7 #if (!TARGET_IPHONE_SIMULATOR && !TARGET_OS_IPHONE && TARGET_OS_MAC)
8   #include <OpenGL/glu.h>
9   #define OPENGL
10 #endif
```

These incompatibilities are present in the Shading Language as well. To run GLSL ES shader on pre GLSL 1.20 hardware precision hint keywords have to be added (Listing 4.2)

Listing 4.2: Precision hint keywords for pre GLSL 1.20 hardware.

```
#ifndef GL_ES
  #if (__VERSION__ < 120)
    #define lowp
    #define mediump
    #define highp
  #endif
#endif
```

4.3 WVS Data Acquisition

The WVS is the data source used in the implementation (*Req01*). This section explains how WVS data is requested and transformed into 3D points.

4.3.1 WVS Request

This subsection explains how WVS image layers (G-Buffers) are requested.

All WVS capabilities are encapsulated in the class `WVSAdapter` (Figure 4.1). On application start-up, one instance of the `WVSAdapter` class is created and initialized with the parameters of the 3DGeoVE on the WVS server in use. In particular the center, the Coordinate Reference System (CRS), and a conversion factor between WVS units and meters for the 3DGeoVE. These values are defined in `AppConfig.h` as `WVS_CENTER_POSITION`, `WVS_COORDINATE_REFERENCE_SYSTEM`, and `WVS_UNIT_TO_METER_FACTOR`.

On camera parameter change (Section 4.3.3), the `AppCore` performs a new point cloud request via the `WVSAdapter`. This is accomplished with the `generateRequest( camera, precision)` method. The `camera` parameter is a container for all camera, view, and frustum related values such as camera position, view direction, up vector, field of view (FOV), near plane, far plane, screen height, and screen width. The `precision` parameter determines the fraction of the request resolution. That implies a value of "one" requests the full resolution, a value of "two" half the resolution, and so forth.

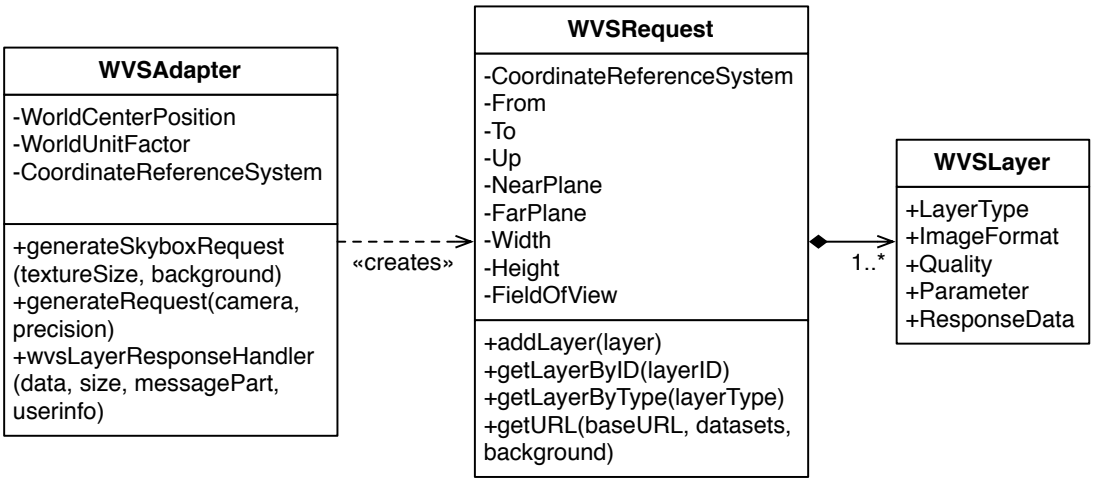

Figure 4.1: UML Class Diagram of the `WVSAdapter`. - The `WVSAdapter` creates `WVSRequest` objects that contain one or more `WVSLayer` objects which describe a G-Buffer. A `WVSLayer` specifies the requested data type, encoding, and a quality indicator for lossy compression.

On request, the creation of the camera's `from` and `to` vectors are scaled to WVS model units and translated to the WVS model center. Then, three `WVSLayer`, namely color, depth, and normal layer, are created and added to a `WVSRequest`. Finally, the `WVSRequest` is transformed into an URL (Listing 4.3) and sent to the WVS server.

Listing 4.3: Example WVS request URL with color, depth, and normal layer.

```
1  http://wvs-server/HttpGetHandler.ashx?
2  SERVICE=WVS&VERSION=0.0.1&REQUEST=GetView&CRS=EPSG:3068&
3  BBOX=1,1,1,1&DATASETS=__ALL__DATA__&STYLES=&SLD=TODO&
4  EXCEPTIONS=BLANK&BACKGROUNDCOLOR=0xFFFFFF&
5  FROM=1366.38,-1872.98,370.15&TO=1366.38,3127.02,370.15&
6  UP=0.00,-0.00,1.00&NEARPLANE=0.50&FARPLANE=5000.00&
7  FOVX=90.00&FOVY=73.73&WIDTH=768&HEIGHT=1024&
8  IMAGELAYERS=COLOR,DEPTH,NORMAL&
9  OUTPUTFORMAT=image/jpeg,image/png,image/png&
10 PARAMETERS=24Bit,32Bit,24Bit&
11 QUALITIES=95,100,100
```

The HTTP request is performed by the `APIFacade` via the `getResourceFromURL( url, callback)` function. The function returns immediately because `APIFacade` performs the actual request asynchronously in a separate thread. After the HTTP-

Multipart message was received from the server, the message is split into its parts and the callback method `wvsLayerResponseHandler()` is executed for each part. In this method, the PNG/JPEG image data of a layer is decoded using `libPNG/libJPEG` into an uncompressed eight bits per channel pixel array and stored in the according `WVSRequest` data structure (Figure 4.2).

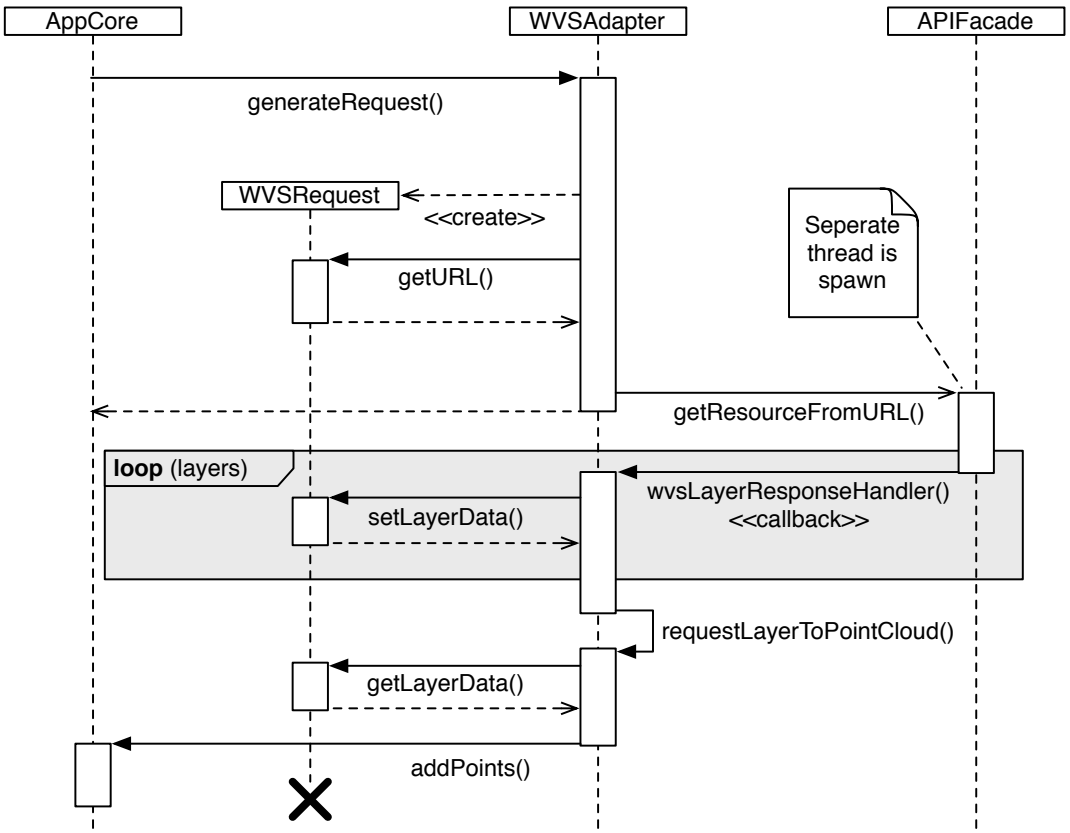

Figure 4.2: UML Sequence Diagram of the WVS Data Acquisition. - The `AppCore` requests new data for the current view from the `WVSAdapter`. The `WVSAdapter` creates an according `WVSRequest` and executes the consequential HTTP request via the `APIFascade` in a separate thread. As soon as the request returns, the `WVSAdapter` receives the G-Buffer data via callback. Within this callback the 3D points are generated and transferred to the `AppCore`.

4.3.2 Image Layer to Point Cloud

This subsection explains how WVS G-Buffers (image layers) are transformed into 3D points. A depth image, viewport parameter and camera parameter provide sufficient information to calculate a corresponding point cloud. All this information can be used to invert the graphics rendering pipeline (view transform, projection, clipping, screen mapping) [AMHH08, p. 16] for all pixels in the depth image.

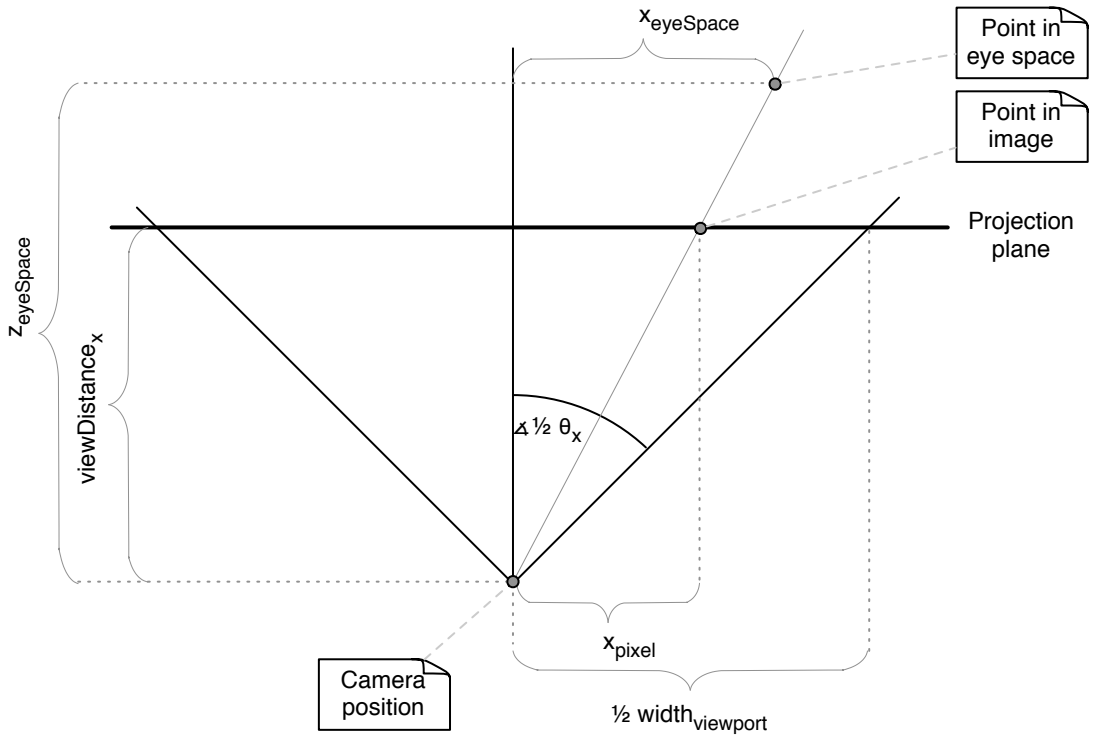

Figure 4.3: 3D point reconstruction using a depth image value. - With the help of the tangent trigonometric function and the concept of similar triangles the value $x_{eyeSpace}$ is calculated using camera FOV θ_x, $width_{viewport}$, and the depth value $z_{eyeSpace}$. Y value reconstruction is analogous and therefore omitted.

All image 2D points are located in a pixel raster on the projection plane within the bounds of the viewport (Figure 4.3). First, the distance between the projection plane and the camera called $viewDistance_{x/y}$ is calculated using the tangent trigonometric function, the camera FOV $\theta_{x/y}$, and the viewport size (Equation 4.1 and 4.2). Afterwards, $x_{eyeSpace}$ and $y_{eyeSpace}$, are respectively calculated using the

concept of similar triangles, $viewDistance_{x/y}$, the viewport size, and $z_{eyeSpace}$ which is the value stored in the depth image layer (Equation 4.3 and 4.4). The result is a point $P_{eyeSpace} = (x_{eyeSpace}\ y_{eyeSpace}\ z_{eyeSpace}\ 1)^T$ in homogeneous notation [AMHH08, p. 905].

$$viewDistance_x = \frac{0.5 * width_{viewport}}{tan(0.5 * \theta_x)} \tag{4.1}$$

$$viewDistance_y = \frac{0.5 * height_{viewport}}{tan(0.5 * \theta_y)} \tag{4.2}$$

$$x_{eyeSpace} = \frac{0.5 * width_{viewport} - x_{pixel}}{viewDistance_x} * z_{eyeSpace} \tag{4.3}$$

$$y_{eyeSpace} = \frac{0.5 * height_{viewport} - y_{pixel}}{viewDistance_y} * z_{eyeSpace} \tag{4.4}$$

The point $P_{eyeSpace}$ is multiplied with the inverse of the camera view matrix ${M_{view}}^{-1}$ to obtain $P_{worldSpace}$ (Equation 4.5). The inverse of M_{view} is not trivial since it is a 4x4 matrix and there is no simple formulae for this task [AMHH08, p. 903]. However, if only uniform scaling was done on the camera, then the upper 3x3 components represent a rotation-only matrix and the forth row a translation vector. Both components can be inverted independently. The 3x3 rotation matrix is inverted by using *Cramer's rule* [AMHH08, p. 903] and the translation vector is inverted by multiplying with -1.

$$P_{worldSpace} = {M_{view}}^{-1} * P_{eyeSpace} \tag{4.5}$$

Finally, $P_{worldSpace}$ is transformed to $P_{octreeSpace}$ by inverting the transforms done on WVS Request (Equation 4.6 and Section 4.3.1)

$$P_{octreeSpace} = {M_{WVSCenter}}^{-1} * {M_{WVSUnits}}^{-1} * P_{worldSpace} \tag{4.6}$$

The point is interpreted as a sphere covering the entire depth pixel that was used to calculate the point position. Therefore the pixel's edge length is approximated using similar triangles and used as sphere diameter (Equation 4.7).

$$edgeLength_{Pixel} = \frac{z_{eyeSpace}}{viewDistance_x} \tag{4.7}$$

4.3.3 Request Trigger

Every network request requires time due to latency and transmission. This time is especially significant on mobile wireless connections. Consequently, an appropriate request strategy is necessary to receive relevant data as fast as possible.

On every loop in the rendering thread, the time, t_{change}, since the last camera parameter change is calculated. If $t_{change} > t_{requestDelay}$ and no request was made yet, a new request with the current camera parameters is triggered. As mentioned above, every request is executed in a separate thread and the thread performs the WVS request, receives the data, and processes the results. These threads are organized in a LIFO (last in, first out) queue with a maximum length of two. If the queue is full and a new request is made, the oldest waiting request in the queue is canceled. This ensures only the most relevant data is requested and downloading/processing of different requests can be executed in parallel.

The insertion of 3D points into the data structure is costly (Section 4.5.2). Therefore, two different $t_{requestDelay}$ triggers exist. The first one is supposed to be fast (e.g., 100 ms after camera movement) and triggers only a low resolution request (e.g., ⅛ of the screen resolution) that generates a small number of coarse 3D points. If the camera parameters did not change until the second trigger (e.g., 1000 ms after camera movement) a full resolution request is triggered. Both values are defined in `AppConfig.h` as part of the `WVS_RELOAD_DELAY` array.

4.4 Memory Allocation

Memory efficiency is important for PBR in real world applications [BWK02]. This applies especially to mobile devices with hard memory restrictions (*Req10*). First, I will briefly explain how memory allocation works. Afterwards, I will outline common problems and a possible solution used in this project.

4.4.1 Malloc

In C/C++ the subroutine `malloc` [PL87] is used to perform dynamic memory allocation. Although it is part of the standard library, the implementation varies across different platforms. Thus, I will focus on the iOS/Mac OS X platform.

In iOS/Mac OS X all memory is allocated by the kernel in 4 KB virtual memory pages using the `mmap` syscall [Sin06, p. 951ff.]. These pages are requested and managed by `malloc` to offer more fine-grained memory allocations on top of them. To keep track of all occupied/free areas of these pages, `malloc` saves metadata (size and location) for every allocated block in use and any free space between blocks.

This strategy has a number of drawbacks. First, `malloc` pays no respect to iOS memory restrictions. This denotes `malloc` will successfully allocate memory beyond the restrictions imposed by iOS. However, as soon as the memory is accessed iOS will send a memory warning notification and eventually terminate the application (Section 2.3.1).

Second, `malloc` occupies memory for the former mentioned metadata. The size of the metadata per allocation is constant and therefore proportional higher for small memory blocks. Consequently, the allocation of a great number of small memory blocks (e.g., for a point cloud) requires a lot memory for metadata.

Third, `malloc` cannot promise a constant execution time because on every allocation a fitting memory block must be searched. The time to search increases due to memory fragmentation and memory fragmentation increases because of variable block sizes issued by `malloc`.

Fourth, to simplify metadata and the free block search `malloc` works internally with memory blocks of different predefined sizes. The smallest block size is 16 bytes for allocations between 1 byte and 496 bytes on 32-bit systems [Sin06, p. 956]. Consequently, small allocations such as 1 byte will always require 15 bytes of padding.

4.4.2 Memory Pool

The shortcomings of `malloc` can be solved with *memory pools* [Mey97, p. 46ff.]. A memory pool preallocates memory and issues only allocations of one specific size from this memory. Consequently, no additional memory is allocated that could cause a memory warning notification. Furthermore, no metadata for the allocation size is required and no fragmentation arises because the allocation size is constant for all allocations. However, I was not able to find a freely available lightweight memory pool implementation which runs on the iOS platform and respects its memory restrictions. Consequently, I developed a tailored memory pool implementation as discussed in the following.

The class `MemoryPool` implements the memory pool. An instance of this class has a number of `MemoryBins` (Figure 4.4). A `MemoryBin` is a bin for a certain constant allocation size and it points to a preallocated block of memory to accomplish this task.

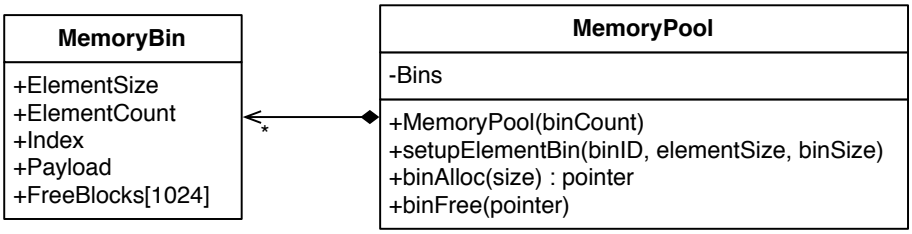

Figure 4.4: UML Class Diagram of the Memory Pool. - A memory pool consists of a number of memory bins. Each memory bin manages a preallocated memory block for a constant allocation size.

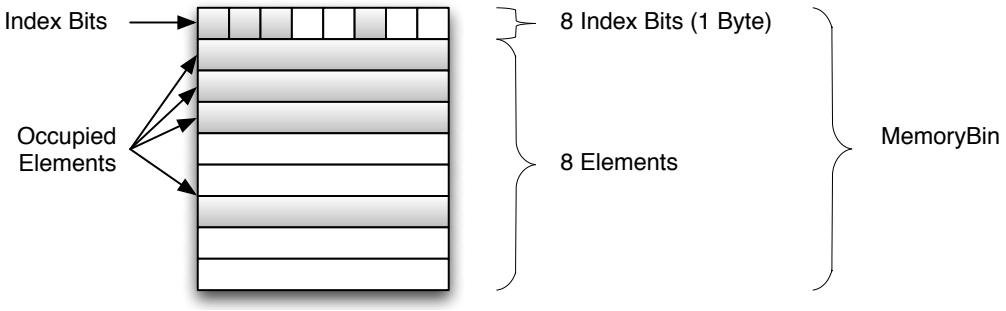

Figure 4.5: Memory layout of a `MemoryBin`. - Every Element occupies one index bit to determine if it is occupied or not.

The method `void setupElementBin(binID, elementSize, binSize)` of the class `MemoryPool` initializes a bin within the memory pool for a specific allocation size defined by the parameter `elementSize`. The entire amount of memory that is preallocated for the bin is defined by the parameter `binSize`. Based on these values the number of available elements is calculated. Every element has an overhead of one *index bit* to indicate if the element is occupied or not (Figure 4.5). 8 index bits are organized in an *index block* with the size of 1 byte. Every `MemoryBin` manages a stack of maximal 1024 pointers directed to an index block that contains at least one unset index bit. In other words, it is pointing to an index block containing an unused element. The index block stack is used to determine a free block as explained below.

The method `void* binAlloc(size)` allocates memory for a given size (requiring a previously initialized bin for that size). Therefore, the method determines a suitable bin and pops the first index block of the index block stack. This index block, 8 bit, is searched for the first unset bit. Once it is found, it is inverted. If the index block still contains zero bits, it is pushed to the stack again. If not, the succeeding index block in memory is checked for unset bits and pushed to the stack in case there are any. Based on the inverted bit the start address of the memory of the element is calculated and returned. This mechanism ensures a constant allocation time $O(n)$ as long as the free elements stack is not full (1024 consecutive free operations would be necessary).

The method `void binFree(ptr)` frees memory. The method iterates through all bins and checks if the memory address to be freed is located in their payload. Once the correct bin is found, the corresponding index block/bit location is calculated. The index bit is set to zero and the block is pushed onto the index block stack.

Both methods could be improved by passing the `binID` of the allocation to the method (no bin search would be necessary). However, I did not take advantage of that to keep the signature of `void* binAlloc(size)` and `void binFree(ptr)` consistent to the standard `malloc` and `free` operations as well as to ease exchangeability and testing.

4.5 Spatial Data Structure

This section discusses the design and implementation of the 3D point data structure. Furthermore, the insertion of 3D points into the data structure as well as its out-of-core representation is explained.

Insertion at run-time (*Req02*), handling of complex geometry (*Req04*), minimal storage cost (*Req05*), and rendering at interactive frame rates (*Req07*) are the key requirements for the underlying data structure. These requirements have the following implications:

Insertion at run-time implies there must not be any preprocessing as necessary in [RD10] or [RL00]. A complete rebuild of such a data structure after modifications of the scene is too costly for interactive applications [WBB+07].

Rendering at interactive frame rates has additional implications. First, a hierarchical, spatial data structure is required to perform effective (e.g., view frustum) culling.

Second, the data structure has to store different levels of detail (also called multi-resolution data structure) to allow an appropriate detail selection which lowers the number of rendering primitives and consequently speeds up rendering. Third, the data structure has to be out-of-core capable to cope with massive data that exceeds main memory.

As stated in [GP07, p. 149], performance, compactness, and mutability may constitute conflicting targets. Given the tight main memory constraints posed by current state-of-the-art mobile devices, I considered in-core compactness of highest significance.

4.5.1 Two Layer Octree Structure

Following Wand et al. [WBB+07], an *octree* [AMHH08, p. 654][Sam90] is the preferred data structure. An octree is a hierarchical, spatial data structure which is especially well suited for dynamic operations because it organizes the embedded space (space partitioning) instead of the content itself (data partitioning) [GP07, p. 149]. Space partitioning decomposes space into disjoint cells which can be modified independently. Data partitioning generates tightly fitting cells around the data. Although data partitioning structures usually have fewer cells, they require a costly rebalance operation after modifications. This is not necessary with the octree concept due to its regular structure [YXZ08].

However, octree implementations that handle sparse distributed small objects (e.g., 3D points) as individual nodes do not scale. In pointer-based octree representations, the pointers between parent and child nodes cause a significant memory overhead (at least 4 byte per node on 32-bit architecture). In pointer-less octree representations [Sam90], in which the address of a node is computed based on its location, the memory wasted for empty cells is an even greater overhead.

To overcome this problem, I employed a two layer octree data structure similar to Wand et al. [WBB+07] (Figure 4.6). The first layer is established by the regular spatial subdivision of the octree into single non-overlapping nodes. Every node can be divided into eight equally sized sub-nodes on the subsequent octree level until the maximum octree level depth is reached. The second layer is established by a regular k^3 grid within each node covering the node's bounding cube. All nodes, and as a consequence, all grid cells of an octree level, have the same spatial extend. The center of a grid cell represents the position of a 3D point. A grid cell can be interpreted either as voxel with the exact

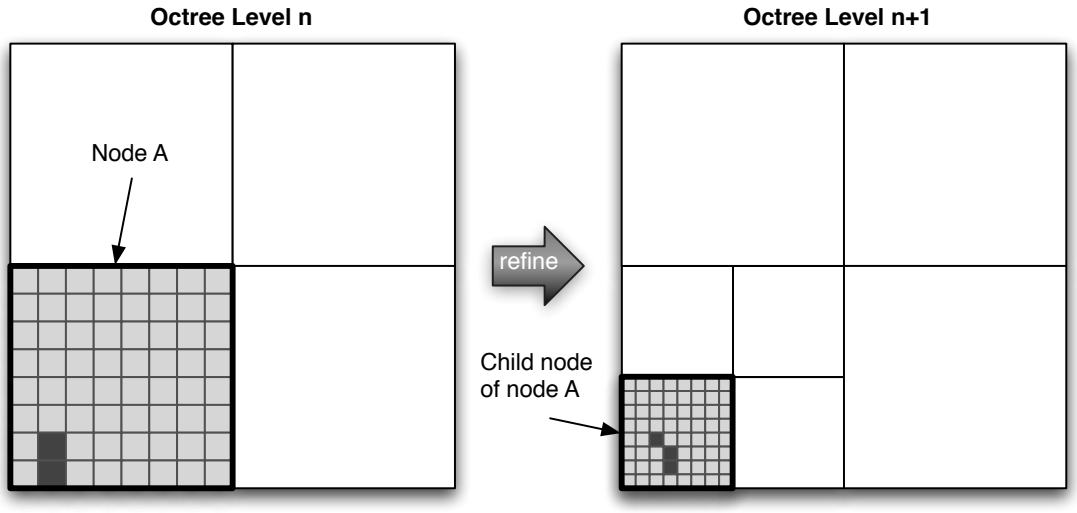

Figure 4.6: Voxels in two consecutive levels in an octree with 8^3 grid nodes. - One dimension is disregarded for better visualization. White squares show octree nodes. Light gray squares show the quantization grid within an octree node. Dark gray squares show actual voxels within a grid. The two voxels in the octree node on the left represent a coarser version of the three voxels in the octree node on the right.

volume of the cell or as sphere that encircles the cell. Consequently, every octree level represents the entire scene with a resolution according to the spatial extend of the grid cell. The root node stores the most coarse representation of the entire scene.

In [WBB+07] only inner nodes store points in a k^3 grid. Leaf nodes store 3D points with an exact position. In my implementation, all 3D point positions are quantized to k^3 grids. This approach saves memory by avoiding unnecessary fine detail as provided by WVS based 3D points.

All octree related functions and data structures are encapsulated within the class `Octree` (Figure 4.7). Every octree instance has a `MemoryPool` for its data allocation (Section 4.4.2) and a `BackingStore` to swap data to disk (Section 4.5.3). The `MemoryPool` contains two `MemoryBins`. One for `Nodes` (52 bytes) and one for `QuantPointBlocks` (128 bytes).

`Nodes` define the coarse structure of the octree. The octree has only a reference to the root node. The rest of the structure is built upon pointer links between nodes. Every node contains a pointer to its `ParentNode` (with the exception of the root node), eight pointers to its child `Nodes`, and two pointers to `Nodes` in a LRU used during

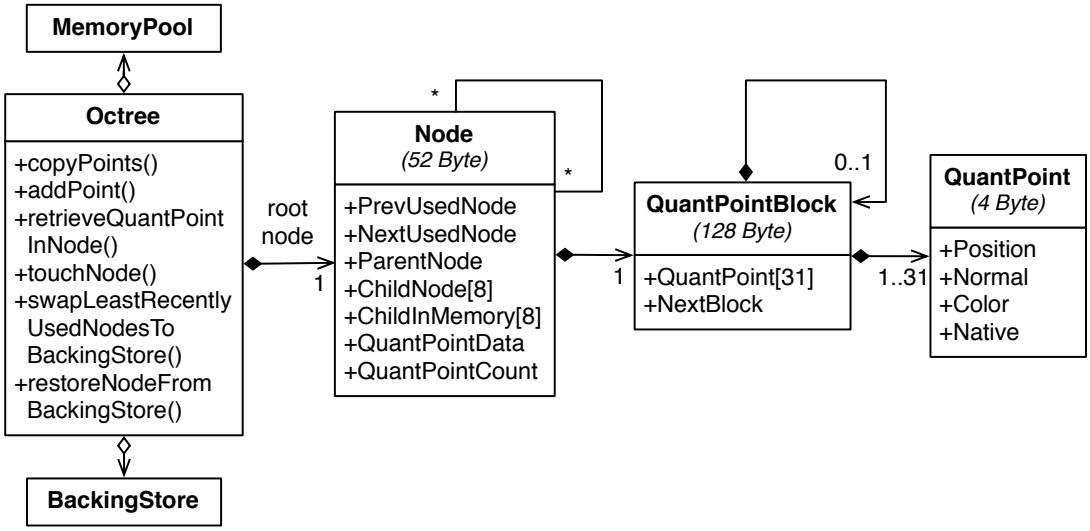

Figure 4.7: UML Class Diagram of the octree data structure. - An octree is built upon `Node` objects that are connected with each other (either via parent-child or recently used list relationship). Every node is associated with a `QuantPointBlock` that stores up to 31 `QuantPoints` (quantized points) within the bounding box of the `Node`. In case a `QuantPointBlock` is full, a new one is referenced with the `NextBlock` pointer (similar to a linked list).

rendering (Section 4.6). The child pointers are implemented as `union` [Str00, p. 257] and therefore used two-fold with three states. If a child pointer is set to `NULL`, no child is present. If a child pointer is set to a value not `NULL`, it points either to a child in memory or to a child on the `BackingStore`. This decision is made based on the `ChildInMemory` bit for that child. The advantage of this approach is a reduced memory footprint of the `Node`. If a `Node` is in-memory, the `BackingStore` location is not required. If a `Node` is on the `BackingStore`, it is not in-memory and consequently does not require an in-memory address. However, in the current implementation the size of the `union` is set to 4 byte which imposes a limitation. It is enough for an in-memory pointer on a 32-bit architecture, but it limits the size of the `BackingStore` to 2^{32} byte = 4 GB. Finally, every `Node` has a pointer `QuantPointData` to its quantized point data and a counter `QuantPointCount` that represents the entire amount of `QuantPoints` stored within it. In summary, with 2 bytes of padding for better alignment a `Node` has a memory footprint of 52 bytes.

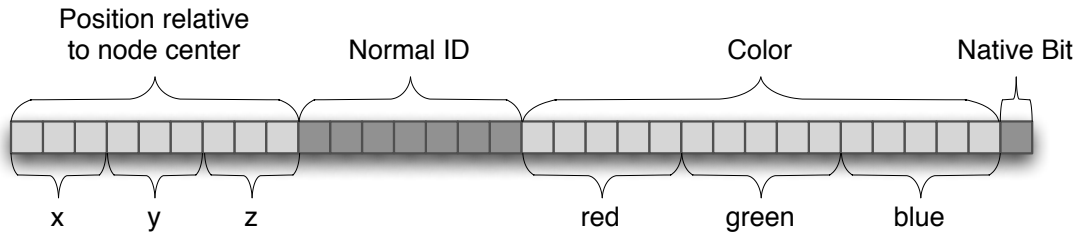

Figure 4.8: Memory layout of a `QuantPoint` - The `QuantPoint` position relative to its `Node` center is encoded with 9 bit (3 bit per dimension). A surface normal that is used to interpret the `QuantPoint` as splat is encoded with a 7 bit look-up table index. The color is quantized to 5 bit per channel and one bit indicates if the `QuantPoint` represents its native LOD or a generated coarser LOD.

A `QuantPointBlock` is a block of memory that stores n (defined in `AppConfig.h` as `OCTREE_POINTS_PER_POINT_DATA_BLOCK`) `QuantPoints` and one pointer which either points to the next `QuantPointBlock` or to `NULL` in case there is no next block. In other words this structures creates a linked list of `QuantPointBlocks`. The technique was chosen to enable an efficient mutable data structure (*Req02*) for the following reasons. First, it is not known in advance how many `QuantPoints` a node will cover and allocating space for k^3 `QuantPoints` would waste memory due to unused but allocated `QuantPoints`. Second, allocating `QuantPoints` individually as a linked list would require an additional pointer for each `QuantPoint` which would pose a significant memory overhead. Third, reallocation of the entire required memory before every modification would not work with the former mentioned `MemoryBin` based `MemoryPool` concept (Section 4.4.2) because of consequential arbitrary memory allocations. Linked `QuantPointBlocks` are considered a trade-off between these options. They work well with a `MemoryPool` and waste only a reasonable amount of memory due to partly filled `QuantPointBlocks` and `NextBlock` pointers.

A `QuantPoint` stored in a `QuantPointBlock` has four attributes (Figure 4.8). It has a position relative to its `Node` center. This is sufficient since the `Node` center position is given implicitly due to the regular octree structure. It has an index for a surface normal look-up table with evenly distributed normals on the unit sphere, a Red-Green-Blue (RGB) color information, and a bit that indicates if a `QuantPoint` represents its native LOD or a generated LOD (Section 4.5.2.2).

The design of the `QuantPoint` memory layout is crucial because it significantly affects the application's memory consumption. A small memory footprint and no padding overhead due to memory alignment are the two main goals of the `QuantPoint` design. I opted for 4 byte per `QuantPoint` because it is the natural word length on a 32-bit architecture and thus is considered to use all resources such as registers or caches as efficiently as possible.

These 32 bit are distributed amongst the former mentioned four `QuantPoint` attributes. The native LOD information requires 1 bit. 5 bit per color channel (red, green, blue) are considered sufficient and derived from the OpenGL ES internal color buffer format `GL_RGB5_A1` [MGS09, p. 261]. The remaining 16 bit are shared between position and normal index. The number of bits applied to the position determines the maximum number of `QuantPoints` in a `Node`. Since all culling and rendering algorithms operate on `Nodes` (Section 4.6), this value effects the rendering performance. A value that is too large will lead to coarser culling and therefore increase unnecessary point submission to GPU. A value that is too low will will lead to finer culling and therefore increase computation on the CPU. A preliminary performance evaluation revealed that 3 bit per dimension for the position is a reasonable compromise. This sums up to 9 bit for the position and enables a `Node` to store as much as $2^{3^3} = 8^3 = 512$ `QuantPoints`. The normal index occupies the rest of the 4 byte block, 7 bit, which are sufficient to distinguish between $2^7 = 128$ different surface normals.

The size of a `QuantPointBlock` is derived from the 8^3 grid per `Node`. An empirical analysis over several WVS data sets with this grid size resulted in an average of 28 `QuantPoints` per `Node`. For better memory alignment I increased that value to 31. Taking the `NextBlock` pointer into account, a `QuantPointBlock` has a memory footprint of $31 * 4 + 4 = 128$ byte. Consequently, the net `QuantPoint` memory footprint is on average approximately 6.4 byte (Equation 4.8).

$$netSize_{QuantPoint} = \frac{size_{Node} + size_{QuantPointBlock}}{averageQuantPoints\ per\ QuantPointBlock} \tag{4.8}$$

$$= \frac{52\ byte + 128\ byte}{28} \approx 6.4\ byte \tag{4.9}$$

4.5.2 Insertion

Insertion at run-time is a key requirement (*Req02*). This section explains how 3D points retrieved from WVS (Section 4.3.1) are transformed into `QuantPoints` and inserted into the octree data structure (Figure 4.9) at run-time. The process is performed within the `addPoint()` method of the `Octree` class.

4.5.2.1 `QuantPoint` Generation

Points in the proposed system are composed of XYZ position, RGB color, surface normal, and a bounding sphere radius. A `QuantPoint` as described Section 4.5.1 stores a reasonable approximation of these values using 4 bytes.

Listing 4.4: `QuantPoint` position in k^3 grid relative to `Node` (defined by its `center` and octree `level`).

```
// Calc all dimensions relative to node center
int32_t relativePositionX = pointPosition->x - center->x;
int32_t relativePositionY = pointPosition->y - center->y;
int32_t relativePositionZ = pointPosition->z - center->z;

// Quantize all dimensions to the node grid
int32_t nodeIncircleRadius = OCTREE_WORLD_EDGE / (1 << (level + 1));
uint8_t invLevel = OCTREE_LEAF_LEVEL - level + 1;
uint8_t x = (relativePositionX + nodeIncircleRadius) >> invLevel;
uint8_t y = (relativePositionY + nodeIncircleRadius) >> invLevel;
uint8_t z = (relativePositionZ + nodeIncircleRadius) >> invLevel;

uint16_t nineBitPosition = ((x << 7) | (y << 10) | (z << 13));
```

The coarse position of a `QuantPoint` as well as its bounding sphere radius is given implicitly by means of the node that contains the `QuantPoint` within the regular octree structure. A `QuantPoint` stores a quantized relative position within the node's k^3 grid only. It is calculated as follows: First, the position relative to the node center is determined. Second, based on the octree level, the quantized k^3 grid position of that value is calculated (Listing 4.4). To speed up computation and circumvent floating point precision errors the components of an octree XYZ position vector are represented as integer values. This is possible due to the fact that the maximum precision (Section 5.3.1) of the octree is determined at compile time (defined in `AppConfig.h` as `OCTREE_LEAF_LEVEL`). `QuantPoints` within nodes at leaf level have a bounding volume

of 2x2x2 *octree units*. Thus leaf level nodes have a spatial extend of 16x16x16 octree units.

The color of a `QuantPoint` is quantized to 5 bit per channel enabling 32,768 different colors which is considered reasonable with respect to image quality (Listing 4.5).

Listing 4.5: `QuantPoint` color quantization.

```
1 uint16_t fiveBitColor = ((color.red   >> 3) |
2                          ((color.green >> 3) << 5) |
3                          ((color.blue  >> 3) << 10));
```

The quantization of a surface normal leads to the question how to distribute a number of points uniformly over the surface of a sphere. Botsch et al. [BWK02] use a normal quantization scheme based on a recursively refined octahedron. However, they require at least 13 bit for reasonable results - too much for the 7 bit limit described in (Section 4.5.1).

I opted for a look-up table solution where 7 bit represent 128 different normals. Although the uniform distribution of n normals on a sphere is still an open research question, a reasonable precise approximation can be calculated by a simulation of points with equal force repelling themselves on the sphere until they settle [SK97]. The result of this simulation is stored as static array in `SevenBitNormalMap.h`. A given normal n is quantized by searching for the best fitting normal out of this array. Best fitting means to find the minimal angle θ between n and every normal in the array by using the dot-product.

However, 128 dot-product calculations per point are computational expensive. To improve the performance of this conversion I employed another look-up table using the fact that the WVS delivers 24 bit quantized normals. Unfortunately a full look-up table for 2^{24} seven bit values requires approximately 16 MB main memory - too much for the limited resources of mobile devices such as the iPhone. However, mapping 2^{24} values onto 2^7 values generates a high probability that two consecutive look-up table keys map to the same value. Using this fact the look-up table for the 24 bit WVS normal can be reduced to an array of 540,766 entries. Every entry contains a 24 bit value that marks the beginning of a consecutive sequence of equal normal keys in the look-up table. Furthermore, every entry stores 7 bit for its associated normal. On quantization, the position of the 24 bit WVS normal value is determined in the array using binary search and afterwards the normal is extracted. Because of the involved

binary search this approach is not as fast as a real look-up table. Nevertheless, it requires only approximately 2 MB main memory. The quantization performance is further improved by using spatial coherence. WVS G-Buffer data is processed with a scan line algorithm and consecutive points often share the same normal. Therefore, the last 24 bit WVS normal value and its according 7 bit index value is stored. If the succeeding 24 bit WVS normal value is the same as the last one, the 7 bit index is the same as well and no look-up is necessary.

4.5.2.2 `QuantPoint` LOD Generation and Insertion

`QuantPoint` LOD generation and insertion is performed in a consolidated process. On insertion, a point traverses the octree in top-down order starting at the root node. Based on the point's center position the appropriate child node in the next octree level is determined (Listing 4.6). If a required child node is not yet present, it is created. If a required child node is on the backing store, it is transferred synchronously into main memory (Section 4.5.3). This process is repeated until the sphere radius of the point is smaller than the incircle radius of the `QuantPoints` current octree level or the octree leaf level is reached (Figure 4.9). On termination, the node that contains the native LOD of the point is found. This node is called *target node*.

Listing 4.6: Calculation of the center position of a child node.

```
1  uint8_t childNodeID = 0;
2  if (parentCenter.x <= pointPosition.x) childNodeID |= 1;
3  if (parentCenter.y <= pointPosition.y) childNodeID |= 2;
4  if (parentCenter.z <= pointPosition.z) childNodeID |= 4;
5
6  childCenter.x = parentCenter.x + (childNodeID & 1 ? 1 : -1) *
7                                    _nodeIncircleRadius[level];
8  childCenter.y = parentCenter.y + (childNodeID & 2 ? 1 : -1) *
9                                    _nodeIncircleRadius[level];
10 childCenter.z = parentCenter.z + (childNodeID & 4 ? 1 : -1) *
11                                   _nodeIncircleRadius[level];
```

In a second loop, all nodes on the shortest path from the target node to the root node are visited, again in bottom-up order. For every node, the appropriate `QuantPoint` with respect to the point position is calculated. The `Octree` function `retrieveQuantPointInNode()` examines if the `QuantPoint` is already in the node.

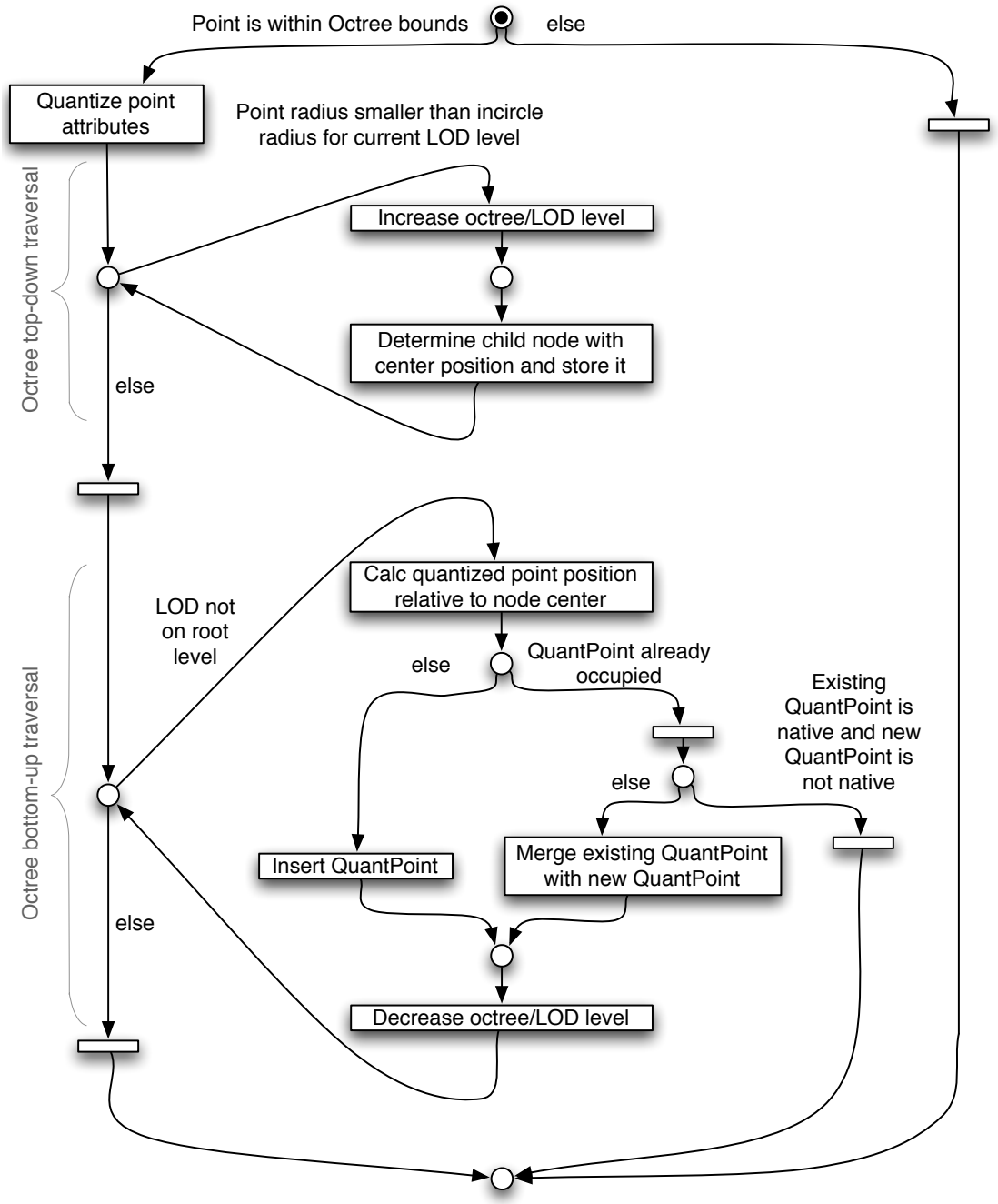

Figure 4.9: FMC Petri Net Diagram of the point insertion process. - At first, all octree nodes for all relevant LODs are determined. Afterwards, `QuantPoints` are inserted/updated in these nodes as necessary.

Listing 4.7: Moving `QuantPoints` in a `QuantPointBlock`.

```
int bytesToMove = (pointsInBlock
                   - (quantPointPtr - &(block->points[0])))
                   * sizeof(QuantPoint);
memmove(quantPointPtr + 1, quantPointPtr, bytesToMove);
```

A `Node` contains a linked list of `QuantPointBlocks` (Section 4.5.1). Every `QuantPointBlock` contains up to 31 `QuantPoints` which are sorted within the block by their grid position. To find a `QuantPoint` all blocks are processed consecutively using binary search. If the `QuantPoint` is not found, the `QuantPoint` is sort inserted in the last block moving other `QuantPoints` with `memmove` if necessary (Listing 4.7). In case the last block is full, a new block is appended. This operation is critical because it is the most expensive part of the entire insert operation (Section 5.1.3). The performance of the method is improved by unrolling the binary search loop [Ril06, p. 132ff.]. The time complexity of the search ranges from $O(log\, n)$ for one single block to $O(n)$ for a large number of blocks. Since the block size is designed in a way that that only one block is used on average, the time complexity for the search is $O(log\, n)$ on average.

Listing 4.8: Calculation of the average RGB color of two `QuantPoints`.

```
static const uint16_t redMask = 31;
static const uint16_t greenMask = 31 << 5;
static const uint16_t blueMask = 31 << 10;

uint16_t r = (((quantPoint->colorNative & redMask) +
              (newQuantPointColor & redMask)) >> 1) & redMask;
uint16_t g = (((quantPoint->colorNative & greenMask) +
              (newQuantPointColor & greenMask)) >> 1) & greenMask;
uint16_t b = (((quantPoint->colorNative & blueMask) +
              (newQuantPointColor & blueMask)) >> 1) & blueMask;

quantPoint->colorNative = r | g | b | nativeBit;
```

If the `QuantPoint` is inserted into the target node, the *native* bit is set. If a `QuantPoint` is already present in a `Node` on the bottom-up traversal and has this bit set, the traversal stops early. I assume that all coarser LODs of a native point are a better approximation of the `QuantPoint` than a `QuantPoint` not being native. If the bit is not set, the RGB colors of both `QuantPoints` are averaged (Listing 4.8). This operation is only mathematically correct for the first merge operation of two RGB

colors. All consecutive merge operations calculate the new average color based on the existing average color and the new color. However, the visual impression is better than simply using the existing or new `QuantPoint` color.

This LOD generation scheme is considered simple. More advanced algorithms with sophisticated error metrics leading to better results require preprocessing procedures [GP07, p. 143] which are not applicable due to the dynamic data structure (*Req02*). Furthermore, an inherent problem on WVS based point clouds is that not necessarily all points with the finest LOD are available. Consequently, it is not possible to calculate the coarser LODs in an exact manner.

4.5.3 Out-Of-Core Representation

As mentioned in Section 2.3.1, mobile devices such as the iPhone or iPad have hard memory constraints. Ongoing acquisition of WVS data leads to an increasing memory usage due to `Node` and `QuantPointBlock` data structures. Eventually the memory usage will exceed the capacity of the main memory.

To overcome this limitation the system utilizes the device's mass storage. This technique is called out-of-core processing and implemented with the `BackingStore` class. The system maintains a working set of actively used `Nodes` in main memory and swaps unused `Nodes` to the mass storage if necessary. This is the case if the `MemoryPool` is not able return a block of memory as requested by the `Octree`. As a consequence, the `Octree` frees memory using its `swapLeastRecentlyUsedNodesToBackingStore()` function. This function triggers the `BackingStore` to write a node including all of its `QuantPoints` to disk and stores a pointer to this data in the parent node. After this process, the `Octree` retries to allocate memory via `MemoryPool`. If this fails again, the process is repeated until enough memory is freed.

In case a node is required during rendering or insertion, it is restored by the `Octree` using the `restoreNodeFromBackingStore()` function. This function uses the `BackingStore` to restore the data and integrates it into the in-memory octree hierarchy again. On insertion, this function is called synchronous, because insertion runs in its own thread. During rendering this function is called deferred synchronous. This means, nodes to be restored are collected during rendering and processed after the frame was rendered if enough time is still available. A preliminary performance test revealed that

a separate *restore thread* as used in [WBB+07] has a negative performance impact on single core mobile devices as the iPhone or iPad.

4.5.3.1 Working Set

The in-memory working set is defined by a "least recently used" (LRU) list of `Nodes`. The LRU list is integrated into the `Node` data structure. A `Node` has two pointers called `prevUsedNode` and `nextUsedNode` to determine its predecessor and successor `Node` in the LRU. Every time the octree accesses a node (e.g. during insertion or rendering), the `Octree` method `touchNode()` is called with the `Node` as argument. Consequently, the `Node` becomes the first item in the list - the most recently used node (Listing 4.9).

Listing 4.9: Put `Node` in front of LRU list.

```
1 _mostRecentlyUsedNode->nextUsedNode = node;
2 node->prevUsedNode = _mostRecentlyUsedNode;
3 node->nextUsedNode = NULL;
4 _mostRecentlyUsedNode = node;
```

4.5.3.2 Backing Store

The backing store is a flat file on the mass storage which is cleared on every start of the application. In the case that a `Node` is swapped to the backing store, the `Node` with its `QuantPoints` is appended to the file. If the `Node` has children that are not yet on the backing store, the children are written to the file first. In the case that a `Node` is restored from the backing store, the mass storage memory that was used for it will be orphaned.

4.6 Rendering

Rendering produces an image which is displayed on the screen based on the `QuantPoints` stored in the octree data structure. This process is composed of three steps: First, the data structure is evaluated. In particular culling and LOD selection is performed to determine the `QuantPoints` relevant for the current frame. Second, the relevant `QuantPoints` are transferred to the GPU. Third, the relevant `QuantPoints` are visualized using the GPU shader engine (Section 2.3.3).

4.6.1 Input-Mode vs. Quality-Mode

Rendering is triggered with the `mainLoop()` function of the `AppCore` in the main thread (Figure 4.10). Within the main loop, the function `render()` is called to perform all tasks.

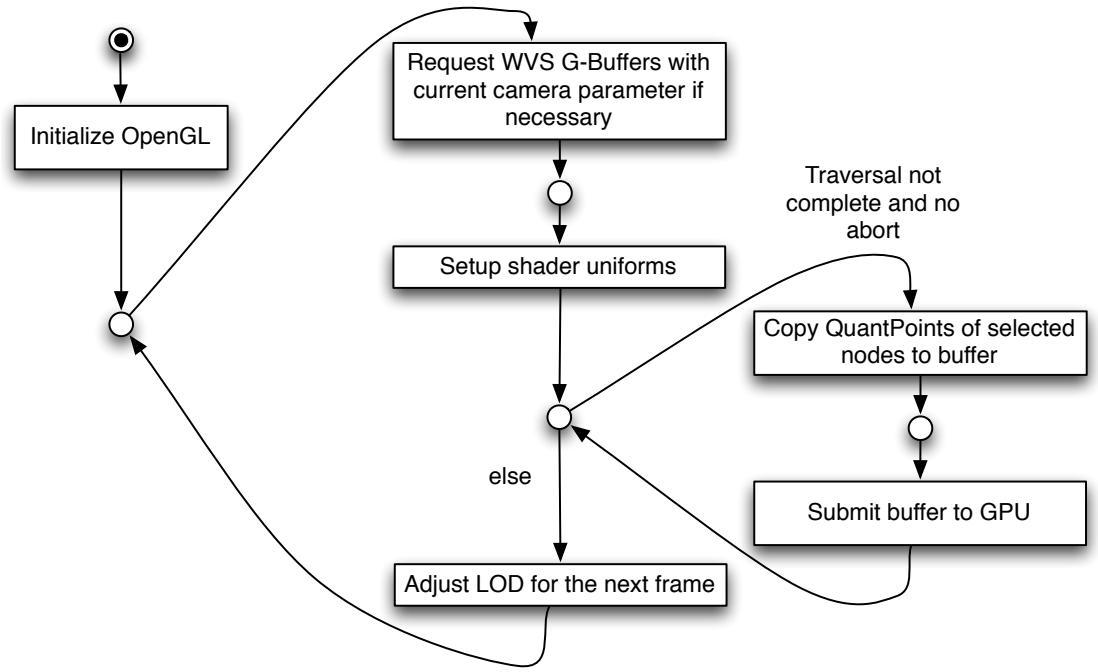

Figure 4.10: FMC Petri Net of the rendering loop. - At first, OpenGL is initialized. Following this, the rendering loop begins. Within the rendering loop, WVS G-Buffers are requested if necessary, the shader uniforms are set, the `QuantPoints` are submitted to the GPU, and the LOD is adjusted according to the time the GPU required to process the submitted `QuantPoints`.

Rendering is performed in either *input-mode* or *quality-mode*. Input-mode is activated right on any user input and keeps being active 500 ms after any user input. At all other times quality-mode is active. During input-mode, the engine adjusts the LOD in a way to keep a certain minimum frame rate (defined in `AppConfig.h` as `MINIMUM_FRAMERATE`) to enable interactive visualization (*Req07*). During quality-mode, the engine gradually increases the LOD until every point maps to a pixel (*Req06*). The UI remains responsive (*Req08*) in quality-mode, because the points are submitted in batches to the GPU. The batch size is chosen in a way to ensure that a batch can be processed in 200 ms. User input is processed after every batch submission and,

consequently, the switch to input-mode can be made at any time.

At the beginning of every frame, a timestamp t_{begin} is made. This timestamp is used to determine the elapsed time since the last camera movement (Section 4.3.3). In case a specific threshold passed and the current view was not yet requested, a new WVS request is triggered. At the end of every frame another timestamp t_{end} is made. The time $t_{frame} = t_{end} - t_{begin}$ indicates the time necessary for the entire frame. Based on this value and the former mentioned rendering mode, the LOD is adjusted for the next frame.

To fulfill the minimal energy consumption requirement (*Req12*), rendering is only performed if necessary. Thus, a new frame is rendered in one or more of the following three events: First, in the event of camera parameter change. Second, in the event of new received WVS data. Third, in the event of LOD change.

In case the engine decides that the current frame must be rendered, the shaders are configured and the recursive octree traversal is triggered (via `copyPointsToBuffer()` method in `Octree`) with a buffer to store a batch of points and a rendering callback to submit this buffer to the GPU. In input-mode the buffer is usually filled once and in quality-mode the buffer is usually filled more than once. If the the buffer is full or no points are left for rendering, the `Octree` calls the rendering callback and processes queued user input. In the case that no user input was made, the octree fills the buffer, again, until all necessary points are rendered.

The octree is traversed in depth-first order. Although breadth-first traversal has the advantage to progressively reconstruct finer LOD version of the model [BWK02], I decided against it because its space complexity $O(nodes\,at\,deepest\,level)$ might exceed main memory (*Req10*). The space complexity of depth-first traversal is much more conservative being $O(octree\,levels)$ [Bra01, p. 256].

4.6.2 Visibility Culling

Visibility culling is a task to reduce the number of points which are submitted to the GPU. In theory $points_{max} = with_{viewport} * height_{viewport}$ is an upper bound for the number of points that must be processed in PBR. But the exact selection of these points is not trivial because it would require to test every single point of the data structure individually. However, coarse culling techniques based on the octree node structure lead to fairly sufficient results. In this thesis, hierarchical view frustum culling and

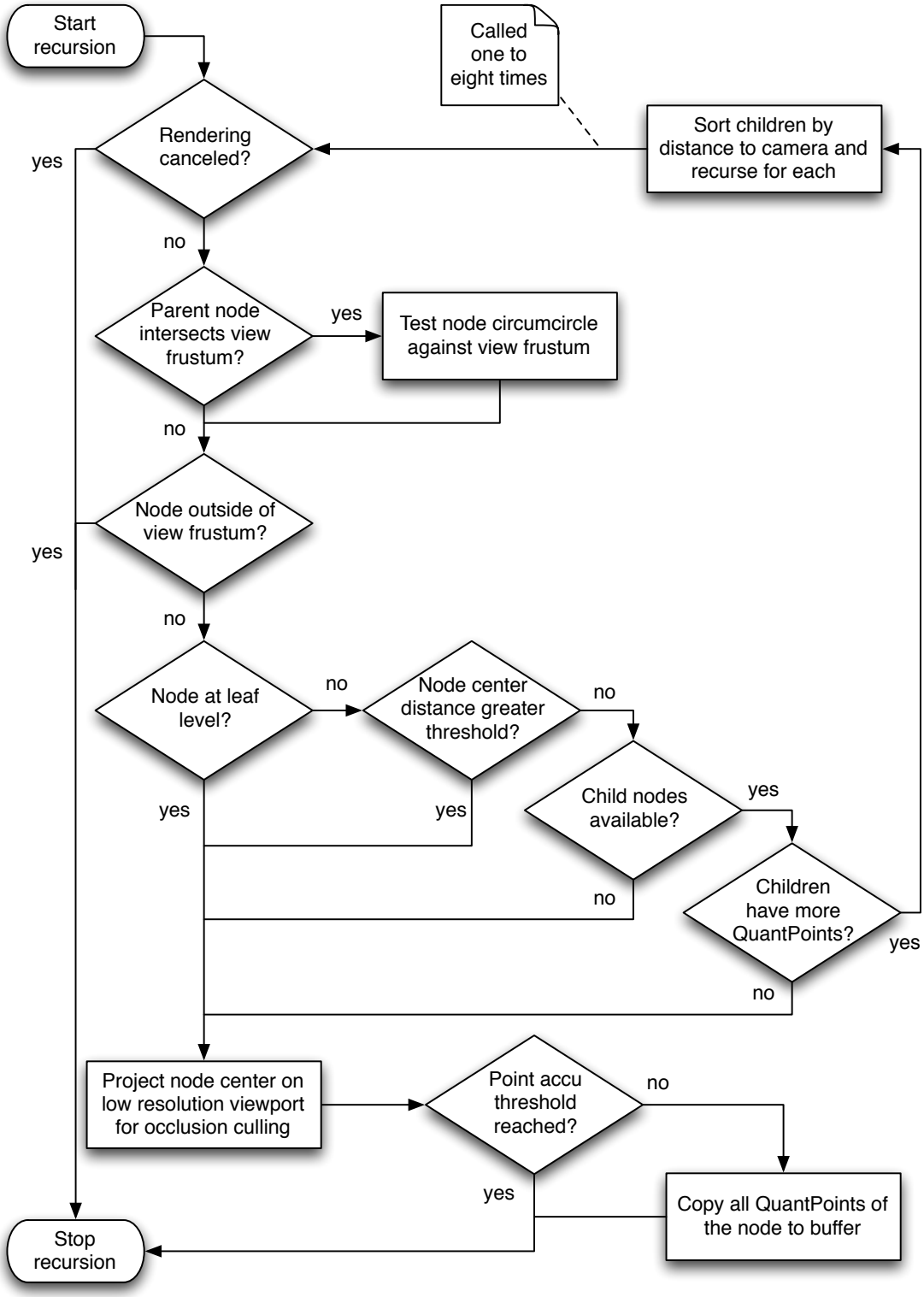

Figure 4.11: Flowchart of the octree traversal. - The focus is on hierarchical view frustum culling, occlusion culling and LOD selection.

occlusion culling were applied. The often recommended backface culling [GP07, p. 275] [AMHH08, p. 662] was omitted because the `QuantPoints` within an octree node do not necessarily share the same normal. Normal cones [RL00] have been considered as a possible solution, but a preliminary performance test did not indicate any significant improvement.

4.6.2.1 Hierarchical View Frustum Culling

During Octree traversal hierarchical view frustum culling [AMHH08, p. 664] is performed on every node (Figure 4.11). Every node is interpreted as sphere using the circumcircle radius of the octree node and checked against all six view frustum planes (Listing 4.10). Three results are possible. If the sphere is entirely *inside* the view frustum, the node is processed and view frustum culling is not performed for any child of the node. If the sphere *intersects* the view frustum, the node is processed and view frustum culling is performed for at least every direct child of the node. If the sphere is entirely *outside* the view frustum, the node is not processed nor any of its children.

Listing 4.10: Function to test a sphere against the six planes of a view frustum.

```
float_t distance;
Relation result = IN;
for(int i = 0; i < 6; ++i)
{
  // Calculate distance to plane (via dot product)
  distance =  _planes[i].x * center->x +
        _planes[i].y * center->y +
        _planes[i].z * center->z +
        _planes[i].w;

  // Check if sphere is completly outside
  if (distance < -radius) return OUT;

  // Check if sphere intersects with plane
  if(distance < radius) result = INTERSECT;
}
return result;
```

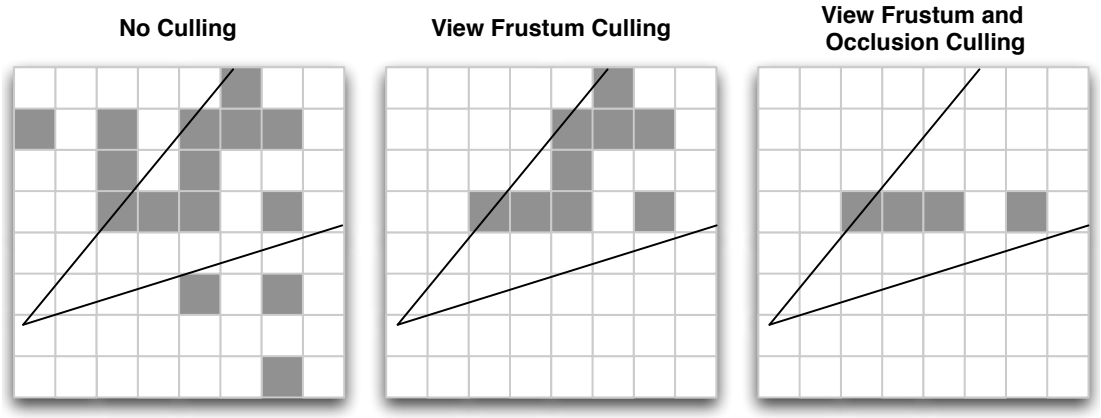

Figure 4.12: Illustration of view frustum and occlusion culling (in 2D). - A data set of 15 `Nodes` that contain one or more `QuantPoints` is visualized on the left. In the center visualization, view frustum culling is applied on the data set and nine nodes remain relevant for rendering. In the right visualization, view frustum culling and occlusion culling are applied on the data set and four nodes remain relevant for rendering.

4.6.2.2 Occlusion Culling

Although hierarchical view frustum culling significantly reduces the amount of points that have to be processed, the remaining number of points can still be unmanageable for limited mobile GPUs. This is due to the fact that within a view frustum points in the foreground entirely occlude points in the background (Figure 4.12). The process to omit these points early in the rendering pipeline is called *occlusion culling* [AMHH08, p. 670].

Visibility preprocessing to perform occlusion culling [MBWW07] would lead to fast run-time performance but is not applicable due to the dynamic data structure requirement (*Req02*). Processing every individual point at run-time and determining its visibility is not feasible due to slow run-time performance either. As a compromise I opted for an *approximate* visibility technique that ensures a fast run-time performance with the drawback of false visibility and false invisibility errors as categorized by Nirenstein et al. [NBG02].

To accomplish this task, octree nodes are depth-first traversed in a near-to-far order based on their distance from the camera (Figure 4.11). Every node N at the octree level l_N, that passes the view frustum test and the LOD selection, is processed as follows. The center of N is projected onto the viewport with a raster size of eight pixel (according

to the 8^3 `QuantPoint` array within a single node). The result are the coordinates x_r and y_r. The triple (x_r, y_r, l_N) identifies a value in an array of `integers` with the dimension $^1/_8\, width_{viewport} \times {}^1/_8\, height_{viewport} \times octree\, levels$. (x_r, y_r, l_N) accumulates the number of `QuantPoints` already in the buffer/submitted to GPU at the octree level l_N that are closer to the camera within the viewport region identified by x_r and y_r. A threshold (defined in `AppConfig.h` as `OCTREE_POINT_ACCU_THRESHOLD`) determines if all `QuantPoints` of the node N are copied to the GPU. If the threshold is reached, N and its child nodes are omitted. If the threshold is not yet reached, (x_r, y_r, l_N) is increased by the number of `QuantPoints` in N and all these `QuantPoints` are copied to the GPU buffer.

4.6.3 LOD Selection

LOD selection is the process to determine how deep to recurse until `QuantPoints` are rendered as every octree level corresponds to a LOD.

The purpose of LOD selection is twofold. First, it allows to adjust the visual quality and consequently the rendering speed of the image. Second, it decreases the number of points sent to the GPU similar to the previously discussed culling techniques. The reason for the second point is that there is no need to send points to the GPU that map to a screen-space spatial extend less than one pixel.

4.6.3.1 Node Selection

Rusinkiewicz and Levoy [RL00] employ a view-dependent LOD metric based on the screen-space projection of the spatial extent of a point sample. Sainz and Pajarola [SP04] discuss the formulas to calculate this metric. However, as discussed in Section 4.5.1 the data structure in my system is based on quantized points. This implies that there are only n (number of octree levels) different kinds of 3D points with respect to their object-space spatial extend. Consequently, a distance d_1 can be calculated at which a `QuantPoint` maps to exactly one pixel using the `QuantPoint's` inner circle radius as well as camera and viewport parameter (Equation 4.10). This calculation is made in advance and the value ${d_1}^2$ is stored in an array called `_distanceLevelThreshold`.

$$d_1 = \frac{2 * radius}{tan\left(\frac{fov_y}{2 * height_{viewport}}\right)} \tag{4.10}$$

This array is used during octree traversal to select nodes with the desired LOD (Figure 4.11). A node is always selected if it is a leaf node. In the case that it is an inner node, the squared distance between node center and camera d^2 is calculated and compared with ${d_1}^2$ for the node's level multiplied with a quality factor q (Section 4.6.3.2). If $d^2 > q * {d_1}^2$ then the node is selected and the decent terminates - otherwise the decent continues (Listing 4.11).

Listing 4.11: LOD selection.

```
if (level == OCTREE_LEAF_LEVEL)
{
  // Leaf node - try to render
  copyVoxelsToGPU = true;
}
else
{
  // Inner node - check if node maps to desired quality
  if (_viewFrustum->squaredDistanceToCamera(&nodeCenter) >
    _distanceLevelThreshold[level] * _renderQuality)
  {
    copyVoxelsToGPU = true;
  }
}
```

However, the status of a node that is not yet selected can still change. Possible child nodes are examined to check if finer detail is available [RL01]. If no children are present at all, the node is selected. If children are present on the backing store and the engine is in input-mode, the children are marked to be loaded from backing store in the future and the node is selected. If children are present on the backing store and the engine is in quality-mode, the children are loaded and traversed immediately.

However, this scheme is error-prone due to incomplete LOD's of WVS data. A parent node N_p might have child nodes $N_{c1}..N_{c8}$. However, $N_{c1}..N_{c8}$ combined not necessarily represent all geometry of N_p. To prevent the possible selection of these incomplete child nodes, the sum of all `QuantPoints` in $N_{c1}..N_{c8}$ is calculated. If this number is smaller than the number of `QuantPoints` in N_p, the finer LOD is considered incomplete. Thus the parent node is selected. Although this technique is only an approximation, it significantly improves the visual results.

4.6.3.2 Quality Factor

The quality factor determines the render quality (called `_renderQuality` in code). A factor of 1 does not influence the distance inequality (Section 4.6.3.1) and thus denotes every `QuantPoint` maps to one pixel. A value between 0 and 1 alters the inequality. A smaller value leads to coarser results.

The value is adjusted after every rendered frame. If the engine is in input-mode, it is increased respectively decreased using a fine granular stepping and a hysteresis to avoid popping artifacts [AMHH08, p. 690] (defined in `AppConfig.h` as `RENDER_QUALITY_FINE_ADJUSTMENT`). If the engine is in quality-mode, the quality is only increased (by a value defined in `AppConfig.h` as `RENDER_QUALITY_COARSE_ADJUSTMENT`), but never above a value of 1.

However, the initial value for input-mode remains a problem. Although any value converges eventually to the desired quality value, the process usually takes a number of frames that might distract the user. A value too small will converge fast but the user will see a number of coarse images. A value too large will converge slowly.

Listing 4.12: LOD estimation.

```
1  *_voxelCount = 0;
2  _renderQuality = 0.0f;
3  FIXPVECTOR3 rootCenter = {0, 0, 0};
4  do
5  {
6    _renderQuality += RENDER_QUALITY_COARSE_ADJUSTMENT;
7    lastTryCount = *_voxelCount;
8
9    countRenderBufferPoints(
10     _rootNode,
11     &rootCenter,
12     0,
13     maxPointCount,
14     MiniGL::ViewFrustum::INTERSECT);
15 }
16 while ((*_voxelCount < maxPointCount) &&
17        (lastTryCount < *_voxelCount));
```

To circumvent this effect, the quality value is estimated on every switch from quality-mode to input-mode using the `estimateInteractiveRenderQuality()` function of the

`Octree`. The function starts with a low quality value and traverses the tree using the current camera settings. On traversal, no data is copied to the GPU, but the number of `QuantPoints` that would be copied are added up (Listing 4.12). The quality value is gradually increased in a loop and the points are counted, again. The loop terminates for one of two reasons: Either the counter remains the same, denoting the model does not provide any more detail, or a specific point count threshold was reached. This threshold defines an empirically determined number of points that a platform is capable of rendering at interactive frame rates (defined in `AppConfig.h` as `OCTREE_INTERACTIVE_RENDERING_POINT_THRESHOLD`).

4.6.4 Data Processing for GPU

This section covers the preparation of the octree data for GPU submission, its transmission from main memory to GPU memory, and its decoding in the GPU vertex shader.

4.6.4.1 Preparation

The processing of sole `QuantPoints` is not sufficient due to the fact that they contain only positional information relative to their enclosing `Node` (Section 4.5.1). Consequently, the `Node's` world-space position is additionally required to calculate the absolute position of a `QuantPoint`. The `Node` position is implicitly encoded in the regular octree structure and can be calculated from the position of its parent `Node` [HPKG08].

The calculation of the absolute `QuantPoint` position is deferred to the vertex shader for two reasons. First, the tight `QuantPoint` layout has a low memory footprint and data transfer from main memory to GPU memory is expensive [Rid10, p. 367] [MGS09, p. 107] [Hou07]. Second, the implementation can utilize CPU as well as GPU (*Req11*).

The `Node` position could be copied to GPU memory by means of three floats (12 byte). Including `QuantPoint` data (4 byte), every point submitted to GPU would require 16 byte. In the following, a technique is introduced that achieves the same result by using only 8 bytes per point in total.

The `Octree` class maintains two arrays of 3D vectors called `_nextFrameRegionMin` and `_nextFrameRegionMax`. The length of these arrays comply with the depth of the octree and they are reset at the beginning of the traversal of every frame. During traversal, the center position P_N of a visited node N at octree level l is compared to the 3D

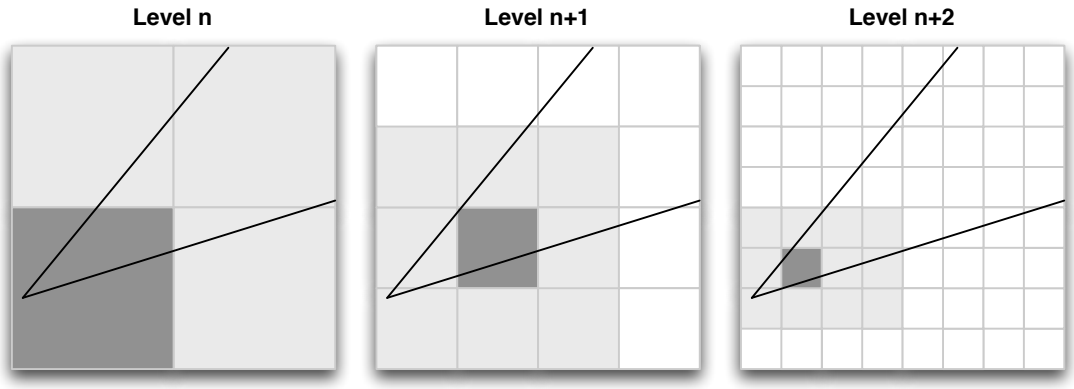

Figure 4.13: Illustration of octree regions (in 2D). - The light gray squares indicate a region, the dark gray squares a region center node.

vector values with the index l stored in the former mentioned arrays. If a component of P_N is smaller than its corresponding component in `_nextFrameRegionMin`, the component in the array is updated. If a component of P_N is greater than its corresponding component in `_nextFrameRegionMax`, the component in the array is updated. At the end of the traversal, the values of the two arrays define an axis aligned bounding box at each level that encloses all nodes for a particular level called region (Figure 4.13). Based on these bounding boxes the position of a region center node for every level is calculated.

In the octree traversal of the subsequent frame, the region center nodes are used to encode the position of the nodes to be submitted to GPU by exploiting frame-to-frame coherence. For every node, the relative position to the region center node of its level is calculated. The offset is encoded in a unit that corresponds to the node edge length of a respective level instead of an absolute distance. Every dimension of the region center offset, as well as the octree level, is encoded with 8 bit - requiring 32 bit in total (Figure 4.14). This is considered sufficient for rendering, because an 8 bit offset covers up to $2^7 = 128$ different nodes for each level in each direction of the region center. Since every node contains 8^3 `QuantPoints`, this technique can render up to $(8 * 2^8)^3 = 2048^3$ `QuantPoints` for each level. This is enough because `QuantPoints` of one level are always in a tight region close to each other due to the former mentioned LOD selection scheme (Section 4.6.3).

`QuantPoint` data and region data sum up to 64 bit or 8 byte per point in total.

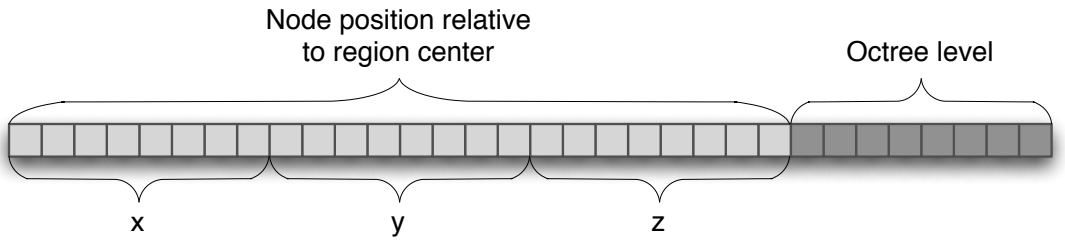

Figure 4.14: Memory layout of an octree node position for GPU submission. - The node position is calculated relative to the region center node of its level, respectively. Every dimension of the region center delta, as well as the octree level, is encoded with 8 bit.

4.6.4.2 Transmission

Vertex attributes can be submitted to GPU in two ways [MGS09, p. 104]:

Array of Structures: A memory layout in which all vertex attributes are arranged in a structure and stored in a single buffer.

Structure of Arrays: A memory layout in which every vertex attribute is stored in a separate buffer.

Although an array of structures is considered more efficient due to memory coherence [MGS09, p. 107], I opted for the alternative to exploit fast data transfer to the GPU using `memcpy` and `memset` [ETC86, p. 663]. Therefore the GPU buffer is divided into two equally sized arrays. One holds generated region data and the other one holds `QuantPoint` data.

The 32 bit region data is generated per node and afterwards set for every `QuantPoint` of the node in the GPU buffer using the `memset` function (Listing 4.13).

Listing 4.13: Setup region vertex attribute using the `memset` function.

```
1 const uint32_t nodePositionLevel = (nodeOffsetX << (0 * 8)) |
2                                    (nodeOffsetX << (1 * 8)) |
3                                    (nodeOffsetX << (2 * 8)) |
4                                    (level       << (3 * 8));
5
6 memset(regionLevelBuffer,
7        &nodePositionLevel,
8        sizeof(uint32_t) * node->quantPointCount);
```

`QuantPoints` are copied to the GPU buffer in chunks of `QuantPointBlocks` using the `memcpy` function (Listing 4.14).

Listing 4.14: Setup `QuantPoint` vertex attribute using the `memcpy` function.

```
1  QuantPointBlock* block = node->data;
2
3  while (block != NULL)
4  {
5    if ((block->next != NULL) ||
6        ((node->quantPointCount % 31) == 0))
7    {
8      memcpy(voxelBuffer, block, sizeof(QuantPoint) * 31);
9      voxelBuffer += 31;
10   }
11   else
12   {
13     memcpy(voxelBuffer, block,
14            sizeof(QuantPoint) * (node->quantPointCount % 31));
15   }
16   block = block->next;
17 }
```

The advantage of this approach is that no `QuantPoint` is handled individually on the CPU. The finest granularity is a `QuantPointBlock`. This approach saves memory read/write operations and consequently improves performance.

The GPU buffer is filled with `QuantPoints` and their region data until the buffer is full or the octree traversal is complete. The octree triggers the render callback to submit the buffer data to GPU via `glDrawArrays` call [MGS09, p. 131] in either case. In the case the buffer was full, it is flushed and the traversal continues. Submitting primitives in batches like this also improves performance [Rid10, p. 367].

Although vertex attribute submission is encouraged to be executed with *Vertex Buffer Objects* (VBO) [MGS09, p. 116] [Rid10, p. 367], I found omitting VBO's to be faster on iOS devices (Section 5.1.4). This observation is also backed by the Apple developer community [App10] and considered an OpenGL driver issue. This might change in the future.

All other data is submitted as uniforms [MGS09, p. 67] to the vertex shader:

Normal map array. 128 normal 3D vectors (12 byte each) that are used to interpret the 7 bit normal index value stored in the `QuantPoint`.

Level Region Center Array. A 3D vector for each octree level representing the center of a region in that level. These vectors are used to recalculate the absolute position of a node based on the `QuantPoint` region data.

Node incircle diameter array. A float value for each octree level representing the node edge length of that level. These values are used to recalculate the absolute position of a node based on the `QuantPoint` region data.

Voxel incircle diameter array. A float value for each octree level representing the voxel edge length of that level. These values are used to recalculate the absolute position of a voxel based on the `QuantPoint` data.

Voxel screen space splat circumcircle factor array. A float value for each octree level representing a factor to calculate the screen space dimension of a voxel at that level. Its calculation is explained in Section 4.6.4.3.

View matrix. A matrix that transforms world space 3D vectors to eye space [AMHH08, p. 16].

Projection matrix. A matrix that projects eye space 3D vectors onto a 2D plane [AMHH08, p. 18].

4.6.4.3 Shader Processing

All vertex attribute and uniform data introduced in the previous subsection are used in the vertex shader to recalculate the point data. At first the tight `QuantPoint` data structure is split in individual, memory-aligned variables. This is a major difference to other solutions such as Botsch et al. [BWK02] that perform this step on the CPU.

However, OpenGL ES Shading Language poses an obstacle because bit-wise operations are not supported [Sim09, p. 40]. To circumvent this, the mathematical `mod`, `floor`, and `*` operator are employed to mimic bit-wise `and`, `left shift`, and `right shift` (Listing 4.15).

Listing 4.15: `QuantPoint` position decoding in vertex shader.

```
// Decode relative voxel position bit pattern
// data1.x = x-------
// data1.y = zzzyyyxx
highp vec3 relativeVoxelPosition = vec3(
  mod(data1.y, 4.0) * 2.0 + floor(data1.x * 0.0078125),
  mod(floor(data1.y * 0.25), 8.0),
  mod(floor(data1.y * 0.03125), 8.0));
```

Since region data is 8 bit aligned no further calculation is necessary to retrieve the relative node position and the octree level (Listing 4.16).

Listing 4.16: Level and `Node` position decoding in vertex shader.

```
// Decode region level bit pattern
// data2.w = LLLLLLLL
lowp int level = int(data2.w);

// Decode node position bit pattern
// data2.x = XXXXXXXX
// data2.y = YYYYYYYY
// data2.z = ZZZZZZZZ
// data2.w = --------
highp vec3 relativeNodePosition = vec4(data2.x, data2.y, data2.z);
```

The octree level is used to retrieve the absolute region center, the node incircle diameter, and the voxel incircle diameter, respectively. Finally, all data is combined to calculate the absolute point position in world space (Listing 4.17).

Listing 4.17: Level and `Node` position decoding in vertex shader.

```
position.xyz = regionCenter[level] +
               nodeIncircleDiameter[level] * relativeNodePosition +
               voxelIncircleDiameter[level] * relativeVoxelPosition;
```

In case the normal data is required (e.g., if backface culling or splatting is activated via compiler switch), the normal index is decoded and used to retrieve the normal with the help of the normal map array (Listing 4.18).

Listing 4.18: Normal decoding in vertex shader.

```
// Decode normal bit pattern
// data1.x = -nnnnnnn
highp vec3 normal = normals[int(floor(mod(data1.x, 128.0)))];
```

The color is decoded and aligned similar to the relative voxel position. Eventually, the absolute point position in world space is transformed to eye space using the `ViewMatrix` and subsequently projected onto the image plane using the `ProjectionMatrix`.

In the vertex shader, fast splatting is achieved via `gl_PointSize` [Sim09, p. 59]. This value represents the diameter of a rasterized OpenGL point in pixel and is based on the viewing parameter and the `QuantPoint` size. The calculation of the precise value is computational complex and therefore slow to execute. In the following, a simplified approximation is discussed.

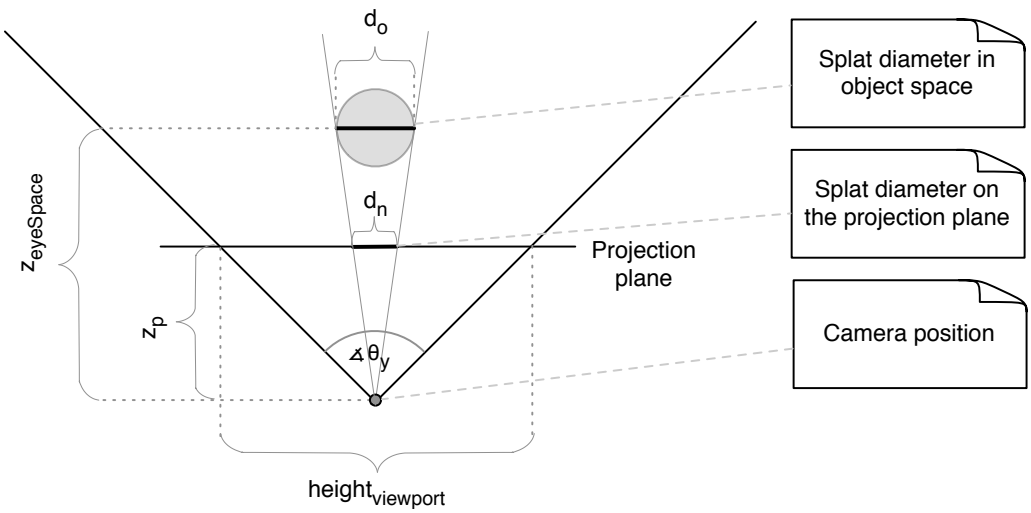

Figure 4.15: Approximate calculation of the splat size. - `QuantPoints` are interpreted as their encircling sphere. This sphere is projected onto the projection plane.

A `QuantPoint` is interpreted as its encircling sphere. This sphere is projected onto the projection plane. According to the concept of similar triangles and the tangent function as shown in Figure 4.15 the following is valid:

$$\frac{d_n}{d_o} = \frac{z_p}{z_{eyeSpace}} \tag{4.11}$$

$$d_n = \frac{z_p}{z_{eyeSpace}} * d_o \tag{4.12}$$

$$tan\left(\frac{\theta_y}{2}\right) = \frac{\frac{height_{viewport}}{2}}{z_p} \tag{4.13}$$

$$z_p = \frac{\frac{height_{viewport}}{2}}{tan\left(\frac{\theta_y}{2}\right)} \tag{4.14}$$

Combining Equation 4.12 and 4.14 leads to:

$$d_n = \frac{1}{z_{eyeSpace}} * d_o * \frac{\frac{height_{viewport}}{2}}{tan\left(\frac{\theta_y}{2}\right)} \tag{4.15}$$

A part of the calculation in Equation 4.15 can be computed in advance:

$$f_{const} = d_o * \frac{\frac{height_{viewport}}{2}}{tan\left(\frac{\theta_y}{2}\right)} \tag{4.16}$$

The viewport and the field of view remain constant. The number of values for d_o corresponds to the number of levels in the octree and is constant as well. The resulting values for f_{const} are calculated on application startup and transmitted to the shader. Consequently, the shader has to perform the following calculation to approximate the size of the OpenGL points:

$$d_n = \frac{1}{z_{eyeSpace}} * f_{const} \tag{4.17}$$

The point size calculated in Equation 4.17 is defined as a non-negative integer and interpreted as a distinct pixel. To avoid rounding errors and ensure closed surfaces the value 1.5 is added to the calculated point size in the shader.

5

Evaluation

The implementation presented in Chapter 4 is evaluated in this chapter. First, the performance of key components is analyzed. Second, visual artifacts induced by the implementation are emphasized. Third, fulfillment to the requirements (Chapter 3) are discussed.

5.1 Performance

In this section, memory allocation, image decoding, 3D point acquisition, and vertex submission are analyzed. All experiments are performed on an Apple iPhone 3GS and on an Apple iPad device, both running iOS 4.2. No experiment is performed on the Mac OS X platform, because a desktop machine always outperforms a mobile device. Before every experiment, the device was shut down and restarted to ensure a clean test bed.

5.1.1 Memory Allocation

In the memory allocation experiment, the `malloc` implementation provided by the operating system is compared with the memory pool implementation used in the rendering system (Section 4.4.2). Allocation speed and allocation memory overhead are the metrics to evaluate these implementations. Allocation speed denotes the time that is required to allocate a block of memory. Memory overhead indicates the memory that is required on top of the requested memory for the allocation management.

The experiment is executed as follows: At the beginning, the free available memory on the device is determined with the help of the `host_statistics` function [Sin06, p. 526]. In the memory pool configuration, a memory pool of 64 MB for 52 byte octree `Node` objects is generated (suitable for 1,287,456 allocations). Afterwards, a timer is started and 1,287,456 `Node` objects are allocated on both configurations. In order to suppress compiler optimizations, a 4 byte value is written to every allocated object. Finally, the timer is stopped and free memory is yet again determined with the `host_statistics` function.

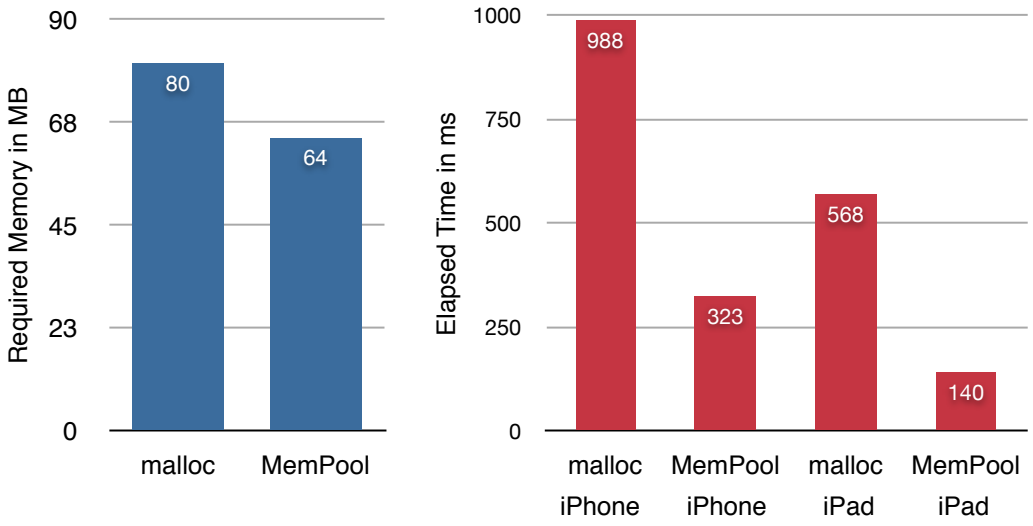

Figure 5.1: Performance comparison between `malloc` and my memory pool implementation on iPad and iPhone 3GS. - The left chart depicts the required memory and the right chart the elapsed time to allocated 1,287,456 objects with a size of 52 bytes.

This experiment runs 100 times for both configurations on both devices (Figure 5.1). On average, the `malloc` implementation used 80.26 MB main memory and the memory pool implementation used 64.29 MB. 1,287,456 allocations for a 52 byte `Node` required 63.85 MB. Consequently, the `malloc` implementation required 16.41 MB and the memory pool 0.44 MB overhead. To accomplish this task, the `malloc` configuration on average required 988 ms on the iPhone 3GS and 568 ms on the iPad. The memory pool configuration on average required 323 ms on the iPhone 3GS and 140 ms on the iPad. I assume that the iPad is able to perform the task almost twice as fast due to the higher CPU and main bus clock rate as well as the larger level two cache (Section 2.3).

	A Full iPad	B Full iPhone	C ⅛ iPad	D ⅛ iPhone
Resolution	768x1024	320x480	192x256	80x120
JPEG Decoding Time	186 ms	66 ms	12 ms	5 ms
PNG Decoding Time	100 ms	39 ms	8 ms	4 ms
JPEG Image Size	150 KB	33 KB	12 KB	3 KB
PNG Image Size	748 KB	144 KB	45 KB	9 KB

Table 5.1: Image file size and decoding time using the image depicted in Figure 5.2 on iPad and iPhone 3GS.

5.1.2 Image Decoding

In the image decoding experiment, the client decoding performance of the image formats supported by the WVS is evaluated. In particular, JPEG decoding using `libJPEG` and PNG decoding using `libPNG` was evaluated. The metric for the evaluation is the time that is necessary to decode a single image.

A preliminary observation was that pure JPEG/PNG image decoding time directly depends on the image content [Sal00, p. 361]. Decoding a color image usually requires more time than decoding a depth or normal image because color images contain much less redundant and much more high frequency information. Consequently, a color image displaying an exemplary urban scene is used for this evaluation (Figure 5.2).

The experiment is performed with a fullscreen image and a ⅛ screen image compressed using JPEG and PNG with 95% quality on both devices (Figure 5.2). The visualized decoding time is an average over 100 image decoding operations for the same image.

On both test devices PNG decoding is almost twice as fast JPEG decoding. However, PNG encoded images required roughly four times the memory as required by JPEG encoded images (Table 5.1).

5.1.3 3D Point Acquisition

In the point acquisition experiment, the performance of 3D point acquisition is evaluated. The metric is the time that is required to request, download and process a set of WVS G-Buffers. The G-Buffers set consists of a JPEG encoded color image, a PNG encoded depth image, and a PNG encoded normal image.

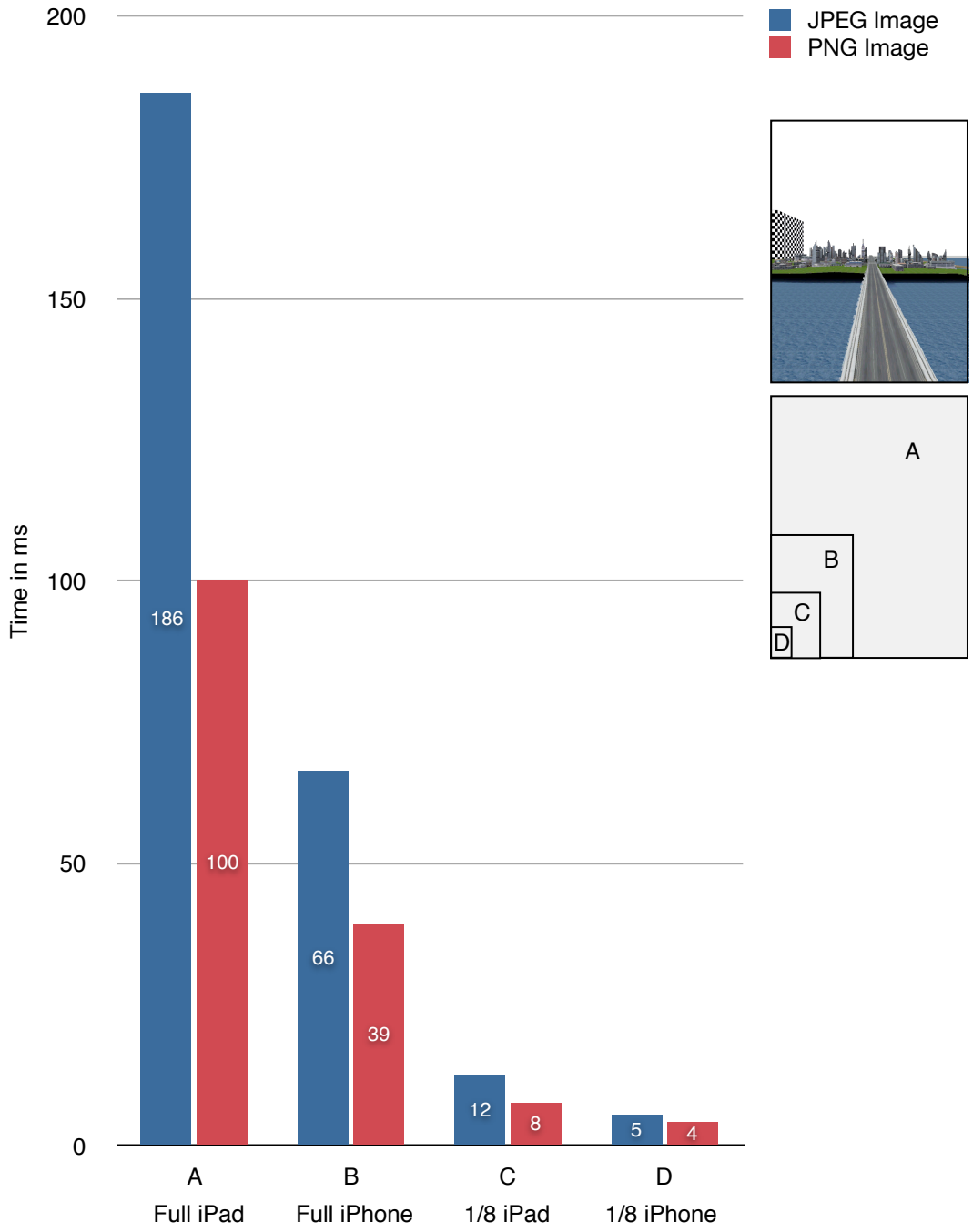

Figure 5.2: Performance comparison between JPEG and PNG image decoding on iPad and iPhone 3GS. - Column A and B depict fullscreen images on the test devices and column C and D ⅛ screen images, respectively. The image used for decoding is depicted on the upper right. The different resolutions A, B, C, and D are depicted on the lower right.

Experiment ID	A Full iPad	B Full iPhone	C ⅛ iPad	D ⅛ iPhone
Requested Resolution	768x1024	320x480	192x256	80x120
Requested Number of Pixels	786,432	153,600	49,152	9,600
Download Size	369 KB	89 KB	33 KB	10 KB
Generated Octree Nodes	39,826	9,409	3,298	765
Generated Octree Points	766,259	157,195	51,389	10,421
Proportion Processing	40%	36%	17%	8%
Proportion Octree Insertion	32%	30%	14%	5%

Table 5.2: G-Buffer resolution, number of pixels, download size, number of allocated octree nodes, number of allocated `QuantPoints`, and the percentage of client side processing of the urban city scene depicted in Figure 5.3 on iPad and iPhone 3GS.

For each device, two configurations, a fullscreen and ⅛ screen set of G-Buffers, were tested with the same urban city scene (Table 5.2). The total time for each run is made up of three parts (Figure 5.3). The first part is concerned with the request, which is the time between the HTTP GET request and the first received packet in response. The second part is concerned with the download, which is the time that is necessary to download all response packets. The third part is concerned with the processing, which is the time for G-Buffer decoding, memory allocation, and `QuantPoint` insertion (Section 4.5.2). All configurations are executed on an empty `Octree` data structure, therefore all necessary data structures are allocated in the processing step.

The request time depends primarily on WVS rendering speed and the download time depends primarily on the wireless connection bandwidth. The client has no influence on either of these values. Nevertheless, they require up to 92% of the entire point acquisition time (Table 5.2) depending on the requested G-Buffer size.

The only part of the point acquisition the client has influence on is the processing on the client. G-Buffer decoding and memory allocation require noticeable time but the most time consuming operation using at least 80% of the processing time is octree insertion. Further code profiling reveals that the majority of that time is spend in the `Octree::retrieveQuantPointInNode()` method. This method performs a binary search for a specific `QuantPoint` in the `QuantPointBlocks` of an octree node (Section 4.5.2.2).

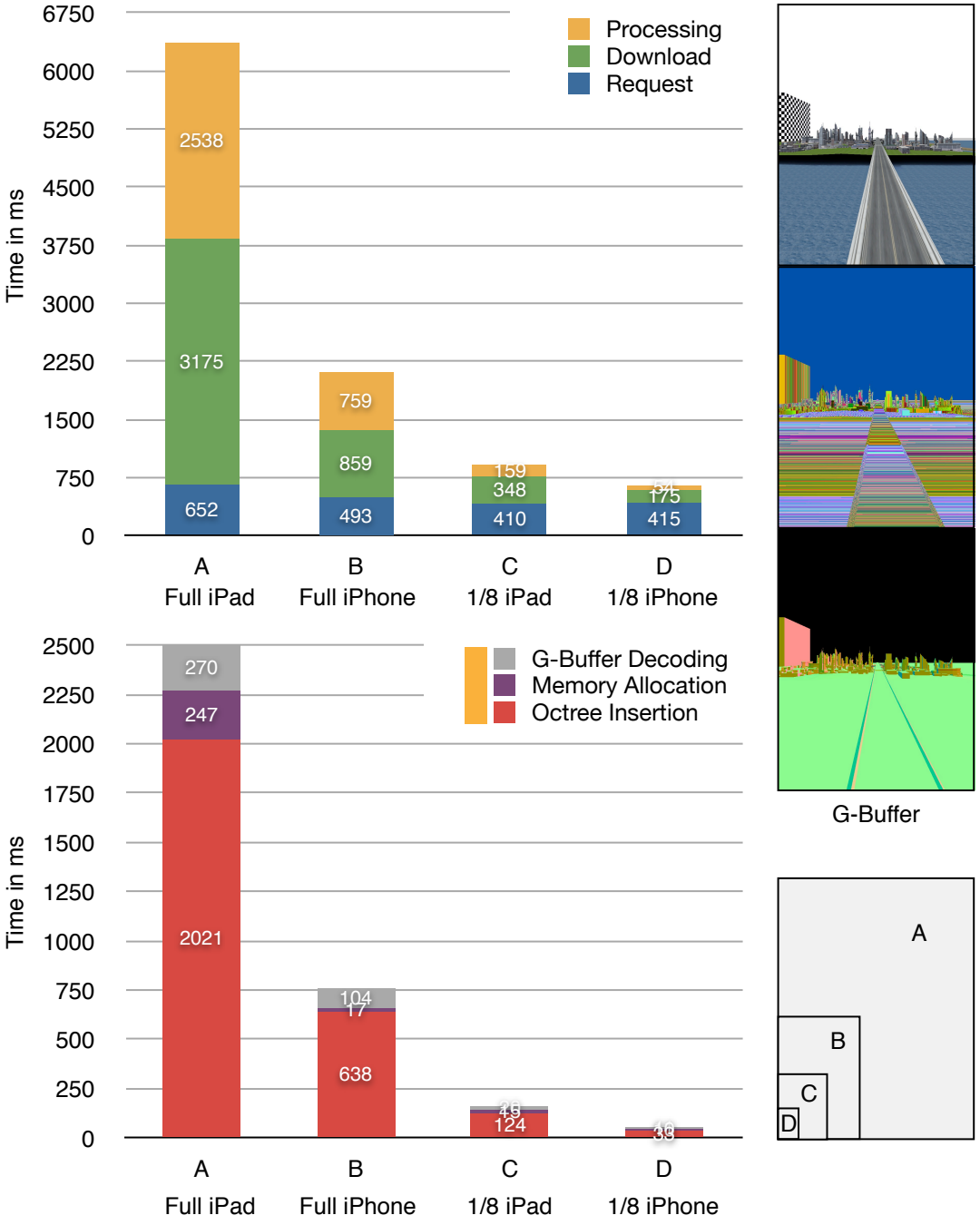

Figure 5.3: Time to request, download, and process WVS G-Buffers on iPad and iPhone 3GS. - Column A and B depict processing time of fullscreen G-Buffers on the test devices and column C and D ⅛ screen G-Buffers, respectively. The value in the lower chart is a breakdown of the processing value of the upper chart. The color, depth, and surface normal G-Buffer are depicted on the right side.

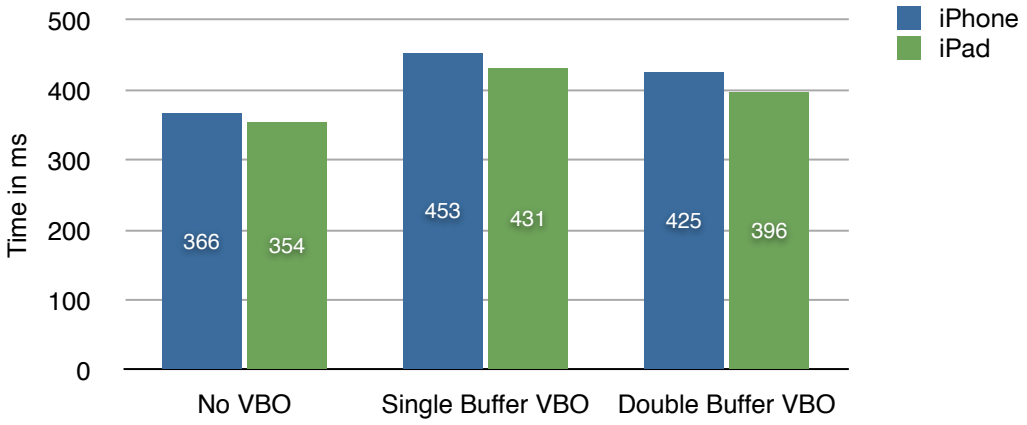

Figure 5.4: Submission and rendering of 1,000,000 points on iPad and iPhone 3GS. - The time is averaged over 100 runs for each configuration.

5.1.4 Vertex Submission

In this section, different means of vertex submission are evaluated. First, direct submissions not using VBOs are evaluated. Second, submissions using single buffered VBOs are evaluated. Third, submission using double buffered VBOs are evaluated. The evaluation metric is the time that is required to submit and render a single frame.

All three configurations are executed on the test devices using 1,000,000 points. The result for a configuration is averaged over 100 runs.

Although the usage of VBOs is encouraged by Apple for performance reasons, experimental results show that not using VBOs is faster on both test devices (Figure 5.4). However, the performance of VBOs can be improved by using double buffered VBOs.

5.1.5 Rendering Speed

In the rendering speed experiment, the performance of the point renderer is evaluated. The metric is the time that is required to render one frame. In addition, the accumulated size of the downloaded WVS images, the total number of `QuantPoints` in the octree, and the number of `QuantPoints` submitted to GPU are recorded.

On each device an urban city scene is explored for 60 seconds starting with an empty octree (Figure 5.5 and Figure 5.6). After 60 seconds the iPhone 3GS has downloaded approximately 2.3 MB and the iPad 11.1 MB of G-Buffer data. This data is transformed

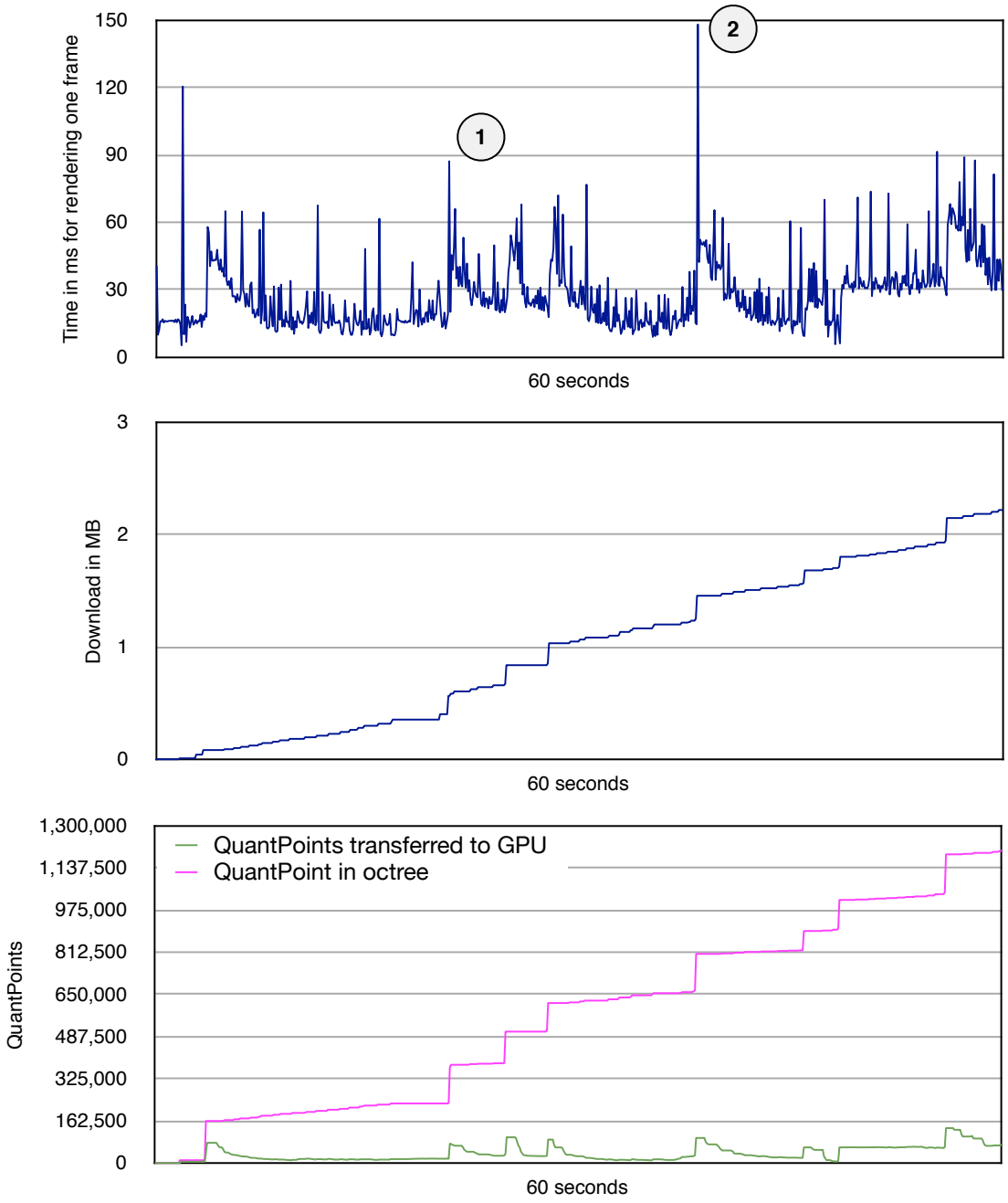

Figure 5.5: Exploration of an urban city scene on the iPhone 3GS. - The top chart shows the time that is required to render a single frame. The middle chart shows the amount of data downloaded from the WVS. The bottom chart shows the number of `QuantPoints` in the octree and the number of `QuantPoints` transferred to GPU. The events (1) and (2) indicate quality-mode rendering with data from the WVS that matches exactly the current view.

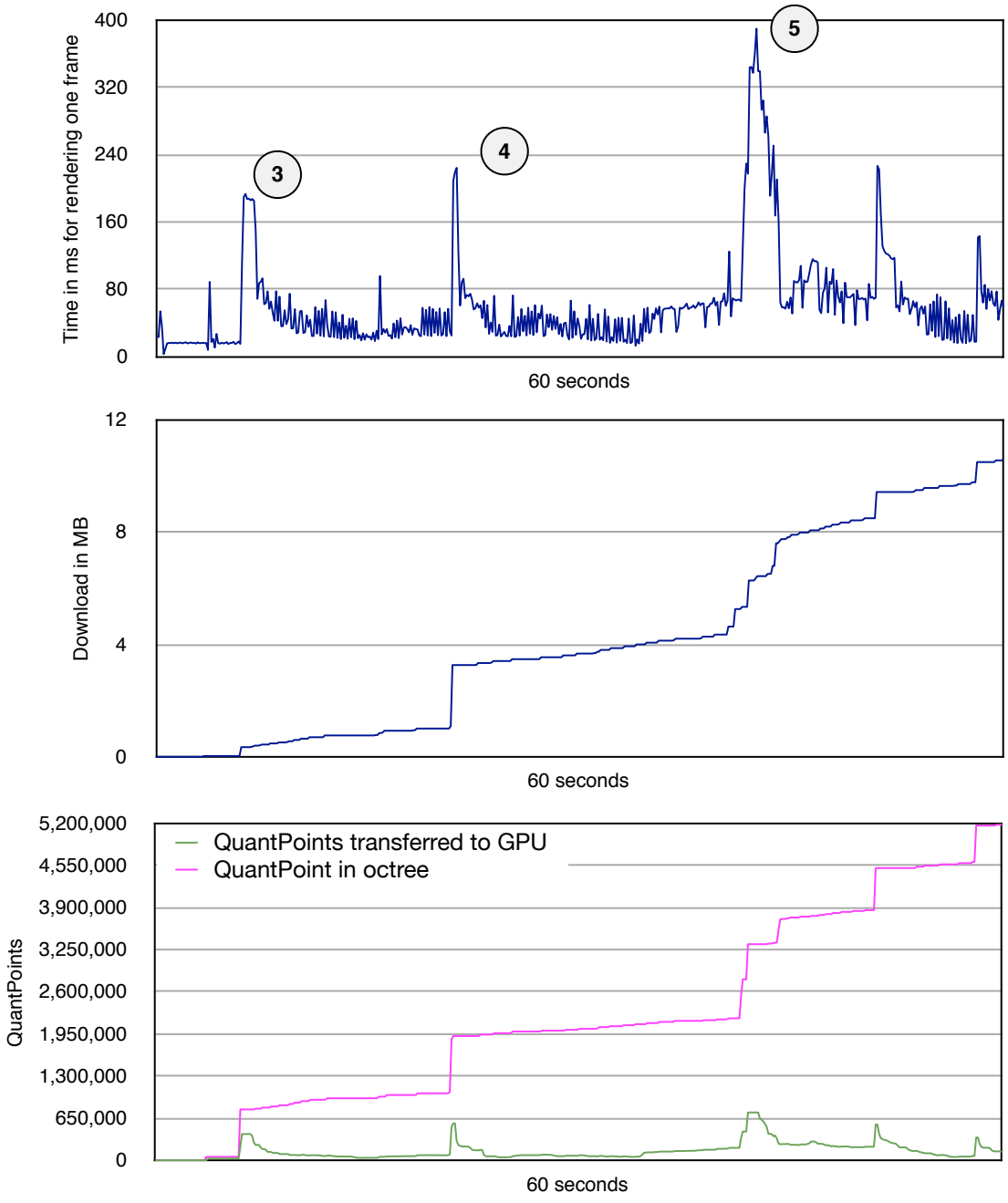

Figure 5.6: Exploration of an urban city scene on the iPad. - The top chart shows the time that is required to render a single frame. The middle chart shows the amount of data downloaded from the WVS. The bottom chart shows the number of `QuantPoints` in the octree and the number of `QuantPoints` transferred to GPU. The events (3), (4), and (5) indicate quality-mode rendering with data from the WVS that matches exactly the current view.

into 1.204.545 `QuantPoints` on the iPhone 3GS and 5.179.657 `QuantPoints` on the iPad, respectively.

The iPhone 3GS requires on average 27.7 ms for a single frame and the engine is 33% of the time in input-mode. The iPad requires on average 61.23 ms for a single frame and the engine is 48% of the time in input-mode.

The events (1) to (5) in Figure 5.5 and Figure 5.6 show a significant decrease in rendering speed. In these events the engine is in quality-mode and new data from WVS became available. Consequently, the engine can send a lot more `QuantPoints` to the GPU. In (1) to (4) the screen is touched right after the new data became visible. Consequently, the engine switches to input-mode and the rendering speed increases. In event (5) the screen is not touched right away. Consequently, the engine remains in quality-mode and the rendering speed remains low.

5.2 Visual Limitations

In this section visual limitations induced by the implementation are emphasized. In the following, all known limitations are visualized and their origins are explained.

5.2.1 Insufficient Precision

A key design decision is the quantization of points to a predefined grid. Although the grid increases its precision with every octree level, it is not practical to have an arbitrary deep octree because of the increased memory usage. Moreover, the precision of WVS G-Buffer data is limited to its resolution. Consequently, the octree depth is limited to 15 levels which provide a reasonable compromise between memory consumption and precision. However, especially on sharp edges, quantization artifacts are visible (Figure 5.7).

I tried to overcome these artifacts by applying Gaussian blur in a post processing shader. Although the filter reduced these artifacts, the overall image quality decreased due to the blur. Therefore, the filter is not used in the current implementation.

5.2.2 Imprecise Quantization

Imprecise quantization can happen if distant fine grained geometry is represented with a single pixel (e.g., radio antenna of the skyscraper in Figure 5.8, left). If the position

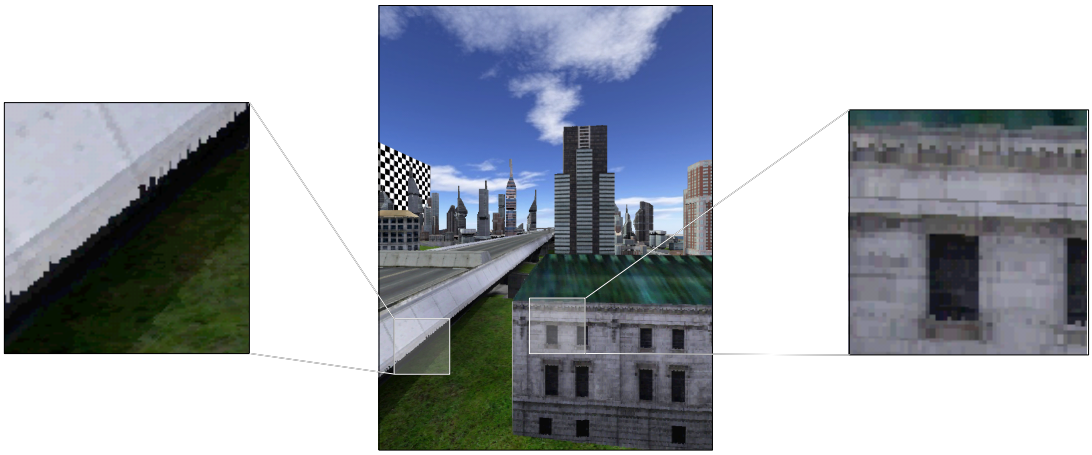

Figure 5.7: Screenshots depicting artifacts due to grid quantization. - The left magnification shows torn up edges in geometry and the right magnification shows quantization artifacts in texture.

Figure 5.8: Screenshots depicting imprecise quantization. - The quantized position of some `QuantPoints` representing the antenna and the front of the skyscraper in the left image are off. Consequently these `QuantPoints` are not recognized as coarser LOD of the geometry in the right images and erroneously rendered.

of the `QuantPoint` generated based on that pixel is off in a way that the `QuantPoint` is added to a wrong `Node`, subsequently added `QuantPoints` with a finer LOD are not underneath that `Node`. As a consequence, the implementation considers these `QuantPoints` as independent geometry rather than the same geometry with different LOD and renders them both (Figure 5.8, right).

5.2.3 Incomplete LODs

WVS G-Buffers contain foreshortened views of the scene. Therefore, each pixel of the color image represents a 3D point with an individual LOD depending on the pixel's depth image value (distance to camera). In the proposed system, 3D points are organized as `QuantPoints` within `Nodes`.

The LOD selection during rendering is based on `Nodes` rather than `QuantPoints` (Section 4.6.3). The inherent problem is that the system can never know if finer detail is available even though a `Node` has children. This is because the aggregated `QuantPoints` of the children might represent less geometry than the `QuantPoints` of the parent node itself.

To overcome this problem, the implementation uses a heuristic. The aggregated number of `QuantPoints` in the child `Nodes` must be greater than the number of `QuantPoints` in the parent `Node` to be considered as complete LOD. However, in some cases cracks arise if the child nodes contain more `QuantPoints` than the parent but do not cover all geometry (Figure 5.9 right).

5.2.4 Surface Cracks

A view with the camera parameter C is rendered without noticeable errors if there was a preceding WVS request with the same camera parameter C (Figure 5.9, left). However, if the camera parameter changes to C', e.g., due to a camera movement, and no WVS request was yet performed, surface cracks occur.

These cracks arise in parts of the scene that were not visible from C. Due to the concept of a G-Buffer, this functioning is inevitable in the case that an object was occluded in C and is visible in C' (e.g., a building in Figure 5.9, right). However, these cracks also occur on entirely visible surfaces (e.g, water and street in Figure 5.9, center). The reason is that the notion of *occlusion* changes in context of G-Buffer based point clouds. In this context, there is no closed or complete surface anymore. Instead there

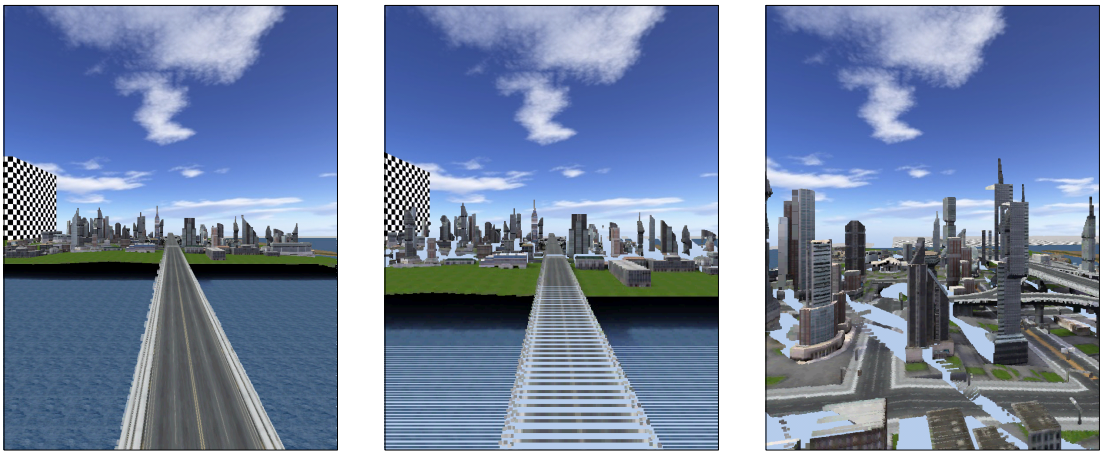

Figure 5.9: Screenshots depicting different surface cracks. - The left image visualizes a point cloud based on WVS G-Buffers that match exactly the camera parameters. The center images shows cracks on surfaces if the camera parameters for that point cloud are changed. The right image shows cracks due to object occlusion.

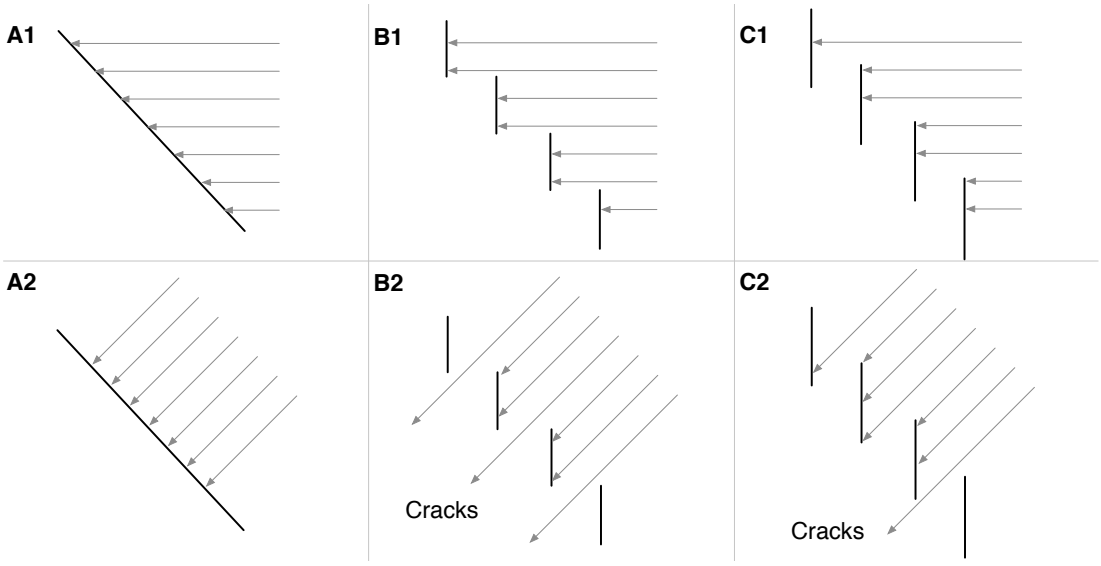

Figure 5.10: Schematic visualization of surface cracks due to altered camera parameter. - A depicts a perfect surface. B depicts a surface approximated by `QuantPoints` with a dimension based on the G-Buffer pixel. C depicts a surface approximated by `QuantPoints` with a dimension based on the G-Buffer pixel taking the surface normal into account. The lower images visualize an altered view point. Rays going through the surfaces represented in $B2$ and $C2$ become surface cracks.

are single individual points with an extend defined by the G-Buffer pixel and these points occlude each other (Figure 5.10, center).

I tried to overcome this obstacle by calculating the `QuantPoint` size depending on the surface normal of the pixel in the G-Buffer (Figure 5.10, right). This technique decreases the number of cracks and works reasonable well for large surfaces. However, it also leads to larger `QuantPoints` and therefore to a coarser scene representation. Moreover, the normals are bound to the G-Buffer pixel raster and therefore represent an average value for the entire pixel. As a consequence, the `QuantPoint` size of small geometry, e.g., street lights, is often calculated to large. Therefore, this technique is not employed in the implementation.

5.2.5 Incomplete Visualization

`QuantPoints` are submitted in batches to the GPU (Section 4.6.4.2). The batch size is chosen in a way that in input-mode a frame is rendered using only a single batch. However, in quality-mode a frame is usually rendered with a number of batches. After every batch submission, user input is processed. If user input is detected, the engine switches from quality-mode to interactive mode. As a result the user might see a half rendered frame.

5.3 Requirement Fulfillment

In this section the fulfillment of the requirements (Chapter 3) are discussed.

5.3.1 Data and Data Structure Requirements

All `QuantPoints` originate from WVS based G-Buffers (*Req01*). New `Nodes` containing new `QuantPoints` can be inserted dynamically into the data structure (*Req02*). Using the `BackingStore`, the data structure can grow beyond main memory (*Req03*). The octree has a maximum depth of 15 levels which is defined at compile time (the maximum level depth can be adjusted to up to 256). A `Node` contains 8^3 `QuantPoints` at maximum. If every `QuantPoint` represents $15\,\mathrm{cm}^3$, an octree with 15 levels can cover a volume with an edge length of approximately $39,322\,\mathrm{m}$ $(0.15 * 8 * 2^{15}\,\mathrm{m})$ or a squared tile with a spatial extend of approximately $1,546\,\mathrm{km}^2$ $(((0.15 * 8 * 2^{15})^2)\,\mathrm{m}^2)$ (*Req04*). The memory footprint of a single `QuantPoint` is 6.4 bytes on average (*Req05*).

5.3.2 Rendering Requirements

If sufficient data is present, the rendered image does not not show cracks (*Req06*). The interactive rendering mode ensures interactive rendering with more than 15 frames per second and a user response time below 0.2 seconds (*Req07*, *Req08*).

5.3.3 Platform Requirements

The implementation is platform-independent and runs on mobile devices with iOS as well as on desktop computer with Mac OS X (*Req09*). The implementation can be configured in a way to limit the amount of maximal allocated main memory to avoid memory related crashes on the iOS platform (*Req10*).

5.3.4 Hardware Requirements

The GPU is employed to decode tightly packed `QuantPoint` data (*Req11*). The implementation renders frames only if new data was received or camera settings changed (*Req12*).

6

Conclusions

In this thesis, concepts and techniques for interactive real-time rendering of 3DGeoVEs represented as point clouds on mobile devices are presented. The rendering is based on a dynamic multi-resolution data structure which is extended at run-time with G-Buffer data provided by a web service such as WVS. In the following, the contributions are recapped, the results are discussed, and future work is outlined.

6.1 Contributions

The main contributions are:

- The presented solution is capable of rendering 3DGeoVE on current mobile devices in real-time.
- Geometry and texture for the 3DGeoVE are derived from G-Buffers requested from a web service and transformed into an octree based point cloud in real-time.
- The system enables the exploration of 3DGeoVE incrementally downloaded from a remote host. Therefore, the amount of data necessary to store the entire 3DGeoVE can exceed the mass storage capacity of the mobile device.
- The octree data structure that manages the 3DGeoVE data can be processed in real-time.
- The memory footprint of the octree data structure can exceed the main memory capacity using out-of-core algorithms which swap unused data to a backing store.

- All 3D point data is quantized as `QuantPoints`. `QuantPoints` are organized within the spatial extent of an octree node in a regular 8^3 grid. They have a 4 byte memory layout that stores the relative position to the octree node center, the color, and the surface normal. This tight memory layout is used by the backing store, the main memory, and the GPU without modification. It is *never* decoded other than when it is on the GPU vertex shader.

- `QuantPoints` are submitted to the vertex shader with their octree node center. Consequently, the vertex shader can calculate the absolute position in world space of a `QuantPoint`. 8 byte of video memory are required for a single `QuantPoint` and its node center. Compared to common methods using 16 byte per point [RD10] the approach saves 50% of the memory bandwidth.

- `QuantPoints` are never handled individually other than in the vertex shader. They are always processed in octree node batches.

- `QuantPoints` are tightly arranged in memory. No bit is wasted.

6.2 Discussion

The results of this thesis indicate that PBR is a viable approach for today's mobile devices.

I estimate that PBR for mobile devices has great potential to outpace traditional polygon-based rendering for the following three reasons. First, PBR enables a constant real-time rendering performance for 3DGeoVE. Second, the mobile hardware performance constantly improves, although the screen resolution of the screens themselves remain the same given the fact that the human eye is naturally limited in its perception. Consequently, the presented solution will perform better in the future. Third, quantized point data (such as the G-Buffer) are a promising exchange format for 3D content. With this approach, it is possible to transmit only intrinsically required LOD to the client. Moreover, the 3DGeoVE can be transmitted incrementally as independent chunks of data. A self-evident requirement for this technique is a reliable and ubiquitous wireless network connection on the mobile device such as today's EDGE/3G technology or the upcoming 4G technology [Mis04].

However, as elaborated upon in Chapter 5, the approach has disadvantages as well. The response and transmission time for G-Buffer requests are the major bottleneck in the system. Moreover, the transformation into `QuantPoints` is a relevant performance factor and resampling on the client side leads to unwanted artifacts. In addition to that, certain geometry (e.g., planar surfaces) could be transmitted using less memory as traditional triangle meshes instead of the employed G-Buffers.

6.3 Future Work

The implementation presented in this thesis can be extended in several ways. The most promising are outlined in the following.

The server could serve the quantized points directly in the `QuantPoint` format as used on the client. Although the exchange format would not be a common representation anymore, the client could save a lot of CPU cycles which is spent during conversion of new data. The insertion process is computational heavy because all points are processed individually at this stage.

This also includes the WVS, which could serve the surface normal of a point with an index as it is used in the system. This would save the costly normal vector to normal index conversion.

All individual tasks (e.g., image decoding, `QuantPoint` creation, or octree processing) are optimized for a single core CPU as it is available in current mobile devices. Mobile devices in the near future will employ multi-core CPUs and therefore could leverage an according implementation.

On the client implementation, the search for a specific `QuantPoint` in a `QuantPointBlock` of an octree node is a performance bottleneck (Section 5.1.3). This could be improved by employing a Bloom filter [Blo70] which is stored in the yet unused 2 bytes of padding in the `Node` data structure (Section 4.5.1).

In general, the presented rendering engine is capable of rendering 3D meshes with textures. However, in the current implementation these functions are only applied to the Skybox. The implementation could be extended to support hybrid rendering of point-based and triangle-based 3D models. This could improve the visual quality of certain geometry such as planar surfaces.

The visual quality of the scene could further be improved on the client side by employing per pixel lightning. This could be achieved in the GPU shader using the normal which is stored in the `QuantPoint`.

References

[Ado10] Adobe Systems Incorporated. Adobe Flash Player. `http://www.adobe.com/products/flashplayer`, 2010. [Online; accessed 01-Mar-2011]. 13

[All10] Alasdair Allan. *Learning iPhone Programming: From Xcode to App Store*. O'Reilly Series. O'Reilly Media, 2010. 14

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. AK Peters Series. A.K. Peters, 3rd edition, 2008. 3, 9, 14, 17, 18, 29, 30, 35, 50, 51, 54, 59

[App10] Apple Developer Forums, Lars Schneider. Why is glMapBufferOES slower than copy via glBufferData? `https://devforums.apple.com/message/289704`, 2010. [Online; accessed 01-Mar-2011]. 58

[ARM05] ARM. Architecture and Implementation of the ARM®Cortex™-A8 Microprocessor. White paper, October 2005. 24

[Ben75] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18:509–517, September 1975. 6

[BLMM94] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard), December 1994. Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986. 20

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13:422–426, July 1970. 81

[Bou97] Tom Boutell. PNG (Portable Network Graphics) Specification Version 1.0. RFC 2083 (Informational), March 1997. 21

[Bra01] Ivan Bratko. *Prolog (3rd ed.): Programming for Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 48

[BWK02] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient High Quality Rendering of Point Sampled Geometry. In *EGRW '02: Proceedings of the 13th Eu-*

rographics workshop on Rendering, pages 53–64, Aire-la-Ville, Switzerland, 2002. Eurographics Association. vii, 7, 8, 31, 41, 48, 59

[CG02] Chun-Fa Chang and Shyh-Haur Ger. Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering. In *Proceedings of the Third IEEE Pacific Rim Conference on Multimedia: Advances in Multimedia Information Processing*, PCM '02, pages 1105–1111, London, UK, 2002. Springer-Verlag. 12

[CH10] Joe Conway and Aaron Hillegass. *iPhone Programming: The Big Nerd Ranch Guide*. Addison-Wesley Professional, 1st edition, 2010. 14

[Dan09] Daniel Filip, Google Inc. Introducing smart navigation in Street View: double-click to go (anywhere!). `http://google-latlong.blogspot.com/2009/06/introducing-smart-navigation-in-street.html`, 2009. [Online; accessed 01-Mar-2011]. 12

[DD04] Florent Duguet and George Drettakis. Flexible Point-Based Rendering on Mobile Devices. *IEEE Computer Graphics and Applications*, 24:57–63, July 2004. 9

[ETC86] Steven V. Earhart, American Telephone, and Telegraph Company. *UNIX Programmer's Manual: System calls and library routines*. UNIX Programmer's Manual. Holt, Rinehart, and Winston, 1986. 57

[FGM+99] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785. 11, 20

[Fra03] Frantisek Franek. *Memory as a Programming Concept in C and C++*. Cambridge University Press, New York, NY, USA, 2003. 24

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 25

[Gle10] Glenn Randers-Pehrson. libPNG. `http://www.libpng.org`, 2010. [Online; accessed 01-Mar-2011]. 21

[GP07] Markus Gross and Hanspeter Pfister. *Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. 3, 5, 6, 17, 35, 45, 50

[Hag10] Benjamin Hagedorn. Web View Service Discussion Paper. *Open Geospatial Consortium Inc.*, 2010. 3

[Hei98] Michael Heim. *Virtual Realism*. Oxford University Press, Inc., New York, NY, USA, 1st edition, 1998. 2

[HHD10a] Benjamin Hagedorn, Dieter Hildebrandt, and Jürgen Döllner. Towards Advanced and Interactive Web Perspective View Services. In William Cartwright, Georg Gartner, Liqiu Meng, Michael P. Peterson, Tijs Neutens, and Philippe Maeyer, editors, *Developments in 3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography, pages 33–51. Springer Berlin Heidelberg, 2010. 3

[HHD10b] Dieter Hildebrandt, Benjamin Hagedorn, and Jürgen Döllner. Image-Based, Interactive Visualization of Complex 3D Geovirtual Environments on Lightweight Devices. In *7th International Symposium on LBS and Telecartography*, 2010. 3

[Hil10] Dieter Hildebrandt. Towards Service-Oriented, Standards-Based, Image-Based Provisioning, Interaction with and Styling of Geovirtual 3D Environments. In Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Neumann, and Holger Giese, editors, *Proceedings of the 4th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering*, number 31. Universitätsverlag Potsdam, April 2010. 3

[HMHW97] Rolf H. Höhring, Matthias Müller-Hannemann, and Karsten Wiehe. Mesh Refinement via Bidirected Flows: Modeling, Complexity, and Computational Results. *Journal of the ACM*, 44(3):395–426, 1997. 2

[Hou07] Mike Houston. Understanding GPUs through benchmarking. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM. 55

[HPKG08] Yan Huang, Jingliang Peng, C.-C. Jay Kuo, and M. Gopi. A Generic Scheme for Progressive Point Cloud Coding. *IEEE Transactions on Visualization and Computer Graphics*, 14:440–453, 2008. 55

[Ind10] Independent JPEG Group. libJPEG. `http://www.ijg.org/`, 2010. [Online; accessed 01-Mar-2011]. 21

[Int92] International Telecommunication Union. JPEG Standard (JPEG ISO/IEC 10918-1 ITU-T Recommendation T.81), 1992. 21

[Kar03] Steven T. Karris. *Mathematics for Business, Science, and Technology.* Orchard Publications, 2003. 17

[KR10] Alex Kan and Jean-François Roy. WWDC 2010 Session 419 - OpenGL ES Tuning and Optimization. `http://developer.apple.com/videos/wwdc/2010/`, 2010. [Online; accessed 01-Mar-2011]. 24

[Lab10] Stanford Computer Graphics Laboratory. The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`, 2010. [Online; accessed 01-Mar-2011]. vii, 5

REFERENCES

[LG91] Didier Le Gall. MPEG: a video compression standard for multimedia applications. *Communications of the ACM*, 34:46–58, April 1991. 12

[LR98] Dani Lischinski and Ari Rappoport. Image-Based Rendering for Non-Diffuse Synthetic Scenes. In *Rendering Techniques*, pages 301–314, 1998. 6

[LS07] Fabrizio Lamberti and Andrea Sanna. A Streaming-Based Solution for Remote Visualization of 3D Graphics on Mobile Devices. *IEEE Transactions on Visualization and Computer Graphics*, 13:247–260, March 2007. 12

[LW85] Marc Levoy and Turner Whitted. The Use of Points as a Display Primitive, Technical Report TR-85022. Technical report, Computer Science Department, University of North Carolina, Chapel Hill, 1985. 3

[MBWW07] Oliver Mattausch, Jiří Bittner, Peter Wonka, and Michael Wimmer. Optimized Subdivisions for Preprocessed Visibility. In *Proceedings of Graphics Interface 2007*, GI '07, pages 335–342, New York, NY, USA, 2007. ACM. 51

[McM97] Leonard McMillan, Jr. *An Image-Based Approach to Three-Dimensional Computer Graphics*. PhD thesis, Chapel Hill, NC, USA, 1997. UMI Order No. GAX97-30561. 12

[MEH+99] Alan M. MacEachren, Robert Edsall, Daniel Haug, Ryan Baxter, George Otto, Raymon Masters, Sven Fuhrmann, and Liujian Qian. Virtual environments for geographic visualization: potential and challenges. In *Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM internation conference on Information and knowledge management*, NPIVM '99, pages 35–40, New York, NY, USA, 1999. ACM. 2

[Mey97] Scott Meyers. *Effective C++ (2nd ed.): 50 specific ways to improve your programs and designs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. 24, 32

[Mey01] Scott Meyers. *Effective STL: 50 specific ways to improve your use of the standard template library*. Addison-Wesley Longman Ltd., Essex, UK, UK, 2001. 23

[MGS09] Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. *The OpenGL ES 2.0 Programming Guide*. OpenGL series. Addison-Wesley, 2009. 15, 25, 39, 55, 57, 58, 59

[Mis04] Ajay R. Mishra. *Fundamentals of Cellular Network Planning and Optimisation: 2G, 2.5G, 3G... Evolution to 4G*. Wiley, 2004. 80

[MS95] David R. Musser and Atul Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995. 23

[NBG02] Shaun Nirenstein, Edwin H. Blake, and James E. Gain. Exact From-Region Visibility Cullin. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 191–202, Aire-la-Ville, Switzerland, 2002. Eurographics Association. 51

[Ope10] Open Geospatial Consortium, Inc. Website. `http://www.opengeospatial.org/`, 2010. [Online; accessed 01-Mar-2011]. 2

[Ped07] Paul Pedriana. EASTL - Electronic Arts Standard Template Library. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.html`, 2007. [Online; accessed 01-Mar-2011]. 23

[PKKG03] Mark Pauly, Richard Keiser, Leif P. Kobbelt, and Markus Gross. Shape Modeling with Point-Sampled Geometry. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 641–650, New York, NY, USA, 2003. ACM. 9

[PL87] Jack J. Purdum and Timothy C. Leslie. *C Standard Library*. Que Corp., 1987. 25, 31

[POW09] POWERVR. POWERVR SGX OpenGL ES 2.0 Application Development Recommendations. `http://www.imgtec.com`, 2009. [Online; accessed 01-Mar-2011]. 14, 15

[PVM+07] Kari Pulli, Jani Vaarala, Ville Miettinen, Tomi Aarnio, and Kimmo Roimela. *Mobile 3D Graphics: with OpenGL ES and M3G*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. 17

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface Elements as Rendering Primitives. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. vii, 6, 7, 10, 17

[QK05] Udo Quadt and Thomas H. Kolbe. Web 3D Service. *Open Geospatial Consortium Inc.*, 2005. 2

[RD10] Rico Richter and Jürgen Döllner. Out-of-Core Real-Time Visualization of Massive 3D Point Clouds. In *AFRIGRAPH '10: Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, pages 121–128, New York, NY, USA, 2010. ACM. 18, 34, 80

[Rid10] Philip Rideout. *iPhone 3D Programming: Developing Graphical Applications with OpenGL ES*. O'Reilly Media, 2010. 14, 24, 55, 58

[Ril06] John Riley. *Writing Fast Programs: A Practical Guide for Scientists and Engineers*. Cambridge International Science Publishing, 2006. 44

REFERENCES

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. 6, 10, 34, 50, 52

[RL01] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: A Viewer for Networked Visualization of Large, Dense Models. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, pages 63–68, New York, NY, USA, 2001. ACM. 11, 53

[Sal00] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, 2000. 8, 65

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. 35

[SGHS98] Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. Layered Depth Images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, New York, NY, USA, 1998. ACM. 6

[Sim09] Robert J. Simpson. The OpenGL®ES Shading Language. `http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf`, 2009. [Online; accessed 01-Mar-2011]. 15, 59, 61

[Sin06] Amit Singh. *Mac OS X Internals*. Addison-Wesley Professional, 2006. 32, 64

[SK97] E. B. Saff and A. B. J. Kuijlaars. Distributing Many Points on a Sphere. *The Mathematical Intelligencer*, 19:5–11, 1997. 10.1007/BF03024331. 41

[SP04] Miguel Sainz and Renato Pajarola. Point-based rendering techniques. *Computers Graphics*, 28(6):869–879, 2004. 52

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. *ACM SIGGRAPH Computer Graphics*, 24:197–206, September 1990. 3

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Vol. 1: The Protocols (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, 1994. 20

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. 24, 37

[Sut63] Ivan E. Sutherland. Sketchpad: a man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, spring joint computer conference*, AFIPS '63 (Spring), pages 329–346, New York, NY, USA, 1963. ACM. 1

[SWND07] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL®Programming Guide: The Official Guide to Learning OpenGL®, Version 2.1.* Addison-Wesley Professional, 6th edition, 2007. 15

[TSL+11] Matthias Trapp, Lars Schneider, Christine Lehmann, Norman Holz, and Jürgen Döllner. Strategies for Visualizing 3D Points-of-Interest on Mobile Devices. *Journal of Location Based Services (JLBS)*, 2011. 1

[WBB+07] Michael Wand, Alexander Berner, Martin Bokeloh, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier, Dirk Staneker, and Andreas Schilling. Interactive Editing of Large Point Clouds. In Baoquan Chen, Matthias Zwicker, Mario Botsch, and Renato Pajarola, editors, *Symposium on Point-Based Graphics 2007 : Eurographics / IEEE VGTC Symposium Proceedings*, pages 37–46, Prague, Czech Republik, 2007. Eurographics Association. 6, 7, 9, 34, 35, 36, 46

[Web11] Web3D Consortium. X3D and Related Specifications. `http://www.web3d.org/x3d/specifications/`, 2011. [Online; accessed 01-Mar-2011]. 2

[Wil08] Tim Wilson. KML. *Open Geospatial Consortium Inc.*, 2008. 2

[WSYN09] Jeong-Ho Woo, Ju-Ho Sohn, Hoi-Jun Yoo, and Byeong-Gyu Nam. *Mobile 3D Graphics SoC: From Algorithm to Chip.* John Wiley & Sons, 2009. 24

[YXZ08] Xin Yang, Duan-qing Xu, and Lei Zhao. Ray Tracing Dynamic Scenes using fast KD-tree Base on Multi-Core Architectures. In *CSSE '08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, pages 1120–1123, Washington, DC, USA, 2008. IEEE Computer Society. 35

[ZPKG02] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross. Pointshop 3D: An Interactive System for Point-Based Surface Editing. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 322–329, New York, NY, USA, 2002. ACM. 9

Declaration

I herewith declare that I have produced this thesis without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other German or foreign examination board.

The thesis work was conducted from 01.10.2010 to 01.04.2011 under the supervision of Prof. Dr. rer. nat. habil. Jürgen Döllner and Dipl. Inform. Dieter Hildebrandt at Hasso-Plattner-Institut, Potsdam, Germany.

Potsdam,