Parallel simplification and compression of reality captured models

Koh, Naimin

2019

Koh, N. (2019). Parallel simplification and compression of reality captured models. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/136785>

<https://doi.org/10.32657/10356/136785>

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).

Downloaded on 26 Apr 2023 17:49:30 SGT

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

**PARALLEL SIMPLIFICATION AND
COMPRESSION OF REALITY CAPTURED
MODELS**

KOH NAIMIN

**SCHOOL OF COMPUTER SCIENCE AND
ENGINEERING**

2019

**PARALLEL SIMPLIFICATION AND
COMPRESSION OF REALITY CAPTURED
MODELS**

KOH NAIMIN

**SCHOOL OF COMPUTER SCIENCE &
ENGINEERING**

A thesis submitted to Nanyang Technological University in partial fulfillment of the
requirements for the degree of
Master of Engineering

2019

STATEMENT OF ORIGINALITY

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

19 July 2019

Date

A handwritten signature in black ink, appearing to read 'Koh Naimin', written over a horizontal dotted line.

Koh Naimin

SUPERVISOR DECLARATION STATEMENT

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

19 July 2019
.....
Date

.....
Assoc Prof Zheng Jianmin

AUTHORSHIP ATTRIBUTION STATEMENT

This thesis contains material from 1 paper published in the following peer-reviewed journal(s) / from papers accepted at conferences in which I am listed as an author.

Chapter 4 was published as Naimin Koh, Wenjing Zhang, Jianmin Zheng, and Yiyu Cai. GPU-based Multiple-Choice Scheme for Mesh Simplification. Proceedings of Computer Graphics International 2018, pages 195-200. DOI: 10.1145/3208159.3208195.

The contributions of the co-authors are as follows:

- Assoc Prof Zheng provided the initial project direction and edited the manuscript drafts.
- Dr. Zhang prepared the manuscript drafts. The manuscript was revised by Assoc Prof Zheng and Assoc Prof Yiyu Cai.
- I co-designed the study with Assoc Prof Zheng and performed all the implementation and experimentation of the method proposed on simplification.
- Assoc Prof Zheng and Dr. Zhang help to analyze the collected result and proposed improvement upon the initial implementation.

19 July 2019

.....
Date

.....
Koh Naimin

ACKNOWLEDGMENTS

I would like to thank my supervisor, Associate Professor Zheng Jianmin for his continuous guidance and support.

I would also like to thank Dr. Zhang Wenjing and Dr. Pradeep Kumar Jayaraman for their patience and great guidance with our paper submission.

I thank Autodesk Asia for partially sponsoring my tuition fees and giving me time off to pursue my studies. Massive thanks for the Reality Solutions team for the continuous support throughout this whole journey.

A great big thank goes to my family for being supportive and understanding throughout this period.

TABLE OF CONTENT

STATEMENT OF ORIGINALITY.....	I
SUPERVISOR DECLARATION STATEMENT	II
AUTHORSHIP ATTRIBUTION STATEMENT	III
ACKNOWLEDGMENTS	IV
TABLE OF CONTENT.....	V
ABSTRACT.....	IX
CHAPTER 1: INTRODUCTION	1
1.1 REALITY CAPTURE RECONSTRUCTION PIPELINE.....	1
1.2 INTRODUCTION TO REALITY CAPTURE TECHNOLOGY	4
1.2.1 <i>Laser Scanner</i>	4
1.2.2 <i>Photogrammetry</i>	5
1.2.3 <i>Simultaneous Localization and Mapping (SLAM)</i>	7
1.3 REPRESENTATION OF REALITY CAPTURED MODELS	8
1.3.1 <i>Point Cloud</i>	9
1.3.1.1 Structured Point Cloud	10
1.3.1.2 Unstructured Point Cloud	10
1.3.2 <i>Triangle Mesh</i>	11
1.3.2.1 Triangle Mesh.....	11
1.3.2.2 Textured Triangle Mesh.....	12
1.4 PROBLEM STATEMENT.....	12
1.5 SCOPE AND OBJECTIVES	13
1.7 CONTRIBUTIONS OF THE PROJECT.....	14
1.8 THESIS ORGANIZATION	16
CHAPTER 2: RELATED WORKS	17
2.1 DENSE POINT CLOUD COMPRESSION	17

2.1.1 General Floating-Point Data Compression	17
2.1.2 Image-Based Point Cloud Compression	18
2.1.3 Spatial Hierarchy Based Compression	19
2.2 TRIANGULAR MESH SIMPLIFICATION	20
2.2.1 Mesh Decimation Operations	21
2.2.1.1 Vertex Decimation	21
2.2.1.2 Vertex Clustering.....	21
2.2.1.3 Edge Collapse	22
2.2.2 Computationally Efficient Decimation	22
2.2.2.1 Computational Efficiency	23
2.2.2.2 Memory-less and Out-of-Core.....	24
2.3 TEXTURE AWARE DECIMATION	24
CHAPTER 3: PARALLEL POINT CLOUD COMPRESSION USING TRUNCATED OCTREE	27
3.1 INTRODUCTION.....	28
3.2 PROPOSED METHOD	30
3.2.1 Bottom-Up Octree Construction	30
3.2.1.1 Octree Representation	31
3.2.1.2 Octree Construction.....	32
3.2.2 Octree Truncation.....	34
3.2.2.1 Variable Length Addressing Scheme	38
3.2.2.2 Offset Jump Address	39
3.2.3 Encoding	40
3.2.3.1 Header Information	40
3.2.3.2 Body Information	40
3.2.4 Decoding.....	41
3.3 EXPERIMENT	43
3.3.1 Quantization Error.....	44
3.3.2 Compression Results.....	45
3.3.3 Performance	46
3.4 SUMMARY	53
CHAPTER 4: GPU BASED MULTIPLE CHOICE SCHEME MESH SIMPLIFICATION	55

4.1 INTRODUCTION.....	55
4.2 METHOD OVERVIEW	56
4.3 QEM-BASED EDGE COLLAPSE.....	56
4.4 MULTIPLE-CHOICE DECIMATION (MCD).....	58
4.5 PARALLEL COLLAPSE BASED ON SUPER INDEPENDENT VERTICES	59
4.6 PROPOSED METHOD	60
4.6.1 <i>Data Initialization</i>	61
4.6.2 <i>Candidate Edges Identification</i>	61
4.6.3 <i>Independent Edges Identification</i>	62
4.6.4 <i>Topology Update</i>	62
4.6.5 <i>Iteration Check</i>	62
4.6.6 <i>Putting it all together</i>	62
4.7 EXPERIMENT	63
4.7.1 <i>Experimental Results</i>	63
4.7.2 <i>Comparison</i>	68
4.8 SUMMARY	68
4.8.1 <i>Future Works</i>	69
CHAPTER 5: SIMPLIFICATION OF TEXTURED MESH.....	71
5.1 INTRODUCTION.....	72
5.2 PROPOSED METHOD	74
5.3 SEPARABILITY OF GEOMETRIC AND TEXTURE SIMPLIFICATION	75
5.3.1 <i>Geometric Simplification</i>	75
5.3.2 <i>Texture Simplification</i>	76
5.3.2.1 Resizing the Texture.....	76
5.3.2.2 Regenerating the Texture Mapping	77
5.3.2.2.1 Mesh Parameterization.....	77
5.3.2.2.2 Texture Packing	78
5.3.3 <i>Section Summary</i>	78
5.4 PARALLEL TEXTURE MAP TRANSFER.....	79
5.4.1 <i>Algorithm Overview</i>	79
5.4.1.1 Definitions	80

5.4.1.2 Overview	80
5.4.1.3 Rasterization	81
5.4.1.4 Ray-Casting.....	82
5.4.1.5 Generating Textures	85
5.4.1.6 Accelerated Ray-Casting	85
5.4.1.7 Parallelizing	87
5.5 EXPERIMENT	88
5.5.1 <i>Other Application</i>	92
5.6 SUMMARY	93
5.6.1 <i>Future Works</i>	93
5.6.2 <i>In Closing</i>	94
CHAPTER 6: REAL WORLD DEMONSTRATION	95
6.1 RECONSTRUCTION PIPELINE SETUP	96
6.1.1 <i>Photogrammetry Inputs</i>	96
6.1.2 <i>Reconstruction Process</i>	98
6.2 EXPERIMENT	102
CHAPTER 7: CONCLUSION	109
7.1 APPLICATION	110
7.2 LIMITATION	111
7.3 FUTURE WORKS	111
REFERENCES	113

ABSTRACT

Reality capture technologies such as laser scanner and photogrammetry are becoming more democratize day-by-day, as such we are currently undergoing an influx of high-resolution photo-realistic 3D models that contain ever-increasing geometric and texture details and resolution. The massive data file size required to store this information can cause difficulty for both the transfer and manipulation of the captured model. This problem is even more prevalent for mobile or handheld capture devices with limited available internal disk space on the device itself.

The three main outputs of a complete reality capture pipeline are dense point cloud, 3d mesh model and texture maps. To reduce the size of each of these outputs, we employ data compression and simplification techniques while striving to retain as much quality as possible. Given the large initial input size and expected long processing time, we explored specifically parallel algorithms in order to fully utilize modern multi-core CPU and GPU to accelerate the computation.

This thesis focuses on the parallelized simplification and compression algorithm for large-scale point cloud, 3D mesh, and textures. These input models are generated from running the full reconstruction of a photogrammetry 3D reconstruction pipeline. We have investigated and proposed parallelizable methods to reduce file size for each of the output types while still managed to retain high visual quality like the raw captures.

CHAPTER 1: INTRODUCTION

This research, “Parallel Simplification of Reality Captured Models”, was conducted in partial fulfillment of the requirements of the Nanyang Technological University Master of Engineering Program by Research. This chapter will present the background knowledge of reality capture technologies and key motivations for this thesis.

Reality capture technology is an emerging class of technologies that concern itself with the accurate capture and digitalization of real-world scenes or objects into a digital model format with the help of scanning hardware and computer vision techniques. The resulting digital models are used for a variety of purposes ranging from large scale city-planning and disaster prevention planning, digital preservation of irreplaceable artifacts, to the studies of microscopic organisms such as water bears (Tardigrades).

We will review the general reality capture reconstruction pipeline, then briefly describe the three main class of reality capture technologies: laser scans, photogrammetry and simultaneous localization and mapping (SLAM). Followed by an introduction to the three main model representation resulting from the reconstruction pipeline: dense point cloud, mesh and textured mesh.

1.1 Reality Capture Reconstruction Pipeline

We will describe how a general multi-modal reality capture reconstruction pipeline can be designed. With the point cloud representation being the common ground between the three main reality capture technologies, it allows for a unified pipeline where after obtaining the point cloud model, the different pipelines can all converge into a single mesh reconstruction flow.

Figure 1.1: Architecture of a multi-modal reality capture reconstruction pipeline.

Figure 1.1 demonstrates a multi-modal reality capture reconstruction pipeline, that allows for the combination of various reality capture technology into a single unified pipeline. Since each method will generate a dense point cloud as output, the subsequent mesh-based reconstruction pipeline can be unified into a singular process. Given the point cloud with normal information, we can perform the surface reconstruction [1]–[3] process to generate the mesh representation of the model. Commonly used surface reconstruction algorithm used is the screened Poisson surface reconstruction [3], [4]. After the generation of the mesh, we will perform mesh parametrization to generate the texture mapping information for the mesh [5]–[8]. With the texture mapping information, we can then perform texture projection back into the source input data to retrieve the color information from the source input data [9]. This is possible because the various registration process has already aligned the source input data with the current 3D coordinate of the mesh, which allows us to map 3D coordinate on the mesh back to the 2D location in the photos, videos or range image to retrieve accurate color information.

The inputs for the photogrammetry pipeline consist of primarily photographs taken either on foot or aerial via unmanned aerial vehicles (UAV) or planes. With the advent of global position system (GPS), this information is also used by the photogrammetry

pipeline to geolocate the scene in the correct real-world geolocation. The inputs for the Lidar/laser scanner pipeline are as the name implied comes from the terrestrial or handheld laser scanner. Using the time-of-flight after firing the laser, a laser scanner can with a very high precision measure the environment around it. The output format of this laser scan data usually comes in the form of either a range image (explained in section 1.2) or a structured point cloud format. The input of the SLAM pipeline can come from various sources to help in the localization and mapping, the primary inputs are the video stream or high-frequency depth sensor. With a calibrated video camera or sensor, the capturing device can in real-time estimate the depth of the environment for each frame captured by the device. The resulting output of the capturing device can either be the raw video stream with calibration information or the estimated depth map for each frame captured. Each of the outputs of the various capturing device will all go through a process of registration to aggregate the captured information into single global world space, after which further noise filtering and error correction processes can be run to remove the inevitable real-world noise captured. After these processes, surface reconstruction is done to generate the mesh out of the cleaned point cloud, this mesh, in turn, goes through texture parameterization and texture projection to arrive at the output of a textured mesh. The textured mesh is now ready for rendering and visualization or dropped into a virtual reality game environment.

At the end of this reconstruction pipeline, we will be left with three major outputs: dense point cloud, mesh and textured mesh. Therefore, we focus our research on these three representations of the reality capture reconstruction pipeline as the subject of this thesis. Because most reality capture reconstruction pipeline follow this general pipeline [10], our contribution to simplification of these three representations can be used regardless of which variant of reality capture technology is being used.

1.2 Introduction to Reality Capture Technology

1.2.1 Laser Scanner

Figure 1.2: Examples of various types of laser scanning devices.

Images courtesy of SPAR3D [11], Artec 3D [12] and Microsoft [13].

The more commonly used specialized hardware for laser scanning are the following: terrestrial LiDAR scanner, handheld laser scanner, and infrared scanner. LiDAR scanner has traditionally been used at construction sites or surveying infrastructures due to their high accuracy and reliability compared to other measuring tools for building scale measurements [14]. The advent of handheld laser and infrared scanners allows for better indoor scanning of smaller objects for replication or quality controls.

They all utilize line-of-sight projection of laser and the subsequent reflection of the laser back onto the scanner's sensor to determine the distance between the surface and the scanner. The distance can be determined by either the attenuation of the strength of the reflection or the time-of-flight for the laser to reflect back to the scanner. With this distance information usually stored as a range image, and the hardware-specific

projection matrix, one can reconstruct a dense point cloud of a localized scene from a single scan [15], [16].

With a high-resolution scan capable of millimeter resolution between each point, each localized scan is typically in the size of several hundred megabytes in size up to one or two gigabytes in size. The resulting file size is highly dependent on the sensor used.

Registration of multiple scans allows for the combination of several localized scans into a single global registered scan. By detection similar features between two scans, a scan registration software can automatically combine individual scans together [17], [18].

One can easily notice, that the combination of these many localized scans can easily result in a complete registered scan that will be several gigabytes in file size at the very least.

1.2.2 Photogrammetry

Figure 1.3: Examples of the photogrammetry capture rig. Images courtesy of Quantum Capture [19].

Photogrammetry is a reality capture technology that allows the conversion of real-world photographs into digital 3D models. By taking photos from many views of the subject of capture, one can accurately reconstruct the digital form of the object through photogrammetry. While the core theory and mathematics behind photogrammetry was discovered long ago [20], [21] and was manually done by hand for aerial images [20], [21], the vast improvement in computing power and optimization of the bundle adjustment solver only recently allow for efficient reconstruction on a large scale [22], [23].

Utilizing computer vision to detect high-level features within a photo [24], followed by a process to match the detected features from one photo to the other photos [25], [26]. Once all the matched photos are found, a bundle adjustment optimization is done on the features and all related features to optimize and estimate both the intrinsic (i.e. lens distortion and focal length) and extrinsic (i.e. position and rotation of the camera) information of all the camera's view which took the photos [27], [28]. The result of the bundle adjustment will be the intrinsic and extrinsic information of all camera views as well as the 3D positions of the corresponding 2D features detected earlier. This whole process is generally term as part of the Structure from Motion (SfM) pipeline [29], [30].

Using the result from the SfM process, a Multi-View Stereo (MVS) process is done to compute a denser point cloud representation of the model [31]–[34]. This representation is considered denser compared to the point cloud resulting from the SfM pipeline which only include points that have 2D features detected (ranging from few thousands to tens of thousands), while MVS can generally create a point for each pixel shared between related views (ranging from few hundreds of thousands to millions), resulting in a vastly denser point cloud compared to SfM's result. MVS uses the intrinsic and extrinsic information of each view to determine if any two views are related. After all the related views are detected, they will be used to generate a depth map for each view, which can be easily converted into point cloud by projecting the point from the depth map using the view's information.

Triangular mesh reconstruction can be done once the dense point cloud is generated from the MVS process. Using Poisson surface reconstruction [3] is the most common

due to the high quality of mesh reconstructed and the capability to reconstruct chunk by chunk of the underlying octree instead of one massive global mesh all at once.

1.2.3 Simultaneous Localization and Mapping (SLAM)

Simultaneous localization and mapping are a rapidly growing field of research having roots in robotics and recently accelerated by the growth in autonomous self-driving vehicles [35]. This class of technologies concerns themselves in real-time surrounding environment reconstruction and agent localization within the reconstructed environment, hence the name. This is highly important to allow for the development of surrounding-aware and dynamic path planning capability for autonomous robots or vehicles when placed in a new environment. While SLAM does use technologies from the laser scanner and computer vision heavily, the emphasis on real-time reconstruction usually comes at the expense of reconstruction quality, which results in lower quality scene reconstruction compared to laser scanning and photogrammetry [36], [37]. The capability to rapidly reconstruct wide areas or the entire interior of the building should not be undermined by the disadvantage of the technologies.

SLAM either utilize mounted laser scanner or calibrated stereo cameras to allow for real-time generation of depth information of the currently observed scene. By integrating and registering each incoming depth frame with the information already gathered, a global scene is reconstructed and updated part by part. Through this approach, one can gradually reconstruct an entire area by scanning one part at a time.

While the surrounding maps are being reconstructed, the new local position of the agent must be updated as well for the agent to figure out where exactly it is now. This is usually done through a combination of GPS or accelerometer sensors on the agent and relative position derived from the incoming depth frames. Using both allow for the correction of errors resulting from either approach.

While in theory, all this sounds easy, just scan and combine right? The truth, however, is that noises and artifacts inevitably exist for any kind of physical capturing device. When integrating new depth information into the existing model, one needs to be able

to determine whether the new information is correct and not noise. Due to the incremental nature of SLAM, the accumulation of error lead to a well-known issue of drifting [38], [39].

1.3 Representation of Reality Captured Models

This thesis will explore the improvements on the simplification and compression algorithms of reality captured models. A brief introduction to the various digital representation of the reality captured models generated along the reconstruction pipeline will be presented here. This chapter will talk about the point cloud representation, followed by the triangular mesh representation and lastly the textured mesh representation.

1.3.1 Point Cloud

Figure 1.4: Example of a colored unstructured point cloud.

Point clouds are a type of digital model representation that consists of millions of points that lay on the surface of the model, which when observed from a distance will approximate the geometric structure of the captured model. In order to accurately approximate a surface with only infinitesimal small points, a very high density of points is required for even a small patch of the surface.

There are two main classifications of the point cloud in the field of reality capture technology: structured point cloud and unstructured point cloud.

1.3.1.1 Structured Point Cloud

This class of point cloud is a more specialized point cloud format that is usually generated by Lidar scanner or other specially calibrated scanners [40]. This point clouds are usually stored in a range image, which is a distorted image that holds the depth information of each point in each individual pixel of the range image [16]. Range images are created after every individual scan by a lidar scanner. By using the calibrated intrinsic matrix of the Lidar scanner, one can undistort and project the depth value of the range image into the complete point cloud.

The structured point cloud is thus named because they have implicit information about the sensor structure. This sensor structure can allow for regeneration of point normal and relative position since the range image is captured from a stationary position (i.e., the laser sensor). Similarly, by exploiting the spatial relationships of the pixels in the range image and sensor intrinsic property, one can efficiently reconstruct the point cloud from a more compact range image format.

1.3.1.2 Unstructured Point Cloud

The unstructured point cloud is a set of unorganized points without any explicit structure behind it. This is the lowest level representation of a point cloud since it does not rely on any implied external structure information, which allows this representation to serve well as an intermediate format for both computer visions and computer graphics.

However, due to the lack of structured information, any additional attribute of the points needs to be carried explicitly by the unstructured point cloud format. This greatly restricts the avenue available for compression of the unstructured point cloud in contrast to the structured point cloud.

In order to derive additional structure from an unstructured point cloud, statistical methods are usually used to infer the implied surface represented by the point cloud, which allows for an approximation of the points' normal [41], [42].

1.3.2 Triangle Mesh

Figure 1.5: Example of a textured mesh.

1.3.2.1 Triangle Mesh

Triangle meshes are a polygonal mesh that is commonly used in computer graphics. The mesh is formed of many sets of triangles that share common vertices and edges with other triangles. This representation allows for a more efficient representation of surface as opposed to the point cloud representation. For example, a single triangle can be used to represent any arbitrary large surface area while a point cloud will require many thousands of points to approximate the same area covered by a single triangle. This efficiency is why the triangle mesh is so commonly used in computer graphics for rendering purposes.

Due to the connectivity information between triangles being available, a higher level geometric algorithm can be utilized for the purpose of mesh simplification [43], [44].

1.3.2.2 Textured Triangle Mesh

Unlike the point cloud where we can assign specific attributes to an individual point, attributes of the triangle's vertices are usually interpolated across the triangle surface to approximate the same effect. However, if the attributes cannot be interpolated smoothly such as texture color information, a mapping is used to map the location on the triangle to a corresponding buffer where the specific attribute was stored.

In the context of texture information, this mapping is called the texture mapping which maps a 2D point on the triangle to another 2D location in a texture image. This allows the mesh to be able to have a much better texture representation than what the simple smoothing of vertex color can achieve [45].

The process to generate this texture mapping is called the mesh parameterization, which seeks to find the bijective mapping between two surfaces. Methods like LCSM and ABB mapping exist that can generate low distortion texture mapping [5]–[7]. While an interesting field of research in and of itself, it is largely out of the scope of this chapter.

For reality captured models the texture themselves are usually generated from actual real-world photos or video inputs. This greatly improves the visual and photorealism when applied onto the mesh model [46], [47].

1.4 Problem Statement

As we discussed earlier, reality capture technologies are an emerging class of methods that uses a combination of hardware sensor and computer vision techniques to accurately capture and reconstruct a digital representation of a real-world object or scene. Hardware such as terrestrial LiDAR scanner, handheld laser scanner or infrared scanner can generate a massive amount of data within a short span of capturing time. Likewise, photogrammetry that makes use of computer vision to stitch and triangulate multiple pairs of photos together into a consistent scene, are easily capable of generating a 3d point for every pixel of the pair of photos used for triangulation.

The relative ease of capturing and generating a massive amount of data using reality capture technologies is creating an interesting class of problem which causes the classical algorithm to start faltering under the sheer volume of data to process. The traditional 3D model created using computer-aided design (CAD) or artist modeling is predominantly of much lower resolution and consist of less organic structures, unlike the reality captured models that fully captured the organic surface of the real-world object which in turn requires a high geometric resolution in order to accurately capture all the details. This vast difference in geometric resolution of the reality captured model applied pressure upon the classical simplification algorithms which were designed for thousands of triangles mesh and not hundreds of millions of triangles.

Specifically, while runtime of classical state of the art simplification methods [48], [49] are acceptable when operating on CAD models, but when applied on the scale of reality captured model these methods start to struggle both in memory usage and computation performance due to the highly serial and global optimization of these methods.

Another important aspect of the classic methods was that because they are designed primarily for non-organic models, they can make certain quality tradeoffs while negligible on an angular and flat structure. It will negatively impact the visual quality of rough and organic surfaces. While one can perform aggressive compression and simplification without regard to the final output quality, doing so will defeat the purpose of capturing such high-resolution capture in the first place. However, earlier methods tend to overlook the ability to control the quality tradeoff against the compression rate, making it difficult to maintain a balance between retention of captured quality and density of the data size.

1.5 Scope and Objectives

While there are many open questions remain in the field of reality capture model, in order to be able to realistically handle the captured data at this scale, we scope our research on the compression and simplification of the reality captured model to more usable size for classical down-stream processing. We focus our research on highly

parallelizable and memory-efficient algorithms in order to properly handle the large input size of the raw uncompressed model.

We then further focus our compression research on the three important digital representation of the real object: the dense point cloud, triangular mesh, and textured mesh representation. These representations are selected as our focus as they represented the most utilized representation of reality captured models generated during the digitalization and reconstruction of the real-world object.

With the problem and scope in mind, we wish to study algorithms that have a well-defined trade-off between file size and representation quality, that allows for users to clearly control the quality at which the simplification is deemed acceptable. After simplification, the model will be able to be further processed using classical downstream processing that was unable to handle the raw uncompressed scale of the original model.

The goal of reducing the overall size of the captured data has many great benefits for the downstream processing, we must still acknowledge that the processing of the initial massive amount of raw data needs to be performed in a highly efficient manner. Hence a need to research computationally and memory-efficient methods.

With the earlier three objectives in mind, this research aims to improve both the compression/simplification and computation performance of high-quality reality captured models while also providing a well-defined control on simplification quality loss.

1.7 Contributions of The Project

In this thesis, we will present in detail three separate contributions on the simplification or compression dealing with the major model representation (point clouds, triangular meshes and textured 3d models) generated from both the photogrammetry and laser

scanning reconstruction pipeline. We also present a demonstration of a real-world use case utilizing the three contributions in conjunction with each other.

Relating to the compression of unstructured point clouds, we proposed the method: “Parallel Point Cloud Compression Using Truncated Octree”. We are currently working on getting this work submitted to a journal.

The next contribution proposed a GPU based scheme to massively parallelize triangular meshes simplification. Our paper: “GPU-based Multiple-Choice Scheme for Mesh Simplification” was accepted into Computer Graphics International 2018 (CGI 2018)

The third contribution listed in this thesis seeks to break the usual limit of simplifying a textured mapped mesh due to the restriction required to maintain texture mapping information. Our method: “Texture Map Transfer” allow for the transferring of high-quality texture information from one model to another more simplified model.

Lastly, we demonstrate a complete photogrammetry reconstruction pipeline that utilized the three listed contributions to simplify the various outputs of the reconstruction. This highlights the practical usage of the contribution and shows its successful utilization in a real-world use case.

1.8 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 reviews some related works that form the foundation of this thesis.

Chapter 3 discusses the parallel unstructured point cloud compression using the truncated octree method applied to compress point cloud data.

Chapter 4 describes the multiple-choice optimization technique applied to triangular mesh simplification.

Chapter 5 discusses the texture mapping transfer method that assisted in the simplification of textured mapped models.

Chapter 6 demonstrates the real-world utilization of the listed contributions.

Chapter 7 concludes the thesis and briefly discusses the remaining future work.

CHAPTER 2: RELATED WORKS

This chapter will first review the related dense point cloud compression methods. Then a taxonomy of core triangular mesh decimation operation and state-of-the-art simplification methods are explored. The last part will discuss the various texture aware mesh simplification methods.

2.1 Dense Point Cloud Compression

Largely due to the recent advancements in remote sensing, high-resolution 3D scanning and simultaneous localization and mapping (SLAM) technologies driven by recent growth in the autonomous self-driving car industry, vast amounts of point cloud data are generated leading to increased research interest in the field of point cloud compression for efficient data storage, retrieval, and processing.

2.1.1 General Floating-Point Data Compression

Since point clouds are floating-point data, it is worth considering methods that perform lossless compression of raw floating-point information.

Ratanaworabhan et al. [50] proposed a fast lossless floating-point compression that managed to achieve a peak compression rate of 1.5 times compression.

More recently Lindstrom [51] proposed a block-based float-point compression scheme that can achieve a higher compression rate for both lossy and lossless compression modes. Despite the high 1.5--4.0 times compression ratio even on lossless mode, the author, however, noted that their methods should not be used to compress unstructured data such as triangle mesh geometry or unorganized point sets. This is because their

method exploits the spatial correlation property of the data for the high level of block compression.

Since our focus is on unstructured point clouds without a structured spatial correlation, we did not further pursue this line of research.

2.1.2 Image-Based Point Cloud Compression

Another direction of research adapts image and video compression algorithms to work with point cloud data. They do this by projecting the 3D point cloud into 2D images or frames and then perform the video compression on the frames.

Chen et al. [52] encoded positional, color and normal information of the point cloud by converting them into 3 space-filling curves which are encoded into each of the three channels of an RGB image.

Shao et al. [53] and Cohen et al. [54] used a block-based intra prediction model for multi-frame range images, where each 2D range image is further subdivided into smaller blocks and video compression algorithms are used on these smaller blocks as an encoder.

Since all the information of the point cloud is now encoded in the 3 channels of the image, normal image compression algorithms can be used to obtain a high level of compression.

Because of the usage of image or video compression as the encoder, the quality of the point cloud is highly dependent on the quality of the image compression algorithm used. This also makes it difficult to control the tradeoff between file size and point cloud quality.

In addition, the restriction of using an image as a storage container has a limiting effect on the maximum precision of the information contained within the channels of an image which makes it difficult to extend the method to obtain higher precision.

Furthermore, the image size is quadratically dependent on the desired quantization error. Since image and video compression methods make heavy use of spatial correlation of nearby pixels, the efficiency of these methods will be severely penalized applied when to unstructured data.

The lack of accurate control over output precision and the unsuitability of working with an unstructured point cloud was the main reason why we chose to use the more flexible spatial hierarchy-based representation.

2.1.3 Spatial Hierarchy Based Compression

Spatial hierarchical representations such as octrees and KD-trees have been used in point cloud visualization [55] and compression for a long time. Many works exploit the uniform hierarchy structure of the various spatial hierarchy to derive higher-level spatial information on the point cloud distribution. Octrees are commonly used since their coding efficiency far surpasses that of KD-trees [56].

Merry et al. [57] focused on uniformly dense and regular point clouds generated by laser scanner using a spanning tree as the spatial hierarchy which is both computationally expensive and inflexible to the distribution of the point cloud. Huang et al. [58] used an octree to guide the compression of their geometry and first proposed the idea of child occupancy code that we later use in our encoding scheme.

Child occupancy code is a 1-bit code for indicating the existence of each possible child of a given node. In the case of an octree, the code fits nicely into a single byte since there are exactly 8 possible children under each octree node. As opposed to addressing each child directly, child occupancy code is a more compact representation compared to addressing individual children (constant 8 bits vs 3-24 bits). This method further attempts to reorder their child traversal order to reduce the entropy of their occupancy code in regard to the probability of the children existing or not, which is used by their probabilistic decoding scheme. However, the occupancy code re-ordering is not necessary for schemes that do not use probabilistic decoding, such as ours.

2.2 Triangular Mesh Simplification

Mesh simplification is a fundamental process in many applications like 3D reconstruction, real-time rendering and virtual reality [59], [60]. The purpose of simplification is to reduce the number of triangles while keeping the simplified mesh as close to the original mesh as much as possible [61]–[63]. This process, however, is usually quite slow for meshes with many vertices. Many parallel algorithms proposed to speed up the simplification by exploiting the parallelism provided by the GPU. To do parallel simplification, the algorithms need to deal with conflict carefully: different threads cannot modify the overlapped part of the mesh. Such a racing problem is the focus of the existing methods. As a result, the simplified mesh may have low quality compared to a sequential greedy algorithm running on the CPU. While the speed is important for the simplification, quality of simplified mesh also plays much importance.

There are two general problems in the research of mesh simplification of reality captured models.

The first problem was due to the high-resolution reality capture technique used; the raw captured model will likely retain very high geometric complexity. As this complexity of the model increases, the pressure placed on the efficiency in both computation time and memory requirement of the simplification algorithm also increases. Thus, we researched both parallelizable and memory-less simplification algorithms. Additionally, the mesh quality of the new simplification method must still achieve similar visual quality as current state-of-the-art simplification methods.

The second problem was that most current state-of-the-art simplification methods primarily focus on the optimal simplification of the texture-less model and managed to achieve remarkable quality, however, reality captured models are usually reconstructed with texture information in addition to the geometric information. To disregard this texture information will greatly impact the value of reality captured model, as humans perceive texture information far greater than geometrical shapes. To tackle this problem without making a compromise on either the geometric or texture quality, we will research into current texture preserving simplification methods.

2.2.1 Mesh Decimation Operations

Mesh simplification found in the literature was broadly classified into these types of decimation operations: vertex decimation, vertex clustering, edge collapse.

2.2.1.1 Vertex Decimation

Figure 2.1: Vertex Decimation

Vertex decimation algorithm [64] iteratively select a single vertex and removes it and all of its connected edges and faces, leaving behind a surface hole. The resulting hole is then covered by re-triangulating over it.

2.2.1.2 Vertex Clustering

Figure 2.2: Vertex Clustering

Vertex clustering [65] based on the idea that vertices that are spatially near to each other could be clustered together into a single vertex, hence achieving the goal of reducing the number of vertices and faces of the model. Generally, a spatial grid either

uniform or octree was used to partition all the vertices into their respective cell and subsequently all the vertices within each cell is combined into a single vertex.

However, since the only criteria for merging vertices is the spatial distance (usually defined as the size of the spatial cell), this can often lead to inconsistency in the topology of the mesh after clustering. Secondly, since all vertices within a cell are collapse together, it is difficult to control the exact amount of simplification to perform.

2.2.1.3 Edge Collapse

Figure 2.3: Edge Collapse

Edge collapse algorithms [63], [66], [67] iteratively select an edge and collapse the edge's two vertices together in a new position that minimizes an error metric score. This operation will generally remove two faces except when the edge is a boundary edge, in which case only one face would be removed by this operation.

2.2.2 Computationally Efficient Decimation

For this report, we will investigate the efficiency of decimation algorithms in terms of computational and memory efficiency since we are targeting a massive model, both criteria are significant areas for improvement.

2.2.2.1 Computational Efficiency

General-Purpose Graphics Processing Units (GPGPU) is used increasingly for parallel computation. The field of mesh decimation also benefited from this trend. Many researchers were dedicated to speeding up the mesh simplification algorithm using the GPU [68]–[71]. Their focus is on how to select the best set of edges for collapsing without incurring conflicts and minimizing geometric error.

[68] utilizes an octree to partition the 3D space to isolate the edges from one cell from the other cells. The simplification is thus dependent on how well the mesh was partitioned. Specifically meshes with non-uniform mesh density tend to produce poorer simplifications results.

[70] choose to collapse edges with the local minimum cost, which guarantees the selected edges are independent of other selected edges. However, a local minimum edge does not ensure global minimum edge cost causing quality issues. The computational efficiency is also dependent on the amount of local minimum edges detected.

Shontz and Nistor [71] use a CPU-GPU algorithm where a major part of the algorithm was executed on the CPU and a certain part of the algorithm was executed on the GPU. It was noted that the transferring of data back and forth the CPU and GPU is highly inefficient and it should be avoided as indicated in many other related GPU-based types of research.

Super independent vertices [72] were adopted to guarantee the selected edges are independent of each other. However, the quality of the simplified mesh generated by these algorithms could be slightly lower than that of a greedy-based sequential algorithm due to the selection of non-global minimum cost edges.

While many existing methods use a triangle soup representation in their work, a more recent work utilized the half-edge data structure in their parallel simplification. Though our method adopts the triangle soup similar to [69], the idea of using a probabilistic multiple-choice scheme to obtain better quality can also be incorporated into other data representations such as the half-edge representation.

2.2.2.2 Memory-less and Out-of-Core

As the laser scan model usually comes in gigabytes of the raw data file, the corresponding requirement to store the model completely in-memory generally will be larger due to the uncompressed nature of the in-memory representation. This problem leads us to research into an algorithm that requires minimal memory overhead or an out-of-core algorithm.

Lindstrom-Turk [67] first introduced a memory-less simplification method that requires only very little memory overhead in addition to the in-memory model, while also having the property of volume preservation. As studied by Hoppe [48] memory-less algorithms not only reduce memory requirement but also improve simplification quality along with computational speed when compared to traditional QEM based decimation.

Wu and Kobbelt [73] instead introduced the use of Multiple Choice Optimization (MCO) [74], [75] which uses a probabilistic approach to randomly select which are the edges that will most likely to be collapsed eventually and collapse them. Since the choice is probabilistic, there is no need to maintain a global structure of all the edges' QEM to aid edge selection in traditional methods. This removal of the global structure also lends itself well to parallelism which we will explore in our work. Building upon their earlier work, Wu and Kobbelt introduced an out-of-core streaming algorithm [76] that showcase the flexibility of the MCO-based decimation.

A more recent development in out-of-core mesh simplification is the works by Ozaki et al. [49] that improved upon the local minimal edge introduced by Grund et al. [70] and extended their idea to allow for out-of-core decimation.

2.3 Texture Aware Decimation

Although much research has gone into improving the geometric aspect of the mesh simplification, there is relatively much less research that tackles the issues of simplifying a textured mesh.

One of the most common ways to preserve texture information on a model while decimating it was simply to disallow the removal of edges that have vertices that have texture coordinate that lay on the boundary of texture charts. Otherwise, it will lead to incorrect texture mapping due to the improper interpolation or merging of texture coordinates by collapsing across texture chart boundaries. This kind of approach limits the efficiency of the simplification algorithm used, since all texture chart boundary edges can't be decimated and in cases where the texture charts are largely fragmented this will severely limit the impact the decimation quality.

Both Garland et al., Sander et al. and Chen et al. [77]–[79] independently proposed the addition of texture coordinate and color attribute as part of the quadric error metric for edge collapse, allowing the simplification algorithm some measure of control in preserving the texture of the model. These approaches trade-off mesh decimation quality to maintain good texture quality by disallowing edge collapse that might distort the texture too much even if the edge provides the least geometric deviation after a collapse.

CHAPTER 3: PARALLEL POINT CLOUD COMPRESSION USING TRUNCATED OCTREE

This chapter explores the novel concept of using a truncated octree to improve the compression rate of point cloud compression using a spatial partitioning scheme.

Existing methods usually exploit the spatial sparseness of point clouds using hierarchical tree data structures like an octree for spatial encoding.

However, such methods can be inefficient when very deep octrees are applied to sparse point cloud data to maintain a low level of geometric error during compression.

We propose a novel octree structure called truncated octree which can improve the compression ratio by representing the deep octree with a set of shallow sub-octrees that can save storage without losing the original structure.

We also propose a variable-length addressing scheme, to adaptively choose the length of an octree's node address based on the truncation level---shorter (resp. longer) address when octree is truncated near the leaf (resp. root) which leads to further compression.

This method can achieve a 40% to 90% compression ratio on our tested models for point clouds of different spatial distributions. For extremely sparse point clouds, the method achieves approximately 7 times higher compression ratio than previous methods. Moreover, the method is designed to run in parallel for octree construction, encoding, and decoding.

3.1 Introduction

As depth sensing and reality capture technology become increasingly advanced and achieve high-resolution output, the resulting point cloud file size captured by such scanners becomes exceedingly large.

Unlike structured point clouds where hardware and temporal knowledge can be exploited to achieve higher levels of compression [80], algorithms for unstructured point clouds can only rely on the low-level spatial relationship as a means of compression.

Many algorithms explored this spatial relationship as the basis of compression, usually using hierarchical structures or octrees to represent the spatial relationship.

While these methods work well on point clouds that are dense, their compression efficiency decreases significantly on point clouds that are sparse or required deep octrees to minimize quantization error. One must not confuse the sparseness of a point cloud distribution with that of guaranteed small file size. Large scale scans of entire city blocks can easily cover square kilometers of an area that even when sparsely sub-sampled, can result in a large point cloud file.

In this paper, we present a novel unstructured point cloud encoding algorithm, which can flexibly handle point clouds of any sparsity while still accounting for both compression rate and quantization quality.

The method is designed to run in parallel on modern multicore CPUs to alleviate the encoding and decoding time for large point clouds leading to 3 times speedup during the encoding phase, and 2 times speedup in the decoding phase.

Like previous works, we also use an octree as the basis of our encoding scheme. The octree is not only beneficial for downstream techniques such as level-of-detail point cloud visualization, surface construction, clustering, normal estimation, etc. but also leads to an intuitive compression algorithm where the quantization error can be controlled by the depth of the octree.

Different from previous works, we use a parallelized bottom-up octree construction instead of the usual top-bottom construction. By doing this, we can avoid unnecessary

traversals from the root to leaf node by early termination after encountering an existing parent node during construction.

Furthermore, due to the upward propagation of nodes creation from leaf nodes, it is relatively easier to parallelize and synchronize in a multithreaded implementation.

We introduce the novel concept of truncated octree that allows us to reduce the file size significantly compared to a full octree. We observed that sparse point data combined with deep octrees (for lower quantization error) leads to long chains of hierarchies in the octree structure. This negatively impacts the benefits brought about by the octree for compression, since long chains of the octree hierarchy are dedicated to improving the quantization error of lone points.

With a truncated octree, the leaf nodes are not referenced with respect to the root, but from nodes that could be in any chosen truncation level of the octree. In effect, this results in the creation of multiple smaller sub-octrees, and each node in the truncation level becomes the root of its corresponding sub-octree. The truncation level is chosen automatically during encoding to minimize the file size. This does not lead to information loss, and the original octree can be reconstructed easily via bottom-up octree construction.

Furthermore, we introduce a variable-length addressing scheme to address each of the sub-octree. We discovered that since the truncation level is dynamically chosen, the addressing space required should also be dynamic to optimally match the requirements of the currently selected truncation level.

For example, if the truncation level is close to the root level, then we need significantly lesser bytes to address the nodes of the sub-octrees and can thereby increase the compression ratio.

We demonstrate and evaluate our method against the state-of-the-art based on the compression ratio and encoding/decoding speed achieved on several models of varying complexity and shapes.

3.2 Proposed Method

Our method has two inputs: a point cloud model $\mathbf{M}$ with a set of 3D vertices, and a user-specified maximum allowed quantization error threshold E_M , which will be used to determine the required depth of the octree D using Equation 1 below.

Let B_M be the diagonal length of the bounding box of $\mathbf{M}$. Then, the maximum geometric error E_M for octree depth D is defined by the following:

$$E_M(D) = B_M \times 0.5^D$$

The output is an encoded binary file that contains the encoded octree representation of the point cloud. This file can be decoded once again to retrieve the quantized point cloud as well as its octree structure which can be reused for further processing. Additional file compression algorithms such as ZIP compression can be applied to the binary file if necessary.

Our method is composed of three main components: bottom-up octree construction, octree truncation with variable length addressing, and encoding/decoding. Bottom-up octree construction allows us to construct the octree representation of the point cloud in parallel. Truncating the octree dynamically allows us to discard the unnecessary hierarchical structure in a lossless manner from the upper levels of the octree and generate a set of sub-octrees such that the total memory usage is lesser than the original octree. Furthermore, variable size addressing adaptively changes the bit length required to represent the octree node address to provide an even higher compression rate.

3.2.1 Bottom-Up Octree Construction

Using a hierarchical structure allows us to distribute the quantization error of the points across each level of the hierarchy, where each level contributes a little bit more

information towards reconstructing the final point coordinates. Because the information required by each level is small, well structured, and shared by all its children, we can efficiently encode and compress them.

3.2.1.1 Octree Representation

Octree [81] is a spatial partitioning data structure popular for its simplicity and uniformly structured hierarchy. Using an octree to encode the underlying geometric information of the point cloud provides several benefits. Firstly, the octree nodes can be subdivided only in the presence of points within their bounds, leading to a sparse representation. Secondly, the octree, unlike other hierarchical structures such as bounding volume hierarchy or BSP trees, uses a regular subdivision hierarchy which the encoder and decoder can exploit to more efficiently compress the point cloud without having to store extra partition information. Since there is no need to store explicit floating-point positions, all the encoding operations are done using integer representation which is more efficient and minimizes possible floating-point rounding errors.

We store only the bounding box in floating-point representation and use it to reconstruct the point cloud to scale. As shown in Equation 1, E_M is the upper bound of geometric error that is introduced by using integer addressing at the specific depth of the octree.

The octree nodes are addressed by the Morton code or a z-order curve [82], [83] which is a mapping from n -dimensional space ($n=3$ in our case) onto a 1D space. Each node of the octree contains a single atomic byte, which represents its child occupancy code [58] of this node. Each bit in this code denotes the existence (or absence) of each of the 8 possible children of the octree node. For each level of the octree, we have a hashmap where the keys represent the node address and the values represent the child occupancy code. The full octree itself is then represented as a list of D hashmaps.

The leaf nodes' positional information can be completely derived from its parent's child occupancy code; hence we don't need to explicitly store leaf nodes in memory. Note that this factor by itself can result in a significant memory saving of roughly 8 times, which is significant in very deep octrees. With this representation, our octree can be considered a sparse octree since it does not explicitly track empty nodes.

3.2.1.2 Octree Construction

Figure 3.1: Example of early termination of upward propagation in a binary tree. Leaf nodes A, B, C, and D are added in order. We can see that B and D had early terminated when they encountered the parent nodes created by A and C respectively.

Usually, the construction of the octree is done from top-down, where any insertion of a new node first starts from the root node followed by recursive traversal towards the bottom. While simple to implement, such a method is inherently serial and inefficient as there is a high number of redundant node traversals from the root to leaf for every insertion.

To improving encoding/decoding performance, we instead employ the bottom-up octree construction [84], [85]. Each node insertion starts at the leaf level and recursively creates or updates the immediate parent node until the traversal reaches the root node. With this approach, we can early terminate the upward traversal from the leaf level, once we reach an existing parent node, see Figure 3.1 for an example.

In a parallel implementation, we need only two synchronization points---first when creating a new parent node, and second, when updating the parent node's child occupancy code. In detail, we require a mutex for each octree level when adding a new node to that level. Since the child status of the parent node is a single atomic byte (1

bit for each of the 8 children), we can update the child status using atomic **fetch_or** instruction with minimum locking penalty.

With the synchronization in place, we can insert each point into the octree in parallel. Due to the synchronization, it might be preferable to insert points that are not spatially near to each other to avoid excessive contention on the same parent, since nearby points are more likely to share the same parent at a higher level. This can be simply achieved by sorting the points randomly before the octree construction.

Table 3.1: Comparison of the number of traversals performed when generating the octree using different construction methods.

Model	Bottom-Up	Top-Down	Savings (%)
Bunny	33,111	40,048	17.32
Gargoyle	7,656,447	13,811,360	44.56
Dragon	28,173,175	57,753,600	51.22
Lucy	93,597,517	224,445,952	58.30
Car	14,934,133	30,258,960	50.65
Truck	30,497,269	42,769,360	52.07
Fountain	104,576,365	719,850,256	85.47
SG28 Station	483,989,488	4,139,535,168	88.31

3.2.2 Octree Truncation

Figure 3.2: A pathological case for existing octree-based compression methods illustrated using 2D points and a quadtree: a sparse point cloud requiring a low quantization error lead to a deep quadtree.

Previous methods [57], [58] retain the entire octree in memory as well during storage and generally perform well on dense and regular point clouds. However, their performance deteriorates when compressing point cloud models that are sparse but require deep octrees to minimize the quantization error.

Due to sparseness in the point distribution relative to the depth of the octree, the octree structure tends to appear "long-legged" where long chains of nodes are created to reduce the quantization error on lone child nodes, which extend all the way to the leaf level. This class of models is usually created when subsampling a dense point cloud to a lower sampling rate.

However, any point cloud represented with a sufficiently deep octree will face this problem, where the leaf cell size becomes much smaller than the minimum distance between two points. When long chains arise in the octree, the memory required for each of the child occupancy codes throughout the chain increases leading to diminishing returns.

Hence, we propose the following truncated octree to alleviate this problem. Rather than storing/encoding the octree nodes with reference to the root node, we instead truncate all the nodes from the root to a determined octree level for a concise representation. This creates a set of sub-octrees with the remaining nodes addressed with reference to each of their corresponding root nodes.

Figure 3.3: Example of truncated quadtree for the dense point cloud.

For a dense point cloud, the optimal truncation level will usually be at the root level of the octree, meaning no truncation, because on a dense point cloud, encoding the whole octree in terms of child occupancy codes is already efficient.

Figure 3.4: Example of truncated quadtree for the sparse point cloud.

On the other hand, for extremely sparse point clouds the optimal truncation level is $D-1$, which is one level above the leaf nodes, it is more efficient to store the individual points rather than the entire octree. This is because we need a very deep octree for sparse point clouds which leads to a larger file size if we encode all the nodes. At this truncation level, the octree address almost resembles a regular sparse voxel grid.

Figure 3.5: Example of truncated quadtree for the clustered point cloud.

For point clouds that have several isolated clusters of dense points, the optimal truncation level will likely fall in the middle range, leading to a set of sub-octrees equal to the number of clusters.

As shown with the above examples, we demonstrated that regardless of the point cloud distribution pattern (dense, sparse or cluster), but our method can select a truncation level that achieves optimal compression rate. Our method is also able to automatically select the optimal truncation level. Since the full octree information in memory, it is

straightforward to compute the best octree truncation level to achieve the smallest file size by scanning over all the nodes in each octree level and computing their size requirements. Then, we can store the addresses (Morton codes) of the nodes in the truncation level, and the child occupancy codes of all their children. We show the benefits brought about by truncation in the Experiments section.

3.2.2.1 Variable Length Addressing Scheme

Table 3.2: List of Octree level and optimal address code length.

Bit Length	Bit Per Axis	Range Per Axis	Max Octree Level
8	2	0-3	3
16	5	0-31	6
24*	7	0-127	8
32	10	0-1023	11
40	13	0-8191	14
48*	15	0-32767	16
56	18	0-262143	19
64	21	0-2097151	22
128	42	0-4398046511103	43

- *Due to the bit flag needed to support offset jump, there is an extra bit used for address code with length in multiples of 3.*

After careful analysis, we determined that using a static length for the node address is inefficient for truncated octrees. Based on the truncation level selected, we can optimally determine an efficient length of the address code to use in encoding the sub-root node addresses. The range of addresses represented by the bit length is shown in Table 3.2.

Clearly, when we truncate the octree, the depth of each sub-octree is much lesser than the original octree, thereby requiring less child occupancy codes to be fully represented. Conversely, as the truncation level approaches the leaf level, the number of sub-octrees also increases.

A large number of sub-root nodes that need addressing burden the compression efficiency. Hence, we choose this length dynamically based on the truncation level. To have better byte alignment, the variable bit length of the address code is chosen in multiples of 8. By optimally selecting the length of the address, each of the sub-root nodes will only require the minimum bits to correctly address their location on the truncation level. Beyond the obvious benefit of a higher compression ratio, optimal address length also helps in reducing the encoding/decoding computing requirements.

3.2.2.2 Offset Jump Address

Table 3.3: List of encoded size obtain using different address code and jump offset length. The octree truncation level of 6 is chosen to fit within the 16-bit full address.

Model	Bit Length		Encoded Size (bytes)
	Full Address	Jump Offset	
Lucy	64	32	67856
	32	16	37632
	16	8	<u>21802</u>
	16	NA	34882
Gargoyle	64	32	81336
	32	16	47172
	16	8	<u>26411</u>
	16	NA	42784
Dragon	64	32	81336
	32	16	23144
	16	8	<u>13839</u>
	16	NA	21450

Lastly, we observe that several nodes in the octree are neighbors. We can hence refer to certain nodes with their original address and their neighbors with a shorter offset jump instead of the full-length address. Using the previously computed full address and adding the offset jump, we can obtain the full address of the next node.

In our implementation, the jump address is half the length of the full address, in order to attain a good balance between space-saving and the range of the offset jump.

Instead of the unsigned integer used by the full address, the jump offset is stored as a signed integer to allow for the negative jump in any direction. Due to the shorter range of the jump offset compared to the full addressable ranges, there will be cases where the next node address exceeds the range of the offset jump, during which we will revert to using the full address and start over with the jump address once again.

To determine whether the address is a full-length address or a shorter offset jump during encoding/decoding, we use a bit flag encoded into the first bit of the address itself.

3.2.3 Encoding

The encoding algorithm will encode the truncated octree into a binary blob which can be written directly to a file or further compress using any file compression algorithm.

3.2.3.1 Header Information

To support the decoding algorithm, additional attribute needs to be written to the encoded file's header:

The bounding box information of **M** .
Depth of the octree **D** .
Truncated depth of the octree **D_t** .
Size of the encoded body in bytes **B** .

Immediately following the header information will be the encoded binary blob.

3.2.3.2 Body Information

We have two simple operations in creating the encoded body: writing the sub-root node address addresses of the nodes in the truncation level and traversing each sub-octree to write the children nodes' child occupancy codes.

Encode (O, D_t):

Write the header information:

$P \leftarrow (0,0,0)$ // Previous sub-root address

For all sub-root node address S in level D_t do:

 If $|P - S|$ is within the range of the offset jump then:

 Write offset jump address

 Else:

 Write full sub-root address

$P \leftarrow$ full address of S

For all Node N in depth first traversal from S do:

 Write N 's child occupancy code

The parallelism of this process can be achieved by parallelizing the iteration of the for all sub-root node transversal. Since each sub-octree is completely independent of each other, we can perform the depth-first traversal in parallel for each of the sub-octrees. An additional step to aggregate the various results of sub-octree traversal into the final binary blob needs to be performed.

At the end of the encoding, the resulting encoded binary blob can be written to file directly or further compressed using a file compression algorithm such as Zip compression [86].

3.2.4 Decoding

The output of the decoding algorithm is the complete octree representation in memory. Even if the encoded octree was truncated, the decoded octree will still be the full untruncated octree so that it can be used for downstream operations.

Since the header information is written as a raw dump, we can easily load and initialize the currently empty octree. The decoding of the encoded body is slightly more involved than the encoding algorithm since we need to track the status of the child node expansion.

Let **P** be the previous sub-root node address, initialized to the zero coordinate (0,0,0). Let **A** be the current sub-root node address and **N** be the current sub-root node. Let **S** be the stack used to track the decoding process and **O** to be the output full octree. Let **T** be the node at the top of **S**.

Decoding (O):

Read the header information:

P <- (0,0,0) // Previous sub-root address

For all byte **B** in file do:

isFull <- first bit of **B** // Check whether **B** is a full address or an offset jump address

If **isFull** == **True** then:

A <- **B**

Else:

A <- **P** + **B**

P <- **A**

N <- **A**'s node

Recursively add the **N** to **O** // Bottom-up construction to root

C <- **N**'s child

Push **C** onto **S**

While **S** is non-empty do:

T <- **S**'s top node

C <- next child of **T**

Create the node for **C** in **O**

Remove **C** from **T**

If **T** has no child left then:

Pop it from **S**

If **C** is not leaf then:

Push **C** onto **S**

The decoding algorithm reads each byte from the file and decodes the address of the sub-root nodes based on whether it is a full address or an offset jump address. We first reconstruct the upper truncated portion of the octree, by recursively upward propagating nodes toward the octree's root. Then, we perform the depth-first traversal of each sub-octree towards the leaf nodes.

To enable parallelism for the decoding process, we need some extra storage overhead in the form of each sub-octree's encoded size, so that the decoder knows when to terminate the processing of a sub-octree.

At the end of the decoding, we will have the complete octree in memory. To regenerate the point cloud from the octree, we just need to iterate through each of the leaf nodes and compute the centroid of the node as the point's position. Alternatively, a probabilistic approach can also be used such as [54], [61], to predict the distribution and position of the points within the octree cell.

3.3 Experiment

Our method is implemented in C++ and evaluated on a computer with an i7-6700 3.4GHz, 8 core CPU. We support CPU parallelism using the Intel Threading Building Blocks (TBB) library [87].

Some of our test models, namely Bunny, Dragon, Lucy, are originally triangular meshes; we remove all attributes except the vertex positions. Other models such as Car, Truck, Fountain, SG28 Station from [88], [89] are real-world scanned data. The baseline unstructured point cloud format used in the comparison is the binary version of the Polygon file format (.PLY).

3.3.1 Quantization Error

Figure 3.6: Quantization error of the gargoyle at octree level 12.
The root mean error is at 0.0033 meters.

Firstly, we examine the quality of the decoded point cloud. Given that our method quantizes the point cloud to the size of the smallest octree cell, the maximum error guaranteed will be half of the diagonal length of the smallest cell. In this respect, our method's vertex position quantization error is equivalent to that of those proposed by Peng et al. [56]. This implies that the selection of proper octree depth has a large impact on the geometric error of the decoded point cloud.

In our experiments, beyond the octree level 12, there is virtually no "visible" difference between the original and compressed point clouds for our test models. We show several results in Figure 3.6 visualizing the quantization errors for the gargoyle model and octree depths. The error is evenly distributed throughout the points.

3.3.2 Compression Results

Table 3.4: File size compression rate compared with other methods, underlined are the best-compressed size.

Model	Points	Size (KB)	Encoded Size (KB)		
			Ours	[51]	[57]
Bunny	2,503	30	17.10	<u>13</u>	27
Gargoyle	833,104	9,764	<u>3,020</u>	3,868	3,583
Dragon	3,608,622	42,289	<u>2,196</u>	16,646	2,444
Lucy	11,921,036	139,700	<u>3,122</u>	55,383	3,483
Car	1,763,091	20,662	<u>2,633</u>	8,223	2,795
Truck	2,463,104	28,865	<u>8,721</u>	11,410	10,714
Fountain	28,467,870	333,6008	<u>39,342</u>	264,117	43,252
SG28 Station	113,898,368	1,334,747	<u>146,382</u>	1,052,707	176,382

Next, we examine the file size compression obtained by our method compared with others. Table 3.4 shows the results obtained with several different models.

The heavily subsampled bunny that represents an extremely sparse point cloud, gargoyle with average density and point count, and Lucy model with both high point density and point count. The car, truck, fountain, and S28 station models are real-world raw laser scan data.

We can notice a wide variance in compression rate ranging from 40% to 90% as reported in Table 3.4 This is largely due to the encoding algorithm performing much better on a dense point cloud, where both truncated octrees encoding, and usage of the offset jump are optimal. While the sparser model with nodes that are far apart, the sharing of parent nodes is minimal, and usage of the full address code instead of offset jump is also much increased. We wish to highlight the flexibility of the method to handle compression of this wide gamut of point cloud density in comparison to a method like [57] which only works well on a regularly sampled point cloud.

The results in Table 3.4, demonstrate the improvement when compressing sparse point cloud compared to [51], [57]. To reflect a sparser point cloud density profile, the experimental models are uniformly subsampled by removing 5% of the original points.

Our method improves the compression rate compared to Merry et al. [57] since they must encode the full octree. Improvements are more pronounced as the depth of the octree level increases or as the point cloud density becomes sparser, as demonstrated by the heavily subsampled bunny model.

Even on denser models, our method has some slight improvement. As mentioned by Lindstrom [51], their method performs poorly when presented with unstructured data that does not exhibit spatial correlation for nearby data points. Notably, our method outperforms both compared methods on all tests.

3.3.3 Performance

Table 3.5: Timing performance (in seconds) for each main step of the algorithm for both parallel and serial implementation. Comparison with the encoding time of [51], [57].

Model	Our Method (Parallel/Serial)			[51]	[57]
	Build Octree	Encode	Decode	Encode	
Gargoyle	0.96/4.06	0.50/1.66	0.90/1.77	1.44	5.38
Dragon	4.91/20.62	1.06/3.52	1.52/3.00	6.15	22.84
Lucy	17.99/63.57	2.82/9.31	3.07/6.07	19.95	80.43
Car	2.886/7.12	0.89/30.13	7.40/14.59	3.00	14.22
Truck	3.816/13.02	1.24/4.10	2.19/4.31	4.71	18.99
Fountain	35.144.80	9.13/30.12	10.76/21.21	16.67	168.41
S28 Station	145.34/510.43	26.25/86.64	47.92/94.40	62.82	660.12

We now discuss the computation and parallel processing capability of the method. By using the bottom-up construction of the octree, we are able to gain both the ability to construct the octree in parallel and also up to a 90% reduction in the unnecessary

nodes traversal as compared to standard top-down octree construction, as reported in Table 3.1.

Due to the simplicity of the encoding and decoding algorithm, our method has a much lower computational complexity compared to the method like [53], [56] while still being able to obtain a good tradeoff between the compression rate and quality. In addition, our method is fully parallelizable and can exploit all available CPU cores which further improve the run-time speed. We report the timing statistics of each step of our method in Table 3.5

Figure 3.7: Effect of truncation the octree at different levels on encoded file size compared to no truncation.

Octree truncation significantly helps in reducing the file size as shown in Figure 3.7, where we explore the relationship between the encoding size against the octree truncation level for dense, sparse and clustered point clouds. Here we explore three different point cloud density profiles, the original dense point cloud (Fountain and S28 Station), a sparse decimated version (Bunny, Gargoyle, Dragon, Lucy), and clusters of isolated points (Car and Truck).

Sparse and clustered point clouds benefit from truncation when it comes to filing compression. While dense point cloud does not have any gain from the truncation, our method is able to fallback to use no truncation and retain the high compression rate from spatial hierarchy-based compression. We wish to highlight that due to the adaptive nature of our method, we can achieve the optimal truncation level for any kind of point cloud profile without any manual intervention.

Variable-length addressing of the nodes after truncation also reduces the file size significantly. Please refer to Table 3.3 that shows approximately 3 times improvement in space-saving can be obtained from using the optimal length address code during encoding. With the use of the jump offset address, a further improvement of around 1.5 times reduction in file size was obtained.

The following Figures 3.8 to 3.11 are showing the quantization errors of the various dataset at different octree depth of 10, 12 and 16. These depths correspond to the maximum quantization error of 0.0009765%, 0.0002441% and 0.00001525% of the models bounding box size.

Figure 3.8: Visualization of point cloud quantization error of the Dragon dataset. Rows: Octree depth 10, 12, 16.

Figure 3.9: Visualization of point cloud quantization error of the Lucy dataset. Rows: Octree depth 10, 12, 16.

Figure 3.10: Visualization of point cloud quantization error of the Truck dataset. Rows: Octree depth 10, 12, 16.

Figure 3.11: Visualization of point cloud quantization error of the Fountain dataset. Rows: Octree depth 10, 12, 16.

3.4 Summary

We have presented a novel parallel point cloud compression algorithm for point cloud of varying density profiles. Our compression method is based on a truncated octree and variable-length addressing scheme to obtain a higher level of compression rate. Parallelism is also achieved by our method for the main portions of the algorithm during the octree construction, encoding, and decoding steps. We have shown that our method can improve the compression rate upon both sparse and dense point cloud over similar related works. Specifically, our method is able to dynamically select the optimal truncation level for any point cloud distribution pattern (sparse, dense or cluster).

Currently, we only consider the position information of the point cloud. In practical usages, a complete point cloud usually requires the color and normal information for either visualization [90] or surface reconstruction [3]. Existing works [52], [58], [91] already tackle this issue of compressing point cloud's color and normal information. As future work, we will investigate ways to integrate those methods into our encoding scheme and explore more ways to encode any extra attributes of the point cloud into the compression scheme.

Another direction to be explored is the suitability of the proposed encoding as a streamable point cloud format. By changing the octree encoding from depth-first to breadth-first traversal, it is possible to obtain point clouds of different levels of details progressively while decoding the rest of the file.

CHAPTER 4: GPU BASED MULTIPLE CHOICE SCHEME MESH SIMPLIFICATION

This chapter will first introduce the “GPU based multiple choice scheme mesh simplification” method. Then we look more in-depth at the three main components of the work, namely the “QEM-based Edge Collapse”, “Multiple-Choice Decimation” and “Parallel Collapse based on Super Independent Vertices”. Finally, we proceed to explain our algorithm in full detail.

4.1 Introduction

Reality capture technologies such as laser scanner and photogrammetry are becoming more democratize day-by-day, as such we are currently undergoing an influx of high-resolution photo-realistic 3D models that contain far greater texture and geometric details and resolution than ever before. The massive data file size required to store this amount of quality can cause difficulty for both the transfer and manipulation of the model. While highly valued in the field of cultural heritage preservation, reverse engineering, architectural design and quality control where such precise representation of the model is required, they are often too large for other interactive uses such as real-time rendering and virtual reality.

Mesh simplification is a technique used to reduce the complexity of the model while trying to retain as much quality as possible at the simplification target. Many mesh simplification methods compute a global optimal for performing the simplification on the model, which restricts the methods uses of parallelism to accelerate computation.

This dissertation is focused on the simplification of large photo-realistic textured models reconstructed through reality capture technologies. We will investigate ways to improve

the performance of existing mesh simplification algorithms in both aspects of speed and memory while also attempting to retain as much texture information as possible after simplification.

4.2 Method Overview

The input to our method is a triangular mesh $T = (V, F)$. Where $V \in \mathbb{R}^{3n}$ is the vertex position. $F \in \mathbb{R}^{3m}$ is the triangle indices, i.e., each triangle defines a triple (i, j, k) indicating the indices of vertices in V . The output is a simplified triangular mesh $T' = (V', F')$ with lesser triangles than previously, the target number of triangles was specified by the user. We do not require the mesh to be manifold or watertight. After these data were read from the disk to CPU RAM, we will allocate the required memory on GPU and start streaming the data unto the GPU memory. After that, all computation is done completely on the GPU.

Our method was composed of three main components: QEM-based edge collapse, probabilistic multiple-choice decimation, and GPU parallelism. QEM was adopted as the edge collapse error cost metric. Multiple-choice decimation was used to select the “best” edge in a probabilistic point of view. The GPU was fully exploited for parallel acceleration. We first give a brief introduction and analysis of these three components and their usage in existing methods in the following sections.

4.3 QEM-based Edge Collapse

QEM [5] is a popular error metric used to measure the error incurred for an edge collapse based on vertex-plane distance. In particular, QEM is well known for the high quality of mesh after simplification. An implementation based on QEM simplification, QSlim is shown to be very successful in both quality and speed.

For a vertex v_i , each of its adjacent triangles defines a plane $q = (a, b, c, d)$, where (a, b, c) is the normalized normal of the plane and d is the distance to the origin. Thus the

squared distance from a point $\mathbf{v}$ to the plane can be calculated as $\mathbf{D}^2(\mathbf{v}) = (\mathbf{v}, \mathbf{1}) (\mathbf{q}^T \mathbf{q}) (\mathbf{v}, \mathbf{1})^T = (\mathbf{v}, \mathbf{1}) \mathbf{Q} (\mathbf{v}, \mathbf{1})^T$. For a vertex $\mathbf{v}_i$, its quadric term is the sum of those of its adjacent triangles: $\mathbf{Q}(\mathbf{v}_i) = \sum_{k \in \mathbf{N}_i} \mathbf{Q}_k$, where $\mathbf{N}_i$ is the indices of triangles adjacent to $\mathbf{v}_i$.

Figure 4.1: A demonstration of edge collapse. Edge $e(v_i, v_j)$ (red) is collapsed into one vertex v_c .

For an edge collapse, we use the QEM to approximate the geometric error introduced by performing the collapse. For an edge $\mathbf{e} = (\mathbf{v}_i, \mathbf{v}_j)$, the resulting vertex after collapsed is denoted as $\mathbf{v}_c$, as shown in Figure 4.1. The error is defined as $\mathbf{D}^2(\mathbf{v}_c) = (\mathbf{v}_c, \mathbf{1}) (\mathbf{Q}(\mathbf{v}_i) + \mathbf{Q}(\mathbf{v}_j)) (\mathbf{v}_c, \mathbf{1})^T$. The position of $\mathbf{v}_c$ is chosen as the one minimizing the error of $\mathbf{D}^2(\mathbf{v}_c)$. Then $\mathbf{Q}(\mathbf{v}_c)$ for the new vertex is updated as $\mathbf{Q}(\mathbf{v}_c) = \mathbf{Q}(\mathbf{v}_i) + \mathbf{Q}(\mathbf{v}_j)$.

In QSlim, a greedy strategy is used to select global minimum cost edges for collapse. All the edges were ranked according to their incurred error $\mathbf{D}^2$. Each time the edge with minimum cost is chosen for edge collapse and the edge costs for adjacent edges are updated and iterate again. A heap data structure is used to hold all the edges and allow for efficient selection of the next best edge.

The simplified mesh is usually of superior quality due to the QEM error metric used. Many derivatives are based on the QSlim algorithm [92]. However, two main problems exist for its greedy algorithm: slow computation and high memory consumption. Since each time we can only collapse one edge and the subsequent collapse depends on the previous ones, the algorithm processing is then highly serial and cannot be parallelized. In addition, we need to compute the cost for all edges at the start and construct a heap data structure to store it all. This will inevitably require a sizable portion of memory and maintaining this structure require significant computation overheads. The requirement of computing all the edges cost at the start also means that the memory and

computation overhead is constant for all decimation count. For the decimation of only a few triangles, this overhead becomes disproportional large.

4.4 Multiple-Choice Decimation (MCD)

Figure 4.2: An illustration of conflicting edges (a) and independent edges (b).

Multiple-choice decimation [73] was proposed to tackle the problem faced by the original QSLim method. It is based on a novel probabilistic analysis. Stemming from the observation, that if the mesh is targeted to be highly simplified, its best edges selection can be more relaxed when compared to the greedy approach. Even if our current choice is not the current best choice, it is considered acceptable if it will eventually be collapsed in the future iteration. For each step, the multiple-choice algorithm selects multiple edges, among which the one with the minimum edge cost is chosen to be collapsed. The more edges we choose as candidates, the less likely we will choose a “wrong edge” which shouldn’t be collapsed. For example, assuming the mesh is targeted to be

simplified to 5% of the original complexity. Each time we pick 8 possible edges randomly and pick the one with the minimum edge cost. The probability for a wrong decision is: $\left(\frac{5}{100}\right)^8 \approx 10^{-11}$

By wrong decision, we mean all the 8 picked edges do not belong to the 95% majority of edges that are supposed to be removed eventually.

An interesting attribute of MCD is by changing p defined as the number of random edge candidates selected per iteration, we can adjust the trade-off between computation speed and mesh quality. For example, when $p = \text{number of edges}$, the algorithm performs exactly like the greedy algorithm (i.e. QSlm) and picks the global minimum cost edge albeit without the supporting optimization of a heap structure. Conversely, when $p = 1$, the algorithm degrades into pure random selection decimation that while extremely fast will generally generate a resulting mesh with unacceptable quality. Through our experiments, we agreed with the suggestion by Wu and Kobbelt [73] that the $p = 8$ can produce a good tradeoff of both fast and good quality decimation model.

Another main advantage of the multiple-choice method is that there is no need to store a full heap data structure as QSlm. This significantly reduces both memory overhead and implementation complexity. Additionally, the edge cost is only computed on demand when choosing a specific candidate edge. A lot of the unnecessary computation is reduced if the target decimation level is low since there is no need to compute every single edge cost before a selection can be done like QSlm. However, the original MCD is based on CPU. It can still be slow for very large meshes that requires a high level of decimation.

4.5 Parallel Collapse based on Super Independent Vertices

GPU based methods [69] were proposed to perform collapse on many edges at once. The main difficulty for such parallel collapse is the problem of thread racing: edges chose by multiple threads may have an overlapped influence area (Figure 4.1). In [69],

a super independent vertices mechanism was used to tackle this issue. Super independent vertices are vertices that are at least 3 rings away from each other. Thus, each super independent vertex defines an isolated region, in which only one edge can collapse without affecting the QEM of other edges. This guarantees the independence of the selected edges.

To identify super independent vertices, multiple threads were used to scan through the whole mesh's vertices and try to lock the surroundings (3-rings) of a certain vertex. If such a lock for a certain vertex succeeds, the vertex is defined as a super independent vertex. For all the super independent vertices, their surrounding regions are disjoint. For each super independent vertex, the adjacent edge with minimum cost is identified as a candidate for collapse. Then all these edges are sorted according to the cost and a subset of these best-scored edges is collapsed together in parallel.

The advantage of the GPU implementation in [69] is clearly the improvement in speed. It can obtain a speedup of up to 5 times faster than QSlim. However, the quality of the simplified mesh is often of lower quality. Unlike QSlim, we cannot select the single edges with global minimum cost, since we must perform the simultaneous multiple collapses across the various super independent region even if some candidate edge cost is suboptimal. In [69] the author proposes to sort all the found edges and choose only a subset of these edges with smaller edge costs for collapsing. However, this will incur more computation as we will then need to do the extra sorting. In addition, it is hard to specify the size of the subset. If the subset is too large, the quality will decrease and otherwise, the parallelism is decreased since the number of collapsed edges each time is reduced.

4.6 Proposed Method

After analyzing the previous algorithms, we propose to combine multiple-choice decimation scheme and GPU-based parallel simplification. Our method will utilize the GPU to do parallel simplification, i.e. we collapse multiple edges at the same time. However, different from [69] which chooses a single edge in an area defined by super

independent vertex, we adopt the multiple-choice selection method to improve the decimation quality in a probabilistic sense.

4.6.1 Data Initialization

We use a similar data structure used in [69] to assist in the simplification. We build a Vertex-Triangle adjacency list. For each vertex, we store references to its adjacent triangles. After the initialization of additional parameters like vertex quadrics, we can proceed to decimation iteration. For each iteration, we will perform a parallel collapse. We first use the GPU to randomly select a set of candidate edges. Then, independent edges are identified by locking the edge surroundings. All the independent edges with the lowest QEM error are then collapsed together. After that, we update the mesh topology. We iterate the process until the number of the remaining triangles reaches the target.

4.6.2 Candidate Edges Identification

We divide our GPU threads into subgroups (e.g. $p = 8$ threads per group), each of which is responsible for choosing one candidate edge. Within each subgroup, all the threads are used to select multiple candidate edges randomly and compute the cost for each edge on demand. The edge with the minimum cost is then chosen as a candidate.

To be specific, each thread is used to find an edge randomly. Since our data structure has no edge information, we do this through the Vertex-Triangle adjacency list. Each thread first randomly selects a vertex, then an edge adjacent to this vertex is again randomly selected and its corresponding QEM edge cost is evaluated. Among the edges selected by the subgroup, we pick the one with the minimum edge cost as a candidate for further edge collapse.

4.6.3 Independent Edges Identification

We then try to identify independent edges. This is done in a similar way in [69] to identify super independent vertices. For each candidate edge, we try to lock its surroundings by marking the one ring triangles adjacent to it. An additional buffer was used for the marking process. Each thread tries to write into the buffer for its own candidate edge. After the marking process, we check the marker for each candidate edge's adjacent triangle. If the marker is consistent with the candidate edge, the edge is independent of the others and collapsed. Otherwise, this edge is dropped without collapse.

4.6.4 Topology Update

After the edge collapse, the topology of the mesh is updated. The triangles collapsed are marked as removed. The triangle indices are updated accordingly. The Vertex-Triangle adjacency list is updated according to [69].

4.6.5 Iteration Check

Then we check the remaining number of triangles. We iterate again if it does not reach the user-specified target count.

4.6.6 Putting it all together

The algorithm is summarized as follows:

- 1) Stream (**V**, **F**) into the GPU and initialize quadrics for vertices.
- 2) Build the Vertex-Triangle adjacency list.
- 3) For each thread, randomly select a vertex and then an edge adjacent to vertex.
- 4) For each selected edge. Compute the edge costs.

- 5) For each thread subgroup, select the edge with minimum cost as a candidate for collapse.
- 6) For each candidate edge, try to lock its surroundings by marking its one-ring adjacent triangles. We adopt a similar strategy used in [69] to lock the surroundings: Each thread just tries to write into the marking buffer for its surrounding vertices without considering conflicts. After that, each thread just checks whether the marks of its surrounding vertices are consistent with itself. If two threads are conflicting and trying to mark the same vertices, the marks cannot be consistent for both threads at the same time.
- 7) For all the edges whose surroundings are consistently locked, do the edge collapse.
- 8) Update the mesh topology.
- 9) If the number of the remaining triangles is larger than the target, go to step 3.

4.7 Experiment

This section will explain the program I implemented that includes the algorithms discussed in this report and discuss the experimental results.

4.7.1 Experimental Results

Our method is implemented in C++ and runs on a computer with CPU i7-6700 3.4GHz and GPU GTX970. The GPU algorithm is programmed with CUDA Toolkit 8. The complete source code of the program is hosted on GitHub: <https://github.com/Naimin/Zephyr>.

We compare our method with QSlim, MCD and super independent vertices [69]. For simplification quality, we use the Hausdorff distance as a measure of geometric deviation used in [69]. The numerical results are summarized in Table 4.1. When comparing with QSlim, we use the QSlim implemented by MeshLab 2016 [93]. Our method is shown to run faster but has a slight loss in mesh quality compared to QSlim. Compared to the CPU-based MCD, our method is faster still thanks to the parallelism provided by the GPU. The quality of our method is comparable with MCD. Our method is also shown to generate lesser geometric deviation when compared to super independent vertices [69].

Table 4.1: Statistics of the simplification result in different models. Timing is in seconds. Quality is measured with Hausdorff distance [69].

Model	Triangles	Target	QSlim		MCD		Our Method	
			Time	Quality	Time	Quality	Time	Quality
Gargoyle	1,726,420	86,320	22.86	0.01966	6.79	0.0478	1.30	0.03542
Dragon	7,218,906	360,944	109.02	0.0816	31.78	0.084	6.39	0.0836
Lucy	28,055,742	1,402,787	469.089	0.0187	124.2	0.0318	28.03	0.032

Figure 4.3: Comparison of the simplification results for the Gargoyle model.

Without the implementation of [69], we do not do a proper comparison of the timing performance. However, it is reported that their method observes a speedup of x1.5-4.2

compared to QSlim. Our method comparatively sees a speedup of $\times 10-17$ against QSlim, which implies our method having at least $\times 2-4$ speedup of [69]. In addition, our method can generate higher quality simplified meshes. We show the Hausdorff distance in Table 4.2, the comparative data were obtained directly from the [69] paper. To ensure the correctness of the comparison, we have evaluated our result using the same “PolyMeCo” software [94] as used by [69].

Table 4.2: Comparison of Hausdorff distance between [69] and our method at different simplification levels for the Gargoyle model.

The target number of triangles	Our method	[69]
25%	0.009	0.01524
10%	0.021	0.03357
5%	0.03542	0.05961

Even though [69] suggests using only a subset of the selected edges for collapse to improve the quality, such a mechanism is very hard to control. If the subset is too small, many independent edges are not allowed to collapse, which largely reduces the performance provided by the GPU. On the other hand, if the subset is too large, the quality of the simplified meshes is compromised. Our method, on the other hand, only needs to select the size of each thread subgroup (i.e. $\mathbf{p}$). We find that the subgroup with size 8 works well for all our models.

Figure 4.4: Comparison of the simplification results for the Dragon model.

Figure 4.5: Simplification of the Lucy model at different levels. The number indicates the percent of the remaining triangles.

(a) Original Mesh

(b) QSlm

(c) MCD

(d) Our Method

Figure 4.6: Comparison of the simplification results for the Fan Disk model. The red circle highlights the high error rate edge collapse performed by MCD compared to the other methods.

4.7.2 Comparison

Compared with existing methods (QSlim, MCD and GPU-based simplification with super independent vertices), our methods have the following advantages:

- Our method can collapse multiple edges at the same time and adopts GPU for parallel computation. The processing time is largely reduced compared to a CPU based MCD and QSlim.
- Each iteration, our method selects the best edge within a sub-group. If the mesh is targeted to be largely simplified, the probability that we will choose the wrong edge will exponentially decrease just as the CPU based MCD. Resulting in the improvement of simplified mesh quality.
- Each iteration, we only need to lock the one-ring triangles adjacent to the edge. [69], on the other hand, requires a three-ring of vertices adjacent to guarantee the independence of super vertex. The locked area of our method is smaller. Leading to lesser thread conflicts in our method, hence improving our parallelism.
- Unlike [69], we do not need to sort the resulting edges according to the edge costs. This can simplify the implementation and reduce the computational and memory overhead.

4.8 Summary

In this report, we present a novel GPU-based decimation algorithm for reality captured models. Our method is based on QEM edge collapse, parallelizable multiple-choice decimation, and super independent vertices.

The choice of multiple-choice decimation as the foundation to allow us to reduce the memory overhead and implementation complexity. While also granting us further use

of parallelism and better fine-tune control over the quality of decimation. Combined with super independent vertices, we successfully developed an implementation of a fully GPU-based decimation framework that achieved the first of our goals of improving computational efficiency.

4.8.1 Future Works

Although our approach can reduce the memory overhead compared to other algorithms, our method still requires the entirety of the model to be loaded into the GPU's video RAM (VRAM). Since the size of VRAM is much smaller than system RAM, our targeted high-resolution model cannot easily fit into the available VRAM size, restricting the use of our method to the availability of VRAM.

As future work, we are looking into improving the algorithm to enable the out-of-core capability for our GPU-based decimation algorithm to resolve the above-mentioned issue. Using the theory behind multiple-choice decimation, we only need to sample part of the whole mesh to perform the decimation. This opened the possibility of performing random sampling in an out-of-core manner. Specifically, while we stream the mesh from disk, we can perform the random sampling on the parts already streamed into memory and apply our GPU-based decimation on just the partially loaded mesh.

CHAPTER 5: SIMPLIFICATION OF TEXTURED MESH

This chapter will talk about how we overcome the traditional problem in texture-aware mesh simplification by dividing the geometry and textural information into the separate part and simplifying them individually. We will also investigate the details of the parallelized texture map transfer algorithm, which is an important part of the proposed pipeline that allows for the separation and reaggregation of geometry and textural information of a textured mesh.

With the recent advent of laser scanning and photogrammetry mesh reconstruction technique, models of extremely high resolution both in terms of geometry as well as texture are now much easily obtainable by professional and hobbyist alike. However, the massive data size of these reality captured models currently still posed a significant challenge to many conventional computer graphic applications. In order to preserve both the performance speed and visual quality of reality captured models, a multi-resolution mesh approach was explored here.

Applications that perform real-time rendering usually desire a multi-resolution model that can provide a different level of geometric approximation of the original surface depending on the viewing distance. Along with these geometric approximations, the model's textures mapping needs to be considered as well to achieve the desired level-of-details.

We present a method for transferring high-resolution texture maps information into a lower resolution model through accelerated ray-casting.

Disclaimer: Koh Naimin is an employee of Autodesk, Inc. and is directly involved in the development of the Autodesk® ReCap Photo™ product [95]. Autodesk® ReCap Photo™ is also used to generate most of the datasets and results as presented in this chapter.

5.1 Introduction

Figure 5.1: Transfer texture map from source mesh (left) to target mesh (right). Even at only 1% geometric and 0.015% texture data size the visual difference between the two at this viewing distance is virtually indiscernible.

Decades of research have gone into studying how to simplify polygonal meshes to reduce complexity while still preserving visual fidelity. Two widely used fundamental polygonal reduction method is using edge-collapsed that is used to construct a progressive mesh [63], and the vertex clustering method that is used in Shaffer, Eric. et al. [96]. Both Maglo, A. et al. [97] and Luebke, D. [98] have an excellent in-depth survey of the various other methods of mesh simplification. These methods, however, generally focus only on the geometric simplification and exclude the other visual property of the model such as texture mapping and shading.

The main difficulty of simplifying meshes with texture mapping is that the texture mapping information, specifically the UV coordinate of each vertex is cannot be interpolated if the coordinate lay on two separate texture charts. Traditionally, to overcome this limitation, vertices with the same position but different UV coordinate are duplicated, and the triangles faces they belong to are disconnected where the UV coordinate differs. This allows the proper interpolation of UV coordinate on triangle

patches that belong to the same texture charts, at the cost of having disconnected edges between triangles patches that belong to different texture charts.

For rendering, this isn't a major issue as the final visual render of the mesh will still be correctly textured and shaded. In the context of mesh simplification however, this issue causes an unwanted side effect, where along the edges that lay between two texture charts cannot be simplified without breaking the interpolation correctness. To circumvent this fundamental problem, classic mesh simplification chose to lock these edges and disallow any simplification operation on these edges.

Figure 5.2: Mesh heavily decimated under texture continuity constraint. Dense triangulation along texture boundary (shown in yellow), while large triangles dominate the interior of texture charts (shown in blue).

While these methods do retain the overall correctness of the UV coordinates after simplification, when examining the quality of the resulting mesh will easily yield issues. The density of the locked edges of the triangle patched will be in stark contrast with the inner areas since the inner areas have undergone mesh simplification and largely

reduced triangle density when compared to the locked edges. This issue manifest problem both visually and on the efficiency of the simplification algorithm itself. When inspecting the wireframe of the mesh, one can easily make out the high-density edge around each triangle patches due to the stark difference in triangle density. This visual aspect of the problem usually does not pose too big an issue when the textures are applied onto the mesh covering up any perceivable mesh density issues.

The detrimental effect on the maximum efficiency of the simplification algorithm, however, is an important issue that got overlook by other researches. Due to the scheme of using locked edges, we can determine the maximum simplification efficiency as a function where the minimum number of edges after simplification must be more than the number of locked edges. This imposes a bound on maximum simplification efficiency that is highly dependent on the quality of the texture mapping itself since the triangle patch's edges must lay on the edges of texture charts. On a heavily fragmented texture map, it is necessary to enforce a large number of locked edges to preserve the UV coordinate correctness. Which in turn massively lowered the simplification efficiency of any algorithm that relies on the locked edges.

To completely avoid the issue of locked edges, we choose to instead separate the simplification of the mesh into its two components of geometric and texture aspects. This allows us full flexibility on the choice of both geometric and texture simplification, while at the same time avoid the above-mentioned problem. We then propose the parallel texture map transfer to allow for the recombining of the separate simplified geometry and texture back into a complete simplified textured mesh.

5.2 Proposed Method

To overcome the traditional issue with the simplification of textured mesh, we decided to separate the simplification of geometric and texture information into individual isolated processes. Beside just overcoming the earlier mentioned issue, we are now also able to choose the best algorithm to perform geometric simplification and texture simplification since they are now separate from each other. This allows us to obtain the

best quality of both geometric and texture simplification without running into an earlier issue where the simplification criteria from one will restrict the quality of the other.

While it is beneficial to separate the geometric and textural simplification, we run into the question of how do we re-combine the separated geometric and textural information back after their individual simplification process? To answer that we introduce the process of texture map transfer which allows the transfer of texture map information from one mesh to another mesh. This allows the transfer of texture information from the original textured mesh onto the simplified mesh, thereby recreating a new simplified textured mesh.

5.3 Separability of Geometric and Texture Simplification

The idea of separating geometric and texture simplification gives a different perspective to the traditional problem of textured mesh simplification. This allows for the breaking of all previous restrictions when trying to preserve UV coordinate correctness, allowing us to pick the algorithms that provide the best simplification quality and efficiency.

With the separation of geometry and texture simplification, we can in effect choose any combination of geometry simplification and texture simplification algorithm. By choosing the most efficient and best quality simplification, we can now achieve much higher quality simplified texture mesh compare to more traditional approaches.

5.3.1 Geometric Simplification

In keeping with our thesis goal of having high computation efficiency, we will be using the GPU-based multiple-choice mesh simplification that we described in chapter 4. Although slightly lower in quality compared to the serial version of the QEM mesh simplification, the improvement in processing speed will out-weight the quality loss, when tackling the massive size of the typical reality captured model.

Any other geometric-based mesh simplification algorithm can be used as the replacement to our choice of GPU-based mesh simplification, depending on your specific use case needed and requirements. This is one of the main benefits of separating geometric and texture simplification.

5.3.2 Texture Simplification

To reduce the texture size of the reality captured model, two approaches are available: resizing the texture or regenerating the texture mapping. We will be regenerating the texture map from the simplified mesh and not the original model, this allows us to change the number of textures map and allow us to decouple the geometric and texture simplification into two different stages.

5.3.2.1 Resizing the Texture

The simplest of the two approaches, we can just uniformly resize the texture map from a higher resolution to a lower resolution. This method is effective in its efficiency in down-sampling texture resolution and simplicity of the method. Due to the relative scale nature of UV coordinates, there is no need to explicitly transform the original UV coordinates of the mesh vertices. This method, however, is unable to reduce the number of textures that are originally created.

5.3.2.2 Regenerating the Texture Mapping

Figure 5.3: Transfer from source mesh's multiple high-resolution textures (left) to one lower resolution texture mapped by the target mesh (right).

This method involved the regeneration of texture mapping information for the mesh, this follows much of the process in creating the original texture mapping in the first place. Since we are regenerating the texture mapping information from the simplified mesh, we have full control over the numbers and resolution of the resulting texture maps. This further allows us the flexibility to control the quality and efficiency of the texture simplification. The process of regenerating the texture map involves two main processes: mesh parameterization and texture packing.

5.3.2.2.1 Mesh Parameterization

Mesh parameterization is the process where we attempt to find the bijective mapping from a 3d surface into a 2d map. One can imagine trying to flatten a patch of 3d surface to try to map them to a 2d chart. Mesh parameterization can only be performed on a disk topological surface which can be “flatten” onto a 2d domain. Usually, a water-tight manifold mesh will require a scissoring algorithm to “cut” the mesh into several patches of disk topological surface, which can be parameterized [99].

Except for completely flat 3D surfaces (such as a single triangle), the process of mesh parameterization will inevitably cause texture distortion on the resulting texture map. Texture distortion can be defined as the difference between the area or the internal angle of the triangles in the 3D surfaces and the resulting triangles texture-mapped charts. However, the two definitions of texture distortion of area or angle are incompatible to each other. To preserve the area of the mapped textures chart will incur distortion on the internal angles of the triangles that form the chart [5], [7], [100]. Similarly, to retain the same inner angles of the mapped triangles, it will cause distortion in the area of the resulting charts. This result in several choices of optimization algorithm to perform mesh parameterization, each optimizing for a different definition of texture distortion [6], [100].

5.3.2.2.2 Texture Packing

After mesh parameterization, we will have a set of 2D texture charts that map back to the 3D triangles. These texture charts only define the shape and relative scale of the texture mapping but do not have the actual 2D texture coordinate yet. Texture packing is the algorithm to collect and fit all the texture charts into individual textures image [101], [102].

To reduce the total size of the texture image, the texture packing algorithm must minimize the waste space between texture charts. This problem is part of the class of NP-hard bin packing problem, only approximate solutions have been proposed. We use a physic-based packing algorithm that simulates 2D gravity on each of the 2D texture charts and allows them to fall into the area of the texture image and stabilized to obtain the final texture coordinates [103].

5.3.3 Section Summary

At this point, we have already simplified the mesh and the new texture mapping information of the simplified mesh. However, the two geometric and texture components are still separated. We still need to be able to transfer the original model's

texture information onto the new set of texture maps of the simplified mesh. This process of texture map transfer will be described in detail in the next section.

5.4 *Parallel Texture Map Transfer*

Texture map transfer is the process where we allow the transfer of texture information from one model's texture maps onto the texture maps of another model. This method works well if the geometric difference between the two models is very small, this once again highlights the importance of geometric simplification methods to have both high compression ratio and low geometric deviation.

By transferring the original model texture information to the simplified model, we can retain the high visual impact of the textures while still manage to use the best geometric simplification method to simplify the model.

5.4.1 *Algorithm Overview*

Figure 5.4: Transfer map algorithm overview.

5.4.1.1 Definitions

A *source mesh* is defined as the highest resolution mesh of the multi-resolution models that have proper texture coordinate mapping (i.e., UV mapping) to one or more high-resolution textures.

While a *target mesh* is defined as one of the sets of simplified meshes derived from the source mesh with their own sets of texture coordinate mapping but may not have any texture information (i.e., blank textures). Note that the *target mesh's* texture coordinate mapping, number of textures and resolution of textures do not strictly need to match that of the *source mesh* or the other meshes in the same multi-resolution model.

5.4.1.2 Overview

The texture map transfer algorithm utilized both rasterization and ray-casting to determine the mapping between the original model and the simplified model. The primary problem the transfer map algorithm must solve first is finding the correspondence between the *target* and *source textures*, before any transfer of texture information can happen. In order to solve this problem, we work starting from the mapped texels on the *target texture* and find the corresponding 3D surface point back on the *target mesh*, from which we bridge the relationship of the *target* and *source meshes'* surface via ray-casting. Once we can establish the relationship between the *target and source meshes'* surface, the texture information extraction and subsequent writing back to the *target texture* can happen.

Now our *target mesh* has proper texture coordinate mapping but these mapping currently points to empty textures. In order to transfer the texture information from the *source mesh's* textures into the *target mesh's* textures, we need a way to correlate every single UV mapped texels in the *target mesh's* textures to that of the *source mesh's* textures. To achieve this, for each mapped texel of the *target mesh's* textures, we find its corresponding point on the surface and ray-cast toward the *source mesh's* surface, the closest point on the *source mesh* that intersects the ray are then used to sample into the *source mesh's* textures which get written back into the *target mesh's*

texel. We can bridge the two 2D texel space by going through the corresponding 3D space.

Because we are performing the ray-casting for each mapped texel which is conceptually similar to backward mapping rasterization, there is no redundant ray-cast thereby ensuring only the minimal amount of computation required is performed.

5.4.1.3 Rasterization

Figure 5.5: Rasterization of a triangle formed using vertices UV-coordinate into raster pixels.

Since the information on which texels were mapped are not stored in the textures but instead need to be derived from the texture coordinates of each *target mesh's* face, to find all the mapped texels we need to sample for, we need to rasterize each face of the *target mesh* unto their respective texture.

Rasterization is the process of taking 3D shapes and rendering it onto a 2D surface. The rasterization process we are using is done in a different sense than conventional 3D rendering rasterization which renders pixels to monitor screen after transforming the 3D vertex through the various rendering pipeline transforms (world, view, projection, clipping then rasterization). In our case, we can map each 3D vertex into a corresponding 2D coordinate simply by using the texture coordinates of the vertex as the 2D coordinate to rasterize. We also pass the vertex's world position and normal as

an attribute to be interpolated by the rasterization process, as well as the triangle and texture id as we will be needing them later.

Once each of the 3 vertices of a face is converted to the 2D coordinate space (i.e., texture coordinates), we can start rasterizing the 2D triangle onto a texture of any resolution, since the texture coordinates space is a normalized space that can be rescaled to any texture resolution. Since the standard scan line algorithm is an incremental algorithm that is not easily parallelizable, we are using an embarrassingly parallel variant from Giesen's blog on rasterizer [104], [105] for this work.

Once each *target mesh's* faces are rasterized into their respective textures, we get a list of raster pixels which contain the texel position (2D), interpolated world position (3D) and surface normal relative to the *target mesh* as well as the id to the texture the raster pixel belong to.

5.4.1.4 Ray-Casting

Figure 5.6: Remapping the raster pixel back to their corresponding 3D position on the mesh surface.

Since the ray-casting sampling step size is one texel in the texture space, to figure out where to cast the ray from and toward which direction in world space (3D), we need to

remap the texture space's texels position (2D) back to the triangle face in world space (3D). Conveniently, as an added effect of rasterization, the interpolated attributes are exactly sampled in per texel step size. Hence, we can simply use the interpolated world position and surface normal of each raster pixel as the ray starting point and ray cast direction respectively.

Figure 5.7: Ray-casting failure case due to geometric difference.

Figure 5.8: Using cage offset inverted ray-casting to handle the failure case.

The geometric difference between the *source* and *target mesh* caused by the mesh decimation process created a critical ray missing problem with the ray-casting approach. Specifically, there is no guarantee that the *source mesh's* surface will be found along the ray direction, in which case the ray-casting will miss or wrongly project to a nearby surface. To overcome this problem, we use the cage offset inverted ray-casting method in which we offset the ray-casting starting point by an empirically set

small factor along the ray-casting direction thereby forming a cage around the *target mesh* which should attempt to completely envelop the *source mesh* then the ray-casting direction is inverted to point back toward *target mesh*. In this ray-casting setup, if the cage envelope the *source mesh* and the geometric difference between the two meshes aren't too large (which is guaranteed by the mesh decimation method we chose), from our experimentation almost all ray-cast sampling tends to be successful.

Figure 5.9: To get the actual distance of intersection, we need to account for the cage offset distance.

The actual ray-casting per raster pixel is done through standard line segment and triangle intersection algorithm from Ericson's book [106], which will report the barycentric coordinate and distance along the ray of where the ray intersects with the triangle, as well as the triangle and texture id of the intersected triangle. We empirically chose twice the cage offset distance as the default ray-casting distance (i.e., line segment length) to avoid unnecessary intersection check on triangles too far from the ray-casting starting point on the cage. Because we require the intersection point that is closest to the *target mesh's* surface and not the cage from which we are ray-casting from, we need to properly account for the actual ray-hit distance by subtracting the reported ray-hit distance by the cage offset distance (Figure 5.9). To improve robustness, we also added a retry mechanism that will iteratively double the ray-casting distance should the ray-cast fail to hit up to a specified number of retries.

5.4.1.5 Generating Textures

Once we have the intersection result for each raster pixel (i.e., mapped texels with interpolated values), we need to figure out where to sample the *source mesh's* texture information to transfer to the *target mesh's* textures. From the ray-cast result, we have the intersected triangle and texture id as well as the barycentric coordinate of the intersection point on the triangle, using this information we first need to find the UV coordinate of the intersected point. Using the barycentric interpolation of the intersection point, we can find the UV coordinate of the intersection point. Once we find the UV coordinate of the intersection point, we can now sample into the *source mesh's* texture.

The way we sample the texture information depends on the texture resolution between the *source* and *target meshes'* textures.

If the *target texture's* resolution is higher or the same as the *source texture's*, then a simple point sampling is sufficient based on sampling theory. Else if the *target texture's* resolution is lower than the *source texture's*, we perform a subsampling with Gaussian pre-filtering [107] to reduce aliasing effect on the generated texture. The Gaussian filter size is doubled for each half size reduction of the *target texture's* resolution compared to the *source textures*.

Once we have sampled the *source texture*, we can now write the color sample into the corresponding texel location in the *target texture*. Recall that each of our raster pixel used to perform the ray-casting from is derived from the *target texture's* mapped texel, hence we already know exactly which texel to write into from each raster pixel's texture space location and texture id. For robustness, if the same texel in the *target texture* is written to multiple times such as in a situation caused by overlapping UV mapping or rasterization without proper tie-breaking handling, the total color value is accumulated and averaged out after all the ray-cast sampling is done to get the final color value.

5.4.1.6 Accelerated Ray-Casting

The number of ray-cast operation is equal to the total mapped texels in the *target mesh's textures*. As the number of texture and resolution of the *target mesh* increase, so too does the number of ray-casting operations. If we are to perform each ray-casting intersection check for each mapped texels with all the faces of the *source mesh*, the

number of operations can easily go into the billions. In particular, the intersection check between the line segment and triangle dominant most of the processing time. Thus, we need a method to early reject as many triangles for intersection tests as possible without causing any false negative.

To achieve this, we explored two spatial partition schemes, the octree partition and the bounding volume hierarchy (BVH) [106]. The idea behind spatial partition is that by abstracting the group of nearby mesh faces into a bounding volume then check can be done on the bounding volume instead to decide whether the further work needs to be done to the child of the bounding volume or reject the bounding volume altogether. Usually, the choice of the bounding volume is of a much simpler geometric such as sphere or cuboid that allows for fast and efficient intersection testing compared to the more complex triangle intersection. We will construct these spatial partitions out of the *source mesh* so that the ray-cast query from the *target mesh* can be accelerated.

The idea of octree as a spatial partition scheme is that starting with the entire model bound within a cube volume, then subdividing the cube uniformly into 8 smaller cube (sub-space), then the repeatedly each sub-cube can be further subdivided into 8 even smaller cubes if the number of triangles bound by the cube is larger than a pre-determined value. The construction of octree is a simple recursively top-down approach and the bounding volume is defined to be a cube that allows for the simpler computation of a cube size and location based on the octree depth.

Our goals are to only investigate the leaf level sub-cubes that are intersected by the ray, as any other cubes will never be hit by the ray. The choice of a cube as the bounding volume, however, meant that every subdivision depth will spawn 8 more sub-cube regardless of whether some of the sub-cube contain any triangles, resulting in some memory wastage. The depth of octree is sensitive to the local density of triangles, where a small but dense part of the mesh can cause a series of very deep sub-cubes to be created to meet the terminating condition.

The bounding volume hierarchy (BVH) partition scheme similarly can start with the entire model bound within a cuboid volume, then the bounding volume is split into two smaller bounding volume by choosing a sorting axis and sorting all the triangles based on the sorting axis, then using the median as a split point, split the sorted triangles into

two sub-space. In each sub-space, a new bounding volume is computed and further split, if the number of triangles contains in this bounding volume, is higher than the predetermined value. In our implementation, the sorting axis is chosen by round-robin over the X, Y and Z axis at each depth level, which results in a more uniform spatial partition over all the 3 main axes. The various scheme can be used for sorting the triangle based on the sorting-axis, one could use the center of the triangle as the sorting point or simply pick one of the vertices of the triangle for sorting, for efficiency and no loss of quality we chose the latter method.

Both schemes work fine for our purpose but let's have a brief discussion about the pro and cons of each scheme. The construction of the BVH is more complex than the octree due to the need to re-compute the bounding volume of each sub-space compared to the octree which gets the new bounding volume by uniform subdivision. The depth of the BVH is bound by $\log_2(N)$ since the partition is strictly binary and balanced while the octree is unbound and sensitive to local triangle density. Due to the general lower depth level and less wasted memory usage (each sub-volume in a BVH will contain at least a triangle), the BVH generally uses less memory when compared to an octree. The more important factor, however, is the querying performance, the BVH overall querying performance is better than the octree due to the lower depth and having less sub-volume to check against (2 sub-spaces vs 8 cubes). So, the verdict is that while the construction time of the BVH is longer than the octree, this one-time cost is amortized by the querying speedup gain when the billions of ray-cast queries are performed, hence we finalized on using BVH as the spatial partition scheme.

A survey by Vinkler, et al. [108] comparing the ray-casting query performance between BVH and Kd-tree which an octree can be generalized into similarly reflect our findings as well.

5.4.1.7 Parallelizing

As you might have realized as we walked through the algorithm, that most part of the algorithm is embarrassingly parallelizable so in this section we will highlight the various portion that will be embarrassing not to parallelize.

- 1) Each face of the *target mesh* can be processed independent of each other, hence any processing done each face can be parallelized.

- 2) Recall why we chose a parallelized rasterization algorithm over the standard incremental scan-line algorithm.
- 3) Each raster pixel is independent of each other, hence each of them can perform the ray-casting query in parallel. Note, special handling is needed when accumulating and writing the final color in case of overlapping UV (see the section on Generating textures.).

5.5 Experiment

Figure 5.10: Transfer texture map from source mesh (left) to target mesh (right). Even at only 1% geometric and 0.015% texture data size the visual difference between the two at this viewing distance is virtually indiscernible.

For the experiments, we showcase the archival grade photogrammetry reconstruction of Garuda from Singapore's Haw Par Villa, used with permission by Autodesk Inc. and credited to Murali Pappoppula for the capture. This model was reconstructed with more than 800 high-resolution still-shots on a tripod and timer to remove all possible vibration on the camera which might disrupt the quality of the photos and reconstruction. Due to this high-quality acquisition, the photogrammetry engine by Autodesk® ReCap Photo™ [95] can produce archival grade digital replication that is deemed fit for the museum's

archival purpose of priceless artifacts. The original source model weight in at a total file size of 60Gb, with 118 million triangles and 127 different 8K resolution textures. This massive size once again highlights the importance of this thesis's goal in reducing the size of reality captured models.

As shown in the side by side rendering above, at the viewing distance the two models are virtually indiscernible from one another. Despite the massive size difference between them of 99% reduction in polygon count and 99.985% reduction in texture size.

Figure 5.11: Source mesh, Garuda model with 118 million triangles and 127 x 8k textures. Model credit to Autodesk

Figure 5.12: Source mesh close-up view, showing the high density of triangles faces. Model credit to Autodesk

Figure 5.13: Target mesh, Simplified Garuda model with 1.18 million (1% of source mesh), re-parameterized to 2 x 8K textures (0.015% of source texture). Credit to Autodesk

Figure 5.14: Target mesh close-up view, showing the result of mesh simplification to 1% of the source mesh. Model credit to Autodesk

Figure 5.15: Mesh difference report from Autodesk® ReCap Photo™ [95] between the source mesh and simplified mesh (1% of source mesh). Most of the model is within the 0.05mm threshold (green region), only a few small regions deviate more than the threshold circled in white.

Because we have separated the geometric simplification from the textural simplification, we are able to much high-quality simplification with a minimum geometric deviation of 0.05mm by using a better simplification algorithm. Similarly, the simplified geometry does not suffer from the high-density seam issues that affect methods that lock boundary edges. To emphasize on the seam issues, we must consider the original model has 127 different textures which contains thousands of separate texture charts, the percentage of boundary edge is in the excess of 17% of the total edges. Using boundary locking algorithms, the maximum simplification rate will be capped 83% and will be unable to reach the current simplification rate of 99%.

By re-parametrizing the texture map of the simplified model and transferring the texture maps, we can reduce the number of textures required to still accurately retain the visual quality of the texture model. Without the reparameterization step, the simplified model will be left out any texture mapping information hence unable to be used as a textured model. To accurately transfer the texture information from the original model, our transfer map algorithm has the capability to select the visually correct pixel from the 127 different texture maps to populate the 2 new texture maps of the simplified model. The ray-casting approach has shown that it is able to accurately select the most appropriate pixel from the original model.

5.5.1 Other Application

In addition, to just transferring the diffuse texture information from the *source mesh* to the *target mesh*, we have created an extension to the algorithm to allow for the generation of normal, displacement, ambient occlusion and even approximate measurement of the difference between two meshes using the information already collected or computed by the algorithm.

When the ray-cast intersects with the *source mesh's* surface, we can also retrieve the surface normal of the triangle at the intersecting point and use this normal to generate a normal map.

Similarly, the displacement map can be generated using the ray-cast result's intersection distance as the displacement value. Mesh difference can be quite closely approximated with the same displacement value as that of the displacement map.

By performing a stratified ray-casting, not toward the *source mesh* but instead back to the *target mesh* itself, we can perform self-occlusion checks and generate an ambient occlusion map for the *target mesh*.

The transfer map algorithm generally still works even if the transfer is between completely arbitrary meshes, so long as the ray-casting sampling can project onto the *source mesh* the transferring of the map can happen. But in this case, there isn't much guarantee on the usefulness of the result. However, a useful application of this capability is that given a complete model and some smaller portion of the same object captured/modeled in a different way, we can transfer from one model to the other even if there is a significant geometric deviation between the models. In these extreme cases, the ray-casting cage distance needs to be set to a higher value.

5.6 Summary

5.6.1 Future Works

As the algorithm relies heavily on rasterization for the early part of the processing, we can re-implement the current CPU rasterization implementation into a GPU implementation by adapting the rendering pipeline to render the *target mesh*'s triangles into the various textures, thereby taking full advantage of hardware rasterization and interpolation to accelerate the computation even further.

Similarly, GPU enabled ray-casting algorithms such as Purcell, et al [109] exist as well that when combined with the GPU rasterization have the potential to complete most of the heavy computation of the algorithm directly in the GPU. The limitation on GPU's memory is a concern however, as it is generally impossible to store the *source mesh* completely on GPU's VRAM. This problem could be possibly tackled by processing each sub-part of the mesh individually by exploiting the spatial partitioning that we have

already built, hence only a small portion of the total mesh data need to be resident at one time.

5.6.2 In Closing

We have discussed the problem with how the traditional algorithm tried to solve the problem of textured mesh simplification by attempting to incorporate both geometric and textural information during simplification. Noting the major drawback of such approaches, we propose the idea of separating the two components and simplifying them individually.

With the separation of geometric and textual simplification, we gain the flexibility to choose the most performing simplification algorithm. This flexibility allows the chosen simplification method to circumvent the traditional issue and achieve maximum efficiency possible for the method.

To bridge the problem of recombining geometric and textural information back into one unified model, we proposed the parallel texture map transfer algorithm which for highly accurate transferring of the color pixel from the original model's texture maps to the simplified model.

The full pipeline of separating and then recombining the simplified components have been demonstrated to produce a massively simplified model that is visually indistinguishable from the original model at a certain viewing distance.

CHAPTER 6: REAL WORLD DEMONSTRATION

Figure 6.1: Photorealistic reconstruction of the NTU's first chairman statue.

To demonstrate real-world results coming from the contributions listed in this thesis, we will take a trip through the photogrammetry reconstruction pipeline with a real-world capture of the statue of the first chairman of Nanyang Technological University captured specifically for this thesis.

By processing this capture through a reconstruction pipeline, we can have a better in-depth understanding of the reconstruction pipeline mentioned throughout this thesis. From

the outputs of the reconstruction pipeline, we can then utilize the various compression and simplification methods proposed contribution in this thesis to gauge the true effectiveness of the proposed methods on a real-world model.

We will see the compression efficiency of the proposed octree truncation point cloud compression method, followed by the GPU based multiple choice scheme simplification of the reconstructed mesh, finally, we will simplify the textured mesh with the geometric and texture separate simplification.

Disclaimer: The author, Koh Naimin was actively involved in the development of the Autodesk® ReCap Photo™ [95] and related photogrammetry technology from Autodesk. The product is used here as an example of academic research and not for any commercial purpose.

6.1 Reconstruction Pipeline Setup

Using the photogrammetry software ReCap Photo™ from Autodesk [95], we can gain access to a commercially available photogrammetry reconstruction pipeline. Although other open-source structure from motion (SfM) [110], [111] and multi-view stereo (MVS) [112] programs can be utilized to create our own reconstruction pipeline, we wish to explore the real-world utilization of reality captured model by understanding how an average user will be able to use a commercially available product to achieve their reality capturing goals.

6.1.1 Photogrammetry Inputs

The inputs of the photogrammetry pipeline consist of only overlapping photos taken from multiple angles of the subject. The photoset consists of 340 photos taken with a Canon EOS 7D Camera and iPhone X of the statue subject. The GPS information is available on the photos. No further modification such as color adjustment or resizing was done to the photoset.

Figure 6.2: Sample of the input photoset used in the reconstruction.

6.1.2 Reconstruction Process

Figure 6.3: Screenshot of Autodesk® ReCap Photo™ when uploading the photoset.

By using ReCap Photo™ [95], we can select the entire photoset and upload it to Autodesk's cloud service to perform the heavy computation of the photogrammetry reconstruction pipeline on the cloud servers.

Figure 6.4: Dense point cloud generated from the reconstruction.

After the SfM and MVS processes had finished, we will have obtained our first main reconstruction output: the dense point cloud model. The point cloud model consists of 6,543,721 points.

Figure 6.5: Mesh generated from the reconstruction.

The subsequent surface reconstruction algorithms will then generate the second output: the mesh model. The mesh consists of 4,669,864 triangle faces and 2,335,410 vertices.

Figure 6.6: Texture mesh generated from the reconstruction (Top). The 10 texture maps generated (Bottom).

Finally, with the mesh parameterization and texture information projection, we will be able to obtain the final textured mesh model. This includes the 10 8K resolutions texture that was generated.

6.2 Experiment

Once we have downloaded the reconstruction outputs from the Autodesk® ReCap Photo™ [95]. We can start running experiments on the reconstruction outputs using our earlier listed contributions.

Figure 6.7: The decoded point cloud using our Truncated Octree Point Cloud Compression Encoding.

We first test the effectiveness of our truncated octree point cloud compression encoding on the dense point cloud model. From the experiments, we can observe approximately 79.44% reduction in file size compared to the original point cloud model of 76.6 MB to 15.3 MB. There also little geometric error as was expected with the known accuracy of the methods with the only root mean square error of 0.000009.

Figure 6.8: The simplified mesh using our GPU-Based Multiple-Choice Scheme Mesh Simplification.

We subsequently test the simplification rate of the GPU-based multiple-choice scheme mesh simplification algorithm on the mesh model. From the experiments of simplifying the model to 1% of the original triangle count, we can observe 4623166 reductions in triangle counts and reduction in file size from 185 MB to 1.6 MB for a total 99.13% file size reduction. Due to the simplification method using the highly accurate QEM error metric, the simplified mesh also exhibits a similar low geometric root mean square error of 0.000096.

Figure 6.9: Close up view of the simplified wireframe and high visual quality obtained by our Parallel Texture Transfer Algorithm.

Lastly, we run the experiments of transferring the texture information from the original textured mesh onto our newly simplified mesh from the previous test. We also reduce the total number of textures used by the model from 10 8K resolution texture to just a single 8K resolution texture after mapping. The 10 texture files use 95.3 MB of space while the newly transferred texture only occupied 16.9 MB of space, resulting in an 82.26% texture space-saving.

Figure 6.10: The single texture generated by our Parallel Texture Transfer Algorithm that contains the color information selected from the 10 previous textures.

Even with only one texture used, the texture transfer algorithm can accurately pick the correct color for each mapped pixel in the texture. This high accuracy in color selection greatly diminishes the visual degradation expected from the massive reduction in available texture space. Figures 6.11 and 6.12 showcase how similar both textured meshes appear despite the simplified model only contain 10% of the texture and 1% of geometric information compared to the original model.

Figure 6.11: Original Reconstructed Model

Figure 6.12: Our simplified model with transferred texture

Table 6.1: Summary of the space-saving obtained for each model representation as well as total file space-saving.

	Original (MB)	Simplified (MB)	Saving (MB) / (%)
Point Cloud	76.6	15.3	61.3 / 79.44%
Mesh	185	1.6	183.4 / 99.13%
Textures	95.3	16.9	78.4 / 82.26%
Total Size	356.9	33.8	323.1 / 90.58%

In total, we have effectively reduced the overall file size of the model from 356.9 MB down to 33.8 MB, vastly reducing the model size by 90.58% without visible loss in quality at the current viewing distance.

We had also demonstrated how we can seamlessly integrate our contributions to a real-world reality capture photogrammetry pipeline. This integration had also successfully demonstrated the capability to vastly reduce the complexity of high-resolution reality captured models without sacrificing much of the visual and geometric quality of the original capture.

CHAPTER 7: CONCLUSION

This thesis has examined the simplification and compression of the three main model representations (point cloud, mesh, and textured mesh) resulting from the reality capture pipeline and proposed novel ways to simplify these representations as part of this research. We have also successfully demonstrated the effectiveness of the contributions with a demonstration of the proposed algorithm on a real-world photogrammetry reconstruction specifically captured for this research.

We provided a high-level introduction to reality capture technology. Which covers the general pipeline of a multi-modal reality capture pipeline and the core technologies that make up the reality capture technology stack. We also introduce the various model representation generated within the reconstruction pipeline which is the focus of simplification in this thesis.

Reviews of related technologies relating to compression of the point cloud, simplification of mesh and texture aware mesh simplification were also presented to provide a general trend of the current literature in this field of research in simplification of reality capture models.

Our contribution of “Parallel Point Cloud Compression Using Truncated Octree”, utilizes the concept of truncated octree to unlock the flexibility of point cloud encoding scheme to be able to handle point cloud distribution of any kind while retaining a high level of quantization accuracy.

The following contribution of “GPU-Based Multiple-Choice Mesh Simplification Scheme”, uses the probabilistic analysis of multiple-choice bin selection to accurately pick the edge to decimate, resulting in both a highly performant and accurate simplification algorithm.

The work on “Simplification of Textured Mesh”, contains two separate main contributions. The first is the introduction of the concept of “Separability of Geometric and Texture Simplification”, which allows for the combinations of the best algorithm for individually simplify geometry and texture aspect. Secondly, we introduced the

algorithm of “Parallel Texture Map Transfer” to allow for recombination of the results from the separate simplified geometry and texture map into the new simplified textured mesh.

7.1 Application

To demonstrate the real-world application of the works contained within this thesis, we also included a real-world demonstration of an entire photogrammetry reconstruction pipeline which utilized the proposed methods to simplify the resulting point cloud, mesh and textured mesh. This highlights both the actual efficiency of the proposed method and the size problem posed by high-quality reality capture reconstructions.

With the real-world demonstration of the effectiveness of our proposed contributions, we can conclude that we had achieved the research goals we had stipulated at the start of the thesis. Specifically, the goals of being able to massively reduce the file size of reality captured model without sacrificing the visual quality of the original model had been met with our contribution being able to achieve compression rates of 80%~99% along with low root mean square geometric error of ~ 0.00009 ranges. The last goal on performance and efficiency had also being met with the high utilization of parallel and GPU implementation of the proposed contributions, which allow for a 3 or 4 times increase in performance speed compared to other methods we had benchmarked against.

Although the focus of this thesis was on reality captured models, we wish to highlight our methods remain applicable to a more traditional CAD model as well. This is due to our focus on the various generic model representation like the unstructured point cloud and mesh representation instead of focusing on highly specialized reality captured representation such as range images or structured scans. We can perform the same optimized simplification and compression on the base data representation regardless of whether the data were reality captured or CAD.

7.2 Limitation

Despite the capability of the contributions, there are a few limitations to the methods that will be discussed here.

Currently, the proposed compression of point cloud and the simplification of mesh are only done on the geometric spatial attributes such as position. The integration of additional attributes such as normal or colors into the compression of point cloud and mesh models are of immediate research interest.

Another issue with the simplification methods was the requirement to have the entire raw model loaded onto the memory before the simplification can be done. Although an extra effort was taken when designing these methods to be as memory efficient as possible when faced with the massive data size of reality captured model in the hundreds of GB. This will still pose a significant limitation to the methods.

Another limitation of the current methods was the performance of the methods. As before although the methods were designed to be a highly parallel algorithm, there are several portions of the methods that could be improved by further parallelism or simplification of the processing required.

7.3 Future Works

Previously we discussed the current limitation of the proposed methods, in this section, we will like to point out the future direction of research in this field.

The integration additional attribute such as color or normal information into the compression of point cloud and mesh models are of immediate research interest. Currently, we are only targeting general geometric attribute of the models, however for the models to be truly useful additional attribute is usually required for downstream algorithms. Hence more work in improving this area need to be done.

Research into out-of-core methods can be further conducted and integrated into the current methods. This will unlock the compression capability of our methods to act on the input model of any size, resolving one of our listed limitations of input model requirement to fit within the available memory.

Another direction of interest is the conversion of our parallel algorithms into GPU-based algorithms, since parallel algorithm already fit well within the GPU framework, doing so will allow for even higher performance than our current performance.

REFERENCES

- [1] S. Xiong, J. Zhang, J. Zheng, J. Cai, and L. Liu, "Robust surface reconstruction via dictionary learning," *ACM Trans. Graph.*, vol. 33, no. 6, pp. 1–12, 2014.
- [2] T. Deng, J. Zheng, J. Cai, and T. J. Cham, "Shading-Based Surface Recovery Using Subdivision-Based Representation," *Comput. Graph. Forum*, vol. 38, no. 1, pp. 417–428, Feb. 2019.
- [3] M. Kazhdan and H. Hoppe, "Screened poisson surface reconstruction," *ACM Trans. Graph.*, vol. 32, no. 3, pp. 1–13, 2013.
- [4] M. Li, L. Nan, N. Smith, and P. Wonka, "Computers & Graphics Reconstructing building mass models from UAV images," *Comput. Graph.*, vol. 54, pp. 84–93, 2016.
- [5] P. Mullen, Y. Tong, P. Alliez, and M. Desbrun, "Spectral conformal parameterization," *Eurographics Symp. Geom. Process.*, vol. 27, no. 5, pp. 1487–1494, Jul. 2008.
- [6] Y. Lipman, "Bounded distortion mapping spaces for triangular meshes," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 1–13, 2012.
- [7] A. Sheffer, B. Lévy, M. Mogilnitsky, and A. Bogomyakov, "ABF++: fast and robust angle based flattening," *ACM Trans. Graph.*, vol. 24, no. 2, pp. 311–330, 2005.
- [8] Yuewen Ma, Jianmin Zheng, and Jian Xie, "Foldover-Free Mesh Warping for Constrained Texture Mapping," *IEEE Trans. Vis. Comput. Graph.*, vol. 21, no. 3, pp. 375–388, 2014.
- [9] L. Wang, H. Shum, Lifeng Wang, Sing Bing Kang, R. Szeliski, and Heung-Yeung Shum, "Optimal Texture Map Reconstruction from Multiple Views 1 Introduction 2 Optimal resampling," in *ieeexplore.ieee.org*, 2001, pp. I-347–I-354.
- [10] M. Hess, V. Petrovic, D. Meyer, D. Rissolo, and F. Kuester, "Fusion of multimodal three-dimensional data for comprehensive digital documentation of cultural heritage sites," in *2015 Digital Heritage International Congress, Digital Heritage 2015*, 2015, pp. 595–602.
- [11] SPAR3D, "SPAR3D," 2019. [Online]. Available: <https://www.spar3d.com/>. [Accessed: 17-Jul-2019].
- [12] Artec3D, "Artec3D," 2019. [Online]. Available: <https://www.artec3d.com/>.

[Accessed: 16-Jul-2019].

- [13] Microsoft, "Kinect for Windows," *Kinect for Windows*, 2019. [Online]. Available: <https://developer.microsoft.com/en-us/windows/kinect>. [Accessed: 16-Jul-2019].
- [14] M. Sturzenegger and D. Stead, "Close-range terrestrial digital photogrammetry and terrestrial laser scanning for discontinuity characterization on rock cuts," *Eng. Geol.*, vol. 106, no. 3–4, pp. 163–182, 2009.
- [15] P. J. Besl, "Surfaces in Range Image Understanding," vol. Springer S, 2012.
- [16] L. Wang and S. Weidong, "A review of range image registration methods with accuracy evaluation," in *In 2009 Joint Urban Remote Sensing Event*, 2009, pp. 1–8.
- [17] C. Dold, C. Brenner, and V. Metrology, "Registration of terrestrial laser scanning data using planar patches and image data," *IAPRS*, vol. 36, no. Part 5, pp. 78–83, 2006.
- [18] J. Schlaier, J. Warnat, and A. Brawanski, "Registration accuracy and practicability of laser-directed surface matching," *Comput. Aided Surg.*, vol. 7, no. 5, pp. 284–290, Jan. 2002.
- [19] Q. Capture, "Quantum Capture," 2019. [Online]. Available: <https://www.quantumcapture.com/>. [Accessed: 16-Jul-2019].
- [20] H. Bertil, *Photogrammetry , basic principles and general survey*. 1960.
- [21] J. Treiterer, "Investigation of Traffic Dynamics by Aerial Photogrammetry Techniques," *Rep. (Ohio State Univ. Eng. Exp. Station., 1967*.
- [22] B. Triggs, P. McLauchlan, ... R. H.-... workshop on vision, and U. 1999, "Bundle adjustment—a modern synthesis," *Springer*.
- [23] C. Wu, S. Agarwal, B. Curless, and S. M. Seitz, "Multicore bundle adjustment," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2011, pp. 3057–3064.
- [24] S. A. K. Tareen and Z. Saleem, "A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK," in *2018 International Conference on Computing, Mathematics and Engineering Technologies: Invent, Innovate and Integrate for Socioeconomic Development, iCoMET 2018 - Proceedings*, 2018, vol. 2018-Janua, pp. 1–10.
- [25] A. Baumberg, "Reliable feature matching across widely separated views," *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, vol. Vol 1, pp. 774–781, 2000.
- [26] L. Qiaoliang, W. Guoyou, L. Jianguo, and C. Shaobo, "Robust Scale-Invariant

- Feature Matching for Remote Sensing Image Registration,” *IEEE Geosci. Remote Sens. Lett.*, vol. 6, no. 2, pp. 287–291, 2009.
- [27] O. D. Faugeras, Q.-T. Luong, and S. J. Maybank, “Camera self-calibration: Theory and experiments,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1992, vol. 588 LNCS, pp. 321–334.
- [28] R. I. Hartley, “Estimation of relative camera positions for uncalibrated cameras,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1992, vol. 588 LNCS, pp. 579–587.
- [29] K. H. Petersl and Gabriele, “The structure-from-motion reconstruction pipeline - A survey with focus on short image sequences,” *Kybernetika*, vol. 46, no. 5, pp. 926–937, 2010.
- [30] Y. Wei, L. Kang, B. Yang, and L. Wu, “Applications of structure from motion: a survey,” *J. Zhejiang Univ. Sci. C*, vol. 14, no. 7, pp. 486–494, 2013.
- [31] H. Jin, S. Soatto, and A. J. Yezzi, “Multi-view stereo reconstruction of dense shape and complex appearance,” *International Journal of Computer Vision*, vol. 63, no. 3, pp. 175–189, 01-Jul-2005.
- [32] S. B. Kang, R. Szeliski, and J. Chai, “Handling occlusions in dense multi-view stereo,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 1, pp. 1103–1110, 2001.
- [33] S. M. Seitz, J. Diebel, D. Scharstein, and R. Szeliski, “A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms,” in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2001, pp. 519–528.
- [34] X. Di, Q. Duan, J. Zheng, J. Zhang, J. Cai, and T.-J. Cham, “Shading-Based Surface Detail Recovery Under General Unknown Illumination,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 2, pp. 423–436, 2017.
- [35] C. Häne *et al.*, “3D visual perception for self-driving cars using a multi-camera system: Calibration, mapping, localization, and obstacle detection,” *Image Vis. Comput.*, vol. 68, pp. 14–27, 2017.
- [36] J. Aulinas, Y. Petillot, J. Salvi, and X. Lladó, “The SLAM problem: A survey,” in *Frontiers in Artificial Intelligence and Applications*, 2008, vol. 184, no. 1, pp. 363–371.
- [37] T. Taketomi, H. Uchiyama, and S. Ikeda, “Visual SLAM algorithms: a survey

- from 2010 to 2016,” *IPSJ Trans. Comput. Vis. Appl.*, vol. 9, no. 1, p. 16, Dec. 2017.
- [38] H. Strasdat, J. M. M. Montiel, and A. Davison, “Scale Drift-Aware Large Scale Monocular SLAM,” in *Robotics: Science and Systems VI*, 2010.
 - [39] T. Botterill, S. Mills, and R. Green, “Correcting scale drift by object recognition in single-camera SLAM,” *IEEE Trans. Cybern.*, vol. 43, no. 6, pp. 1767–1780, 2013.
 - [40] G. Vosselman, B. G. H. Gorte, G. Sithole, and T. Rabbani, “Recognising structure in laser scanner point clouds,” in *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*, 2003, vol. 46, no. 11, pp. 33–38.
 - [41] N. J. Mitra, A. Nguyen, and L. Guibas, “Estimating surface normals in noisy point cloud data,” *Int. J. Comput. Geom. Appl.*, vol. 14, no. 04n05, pp. 261–276, 2004.
 - [42] T. Dey, G. Li, and J. Sun, “Normal estimation for point clouds: a comparison study for a Voronoi based method,” in *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics*, 2005, pp. 39–46.
 - [43] P. Boguslawski, “Modelling and analysing 3D building interiors with the dual half-edge data structure,” no. January 2011, pp. 1–20, 2011.
 - [44] E. Danovaro, L. De Floriani, P. Magillo, E. Puppo, D. Sobrero, and N. Sokolovsky, “The half-edge tree: A compact data structure for level-of-detail tetrahedral meshes,” in *Proceedings - International Conference on Shape Modeling and Applications, SMI’05*, 2005, vol. 2005, pp. 334–339.
 - [45] P. S. Heckbert, “Survey of Texture Mapping,” *IEEE Comput. Graph. Appl.*, vol. 6, no. 11, pp. 56–67, 1986.
 - [46] M. Cabrelles, A. E. Seguí, S. Navarro, S. Galcerá, C. Portalés, and J. L. Lerma, “3D Photorealistic modelling of stone monuments by dense image matching,” in *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences. Commission V Symposium*, 2010, vol. Vol. XXXV, pp. 121–124.
 - [47] T. Hanusch, “A new texture mapping algorithm for photorealistic reconstruction of 3D objects:,” *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci.*, vol. 37, no. B5, pp. 699–705, 2008.
 - [48] H. Hoppe, “New quadric metric for simplifying meshes with appearance attributes,” in *Proceedings Visualization’99 (Cat. No. 99CB37067). IEEE*, 1999, pp. 59–510.

- [49] H. Ozaki, F. Kyota, T. K.-EGPGV, and U. 2015, "Out-of-Core Framework for QEM-based Mesh Simplification.," *EGPGV*, pp. 87–96, 2015.
- [50] P. Ratanaworabhan, J. Ke, and M. Burtcher, "Fast lossless compression of scientific floating-point data," in *Data Compression Conference Proceedings*, 2006, pp. 133–142.
- [51] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2674–2683, 2014.
- [52] X. Chen and S. Zhang, "Three dimensional range geometry and texture data compression with space-filling curves," *Opt. Express*, vol. 25, no. 21, p. 26103, 2017.
- [53] Y. Shao, Q. Zhang, G. Li, and Z. Li, "Hybrid Point Cloud Attribute Compression Using Slice-based Layered Structure and Block-based Intra Prediction," *arXiv Prepr.*, vol. arXiv:1804, Apr. 2018.
- [54] R. A. Cohen, D. Tian, and A. Vetro, "Point Cloud Attribute Compression Using 3-D Intra Prediction and Shape-Adaptive Transforms," in *Data Compression Conference Proceedings*, 2016, pp. 141–150.
- [55] P. Rosenthal and L. Linsen, "Image-space point cloud rendering," *Proc. Comput. ...*, no. May 2017, pp. 1–8, 2008.
- [56] J. Peng and C.-C. J. Kuo, "Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition," *ACM Trans. Graph.*, vol. 24, no. 3, p. 609, 2005.
- [57] B. Merry, P. Marais, and J. Gain, "Compression of dense and regular point clouds," in *Computer Graphics Forum*, 2006, vol. 25, no. 4, pp. 709–716.
- [58] Y. Huang, J. Peng, C. C. J. Kuo, and M. Gopi, "A generic scheme for progressive point cloud coding," in *IEEE Transactions on Visualization and Computer Graphics*, 2008, vol. 14, no. 2, pp. 440–453.
- [59] Y. Cai *et al.*, "Madam snake white: A case study on virtual reality continuum applications for singaporean culture and heritage at Haw Par Villa," *Presence Teleoperators Virtual Environ.*, vol. 26, no. 4, pp. 378–388, 2018.
- [60] P. Chiang, Jianmin Zheng, You Yu, Koon Hou Mak, Chee Kong Chui, and Yiyu Cai, "A VR Simulator for Intracardiac Intervention," *IEEE Comput. Graph. Appl.*, vol. 33, no. 1, pp. 44–57, 2012.
- [61] J. Smith, G. Petrova, and S. Schaefer, "Progressive encoding and compression of surfaces generated from point cloud data," in *Computers and Graphics (Pergamon)*, 2012, vol. 36, no. 5, pp. 341–348.

- [62] W. Guan, J. Cai, J. Zhang, and J. Zheng, "Progressive coding and illumination and view dependent transmission of 3-D meshes using R-D Optimization," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 20, no. 4, pp. 575–586, 2010.
- [63] H. Hoppe, "Progressive meshes," *Proc. 23rd Annu. Conf. Comput. Graph. Interact. Tech. ACM*, pp. 99–108, 1996.
- [64] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen, "Decimation of triangle meshes," *ACM SIGGRAPH Comput. Graph.*, vol. 26, no. 2, pp. 65–70, 2005.
- [65] J. Rossignac and P. Borrel, "Multi-resolution 3D approximations for rendering complex scenes," in *Modeling in Computer Graphics*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 455–465.
- [66] M. Garland and P. S. Heckbert, "Surface simplification using quadric error metrics," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques. ACM*, 1997, pp. 209–216.
- [67] P. Lindstrom and G. Turk, "Fast and memory efficient polygonal simplification," in *Proceedings Visualization'98 (Cat. No. 98CB36276). IEEE*, 1998, pp. 279–286,.
- [68] C. DeCoro and T. Natalya, "Real-time mesh simplification using the GPU," *Proceedings of the 2007 symposium on Interactive 3D graphics and games. ACM*, pp. 161–166, 2007.
- [69] A. Papageorgiou and N. Platis, "Triangular mesh simplification on the GPU," *Vis. Comput.*, vol. 31, no. 2, pp. 235–244, 2015.
- [70] N. Grund, E. Derzapf, and M. Guthe, "Instant Level-of-Detail," in *Vision, Modeling, and Visualization (2011)*, 2011, pp. 293–299.
- [71] D. M. Shontz, Suzanne M. and Nistor, "CPU-GPU Algorithms for Triangular Surface Mesh ...," in *Proceedings of the 21st International Meshing Roundtable*, 2013, pp. 475–492.
- [72] T. Odaker, D. Kranzlmüller, and J. Volkert, "GPU-accelerated real-time mesh simplification using parallel half edge collapses," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, vol. 9548, pp. 107–118.
- [73] J. Wu and L. Kobbelt, "Fast Mesh Decimation by Multiple-Choice Techniques," *Vision, Model. Vis.*, pp. 241–248, 2002.
- [74] Y. Azar, A. Broder, A. Karlin, and E. Upfal, "Balanced Allocations," in *Proceedings of the Twenty-sixth ACM Symposium on Theory of Computing*, 1994, vol. 29, no. 1, pp. 593–602.

- [75] B. Vöcking, "Symmetric vs. asymmetric multiple-choice algorithms (invited paper)," in *Proceedings of the 2nd ARACNE workshop*, 2001, pp. 7–15.
- [76] J. Wu and L. Kobbelt, "A stream algorithm for the decimation of massive meshes," *Graph. Interface*, vol. 3, pp. 185–192, 2003.
- [77] M. Garland and P. S. Heckbert, "Simplifying surfaces with color and texture using quadric error metrics," in *ieeexplore.ieee.org*, 2002, pp. 263-269,.
- [78] P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe, "Texture mapping progressive meshes," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM*, 2001, pp. 409–416.
- [79] C. C. Chen and J. H. Chuang, "Texture adaptation for progressive meshes," *Comput. Graph. Forum*, vol. 25, no. 3, pp. 343–350, Sep. 2006.
- [80] H. Yin and C. Berger, "Mastering data complexity for autonomous driving with adaptive point clouds for urban environments," in *IEEE Intelligent Vehicles Symposium, Proceedings*, 2017, pp. 1364–1371.
- [81] M. Donald, "Geometric Modeling Using Octree Encoding," *Comput. Graph. IMAGE Process.*, vol. 19, pp. 129–147, 1982.
- [82] L. Stocco and G. Schrack, "Integer dilation and contraction for quadrees and octrees," in *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing.*, 2002, pp. 426–428.
- [83] G. Schrack, "Finding neighbors of equal size in linear quadrees and octrees in constant time," *CVGIP Image Underst.* 55.3, pp. 221–230, 1992.
- [84] H. Sundar, R. S. Sampath, and G. Biros, "Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel," *SIAM J. Sci. Comput.*, vol. 30, no. 5, pp. 2675–2708, 2008.
- [85] T. Karras, "Maximizing Parallelism in the Construction of BVHs, Octrees, and kd Trees," *Eurographics/ACM SIGGRAPH Symp. High ...*, pp. 1–5, 2012.
- [86] M. Urban, P. Neogrady, V. Kello, and A. J. Sadlej, "ZLIB Spec," *Potentials*, pp. 557–565, 1996.
- [87] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*, vol. 23, no. 4. 2007.
- [88] S. Choi, Q.-Y. Zhou, S. Miller, and V. Koltun, "A Large Dataset of Object Scans," *arXiv Prepr.*, no. arXiv:1602.02481, Feb. 2016.
- [89] T. Hackel, N. Savinov, L. Ladicky, J. D. Wegner, K. Schindler, and M. Pollefeys, "Semantic3D.net: A New Large-Scale Point Cloud Classification Benchmark," in *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial*

Information Sciences, 2017, vol. 4, no. 1W1, pp. 91–98.

- [90] B. Eckart, A. Kelly, H. S. Narasimhan, and J. Kautz, “Compact Generative Models of Point Cloud Data for 3D Perception,” *Dr. Diss. Carnegie Mellon Univ. Pittsburgh*, 2017.
- [91] S. Rousseau and T. Boubekeur, “Fast lossy compression of 3D unit vector sets,” in *SIGGRAPH Asia 2017 Technical Briefs*. ACM, 2017, p. 23.
- [92] N. Coll and T. Paradinas, “Accurate simplification of multi-chart textured models,” *Comput. Graph. Forum*, vol. 29, no. 6, pp. 1842–1853, 2010.
- [93] P. Cignoni, M. Callieri, ... M. C.-E. I., and U. 2008, “Meshlab: an open-source mesh processing tool,” *Eurographics Ital. chapter Conf.*, vol. 2008, pp. 129–136, 2008.
- [94] S. Silva, J. Madeira, B. Sousa, and S. Ieeta, “PolyMeCo — A Polygonal Mesh Comparison Tool,” *Ninth Int. Conf. Inf. Vis. (IV’05)*, IEEE, pp. 842–847, 2005.
- [95] “Autodesk ReCap Photo,” *Autodesk, Inc*, 2019. [Online]. Available: <https://www.autodesk.com/products/recap/overview>. [Accessed: 18-Jun-2019].
- [96] E. Shaffer and M. Garland, “Efficient adaptive simplification of massive meshes,” in *Proceedings of the conference on Visualization’01. IEEE Computer Society*, 2001, pp. 127–134.
- [97] A. Maglo, G. Lavoue, F. Dupont, and C. Hudelot, “3D Mesh Compression: Survey, Comparisons, and Emerging Trends,” *ACM Comput. Surv.*, vol. 47, no. 3, p. 1, 2015.
- [98] D. P. Luebke, “A developer’s survey of polygonal simplification algorithms,” *IEEE Comput. Graph. Appl.*, vol. 21, no. 3, pp. 24–35, 2001.
- [99] Y. Wang and J. Zheng, “Tubular triangular mesh parameterization and applications,” *Comput. Animat. Virtual Worlds*, vol. 21, no. 2, pp. 91–102, Mar. 2010.
- [100] B. Lévy, S. Petitjean, N. Ray, and J. Maillot, “Least squares conformal maps for automatic texture atlas generation,” in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’02*, 2002, p. 362.
- [101] T. Itoh, Y. Yamaguchi, Y. Ikehata, and Y. Kajinaga, “Hierarchical Data Visualization Using a Fast Rectangle-Packing Algorithm,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 3, pp. 302–313, 2004.
- [102] K. Kaul and B. Christian-A, “A Genetic Texture Packing Algorithm on a Graphical Processing Unit,” *Proc. 9th Int. Conf. Comput. Graph. Artif. Intell.*,

2006.

- [103] X. Jia and R. A. Williams, "A packing algorithm for particles of arbitrary shapes," *Powder Technol.*, vol. 120, no. 3, pp. 175–186, 2001.
- [104] F. "ryg" Giesen, "Optimizing the basic rasterizer," *The ryg blog*, 2019. [Online]. Available: <https://fgiesen.wordpress.com/2013/02/10/optimizing-the-basic-rasterizer/>. [Accessed: 18-Jun-2019].
- [105] J. Pineda, "A parallel algorithm for polygon rasterization," *ACM SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 17–20, 2005.
- [106] C. Ericson, *Real-time collision detection*. 2004.
- [107] P. Mazzoleni, F. Matta, E. Zappa, M. A. Sutton, and A. Cigada, "Gaussian pre-filtering for uncertainty minimization in digital image correlation using numerically-designed speckle patterns," *Opt. Lasers Eng.*, vol. 66, pp. 19–33, 2015.
- [108] M. Vinkler, V. Havran, and J. Bittner, "Bounding volume hierarchies versus kd-trees on contemporary many-core architectures," in *Proceedings of the 30th Spring Conference on Computer Graphics*. ACM, 2014, pp. 29–36.
- [109] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques - SIGGRAPH '02*, 2002, p. 703.
- [110] J. L. Schönberger, E. Zheng, J. M. Frahm, and M. Pollefeys, "Pixelwise view selection for unstructured multi-view stereo," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, vol. 9907 LNCS, pp. 501–518.
- [111] C. Sweeney, T. Hollerer, M. T.-P. of the 23rd ACM, and U. 2015, "Theia: A fast and scalable structure-from-motion library," *Proc. 23rd ACM Int. Conf. Multimedia*. ACM, pp. 693–696, 2015.
- [112] P. Moulon, P. Monasse, R. Perrot, and R. Marlet, "OpenMVG: Open multiple view geometry," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, vol. 10214 LNCS, pp. 60–74.