Multi-scale simplification and visualization of large 3D models

Rui Li

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

JJL

SORBONNE UNIVERSITÉ

LJLL

Doctoral School **École Doctorale Sciences Mathématiques de Paris Centre**

University Department **Laboratoire Jacques-Louis Lions**

Thesis defended by **Li RUI**

Defended on **February 22, 2024**

In order to become Doctor from Sorbonne Université

Academic Field **Mathématiques appliquées**

Speciality **Analyse numérique**

Multi-scale simplification and visualization of large 3D models

Thesis supervised by	Didier SMETS	Supervisor
	Philippe WALTER	Co-Supervisor

Committee members

Referees	Michael GUTHE	Professor at University of Bayreuth
	Eric GUÉRIN	HDR Associate Professor at Institut National des Sciences Appliquées de Lyon
Examiners	Nicolas MELLADO	Reasearcher at Institut de Recherche en Informatique de Toulouse
	Pascal FREY	Professor at Sorbonne Université
Guest	Vincent LECLERCQ	Lead Programmer at Ubisoft Paris
Supervisors	Didier SMETS	Professor at Sorbonne Université
	Philippe WALTER	Professor at Sorbonne Université

Colophon

Doctoral dissertation entitled "Multi-scale simplification and visualization of large 3D models", written by Li Rui, completed on May 22, 2024, typeset with the document preparation system LaTeX and the yathesis class dedicated to theses prepared in France.

Keywords: mesh simplification, multi-resolution meshes, 3d reconstruction and visualization, real-time rendering

Mots clés : simplification de maillage, maillages multi-résolution, reconstruction et visualisation 3d, rendu temps réel

This thesis has been prepared at

Laboratoire Jacques-Louis Lions
Sorbonne Université
Campus Pierre et Marie Curie
4 place Jussieu
75005 Paris
France

☎ +33 1 44 27 42 98
Web Site `https://ljll.math.upmc.fr/`

To my family
献给我的家人们

To my dear friends
À mes chers amis
献给我可爱的朋友们

Acknowledgements

Remerciements

致谢

First and foremost, I would like to extend my thanks to Eric Guérin and Michael Guthe for serving as reviewers for my thesis. It has been an honor to have the very experts whose articles I read at the beginning of my PhD be the ones to review my thesis. Thank you both for the careful reviews and the kind words.

I would also like to thank Pascal Frey, Nicolas Mellado and Vincent Leclercq for serving as members of my jury and taking time out of their busy schedules to come listen to my presentation.

This thesis would not exist without Didier Smets and Philippe Walter. Their support has not only made this thesis possible but has also provided me with the opportunity to come to Paris, enjoy life, and engage in research in this extraordinary city. Philippe, thanks for sharing extraordinary archaeological 3D models generously. To Didier, thanks for not only all the patience, guidance, and encouragement these last three years. I can't wait until the next time we discuss algorithm and codes in front of your blackboard or computer screen [1]. At the same time, thanks for your help [2] and advice [3] in adapting and settling into my life in Paris.

Je souhaite aussi remercier Catherine, Malika, Salima et Corentin de si bien s'occuper de nous. Je tiens également à remercier Nora de l'ISCD pour m'avoir patiemment aidé avec tout mon ordre de mission.

I would also like to thank the PhD students I met at LJLL. Thank you for being there for the past three years, whether it is Crous 11h45 team, LabTea, GTT, Secret Santa events, or our film club.

It brings me great joy to share the same office with Allen, Alexandra, Anatole, Aloïs, Dawei, Fatima, Lucas, Roxane, Ramon, Matthew, Gotran, Nicolas, Nga and Siguang. Anatole, the magician of LJLL, thanks for assisting in calling Ameli and invite me to your party! Matthew, thanks for lending me your bicycle in Arcachon. Nga, our conversations seem ceaseless, even when the other PhDs in our office march for a strike, we find ourselves engaged in day-long discussions (our own form of strike). Also, thanks for sharing K-pop knowledge! Alexandra, it was a pleasure discussing films with you over lunch. Siguang (齐思广), thanks for helping me solve the posture of the meteorite in outer space. Roxane, merci d'avoir supporté mon français cassé et d'avoir révisé ma lettre de motivation et mon résumé de thèse rédigés en français. C'est un plaisir de regarder des films avec toi (que ce soit dans un cinéma parisien ou au Festival de Cannes) et de me faire découvrir de bons films. Merci de m'avoir donné les livres [4] , je ferai de mon mieux pour les terminer. Les crêpes que tu as faites sont délicieuses aussi ! J'espère lire ton

[1] Try to help me find bugs in my code.
[2] Help me apply for Cité Universitaire and other administrative stuff.
[3] Velib and other useful tips for living in Paris.
[4] Les bandes dessinées d'Agnès Varda et Les Cahiers du cinéma.

roman bientôt ! Allen (方君陶), it is a pleasure to meet you here and chatting with you every day in the office. Thank you for always helping me check for typos in my articles and my thesis [5], cooking lots of meals for me, and taking me to nice restaurants in Paris. 希望你可以继续努力, 多学会一些菜做给我吃。

For PhDs in other offices: Chourouk, thanks for always patiently answering my tons of questions and giving me ideas. It has been a pleasure managing LabTea with you! Thanks for cooking me Lebanese food and teaching me how to drink mate tea [6]. Augustin, thanks for helping me call the Mexican Embassy, and the hiking training in Arcachon [7]. Maria, thanks for inviting me to the films and taking me to Shodai Matcha for dessert. I enjoy the interesting chat after each film. Thanks for the encouragements when I was developing my little game! Lise, it was really fun to make dumplings with you, and share chicken feet in Sucrepice. Let's be fake "weeb" together! Thanks for creating the cinema club in the first place, it is interesting to talk about all different cultures and exhanging funny videos with you. May I see Zombra soon! Elane, thanks for tips for growing tulips. Thomas, thanks for the discussions on development of my little game. Ludovic, thanks for helping me arrange my pré-soutenance and BigBlueButton links. Elanor, thanks for taking over the LabTea, and helping me to check my defense room. Ioanna, it was fun walking with you on the beach at La Grande-Motte and discussing how to develop photos. Lucas, Charles, Robin, Rémi, Annamaria, Lucia, Federica, Nicolaï, Jules, Pierre, Juliette, Christobal, Kala, Sebastian and Yvonne, It was fun to chat with you and participate in various activities in the laboratory.

感谢实验室的中国小队：张明月，陈良英，冯悦， 吕泽钰，刘海波，卢柳迪，牛景瑞，陈恭，王艺朋，戴天瑞，戴睿阳，陈哲，梁睿康，齐思广，马翔宇，方君陶， 赵薄熙。很开心和你们在巴黎吃吃喝喝，谈天说地。

Mingyue Zhang (张明月), thanks for responding to all my requests, helping me move even though you were suffering with cervical spondylosis. It was really fun to take care of Mina [8]. Thanks for listening to my endless chatter in the coffee room. Liangying Chen (陈良英), thanks for influencing me with your brilliance as a scholar and making me less lazy. I hope we can have more than one drink together in the future [9]. Yue Feng (冯悦), It was fun to plan trips with you [10]. Zeyu Lyu (吕泽钰), thanks for sharing delicious matcha cake with us. Haibo Liu (刘海波), your fried lamb with green onions is tasty, and it was a pleasure making dumplings with you! Liudi Lu (卢柳迪), thanks for answering all my questions patiently and telling me a lot of tips 小妙招. Jingrui Niu (牛景瑞), thanks for helping me move, change the refrigerator and ordering dumplings together (饺友). Gong Chen (陈恭), thanks for inviting us to lunch. Yipeng Wang (王艺朋), thanks for introducing me to other Chinese PhDs in the lab when I first came. Tianrui Dai (戴天瑞), thanks for helping me move my refrigerator and show us how to Just Dance. Ruiyang Dai (戴睿阳), thanks for the panettone! Ruikang Liang (梁睿康), thanks for managing LabTea and sharing lucky cookies with us. Zhe Chen (陈哲), do not get drunk alone! Xiangyu Ma (马翔宇), thanks for the Kung Pao Chicken. Boxi Zhao (赵薄熙), it was nice to meet you at the PhD-master meeting.

Diqi Xu (许迪淇), thanks for helping me since I first came to Paris six years ago, always cutting my bangs and being my photographer. I really miss eating rice noodles (螺蛳粉) and watching soap operas with you! Yingsai Zhou (周英赛), thanks for driving us to Netherlands. The whole trip was full of fun! Man Zhou (周满) and Christophe Sun, thanks for feeding me

[5] 没想到可以让普林斯顿学子帮我改英语。
[6] Even though mate is a bit bitter to me.
[7] I feel like my hiking ability has evolved.
[8] Even though she bit me.
[9] Do not start swinging after your first drink.
[10] Even if they only exist in our minds.

with various foods[11]. Chuan Jiao (焦川), the trip to the Alps was truly breathtaking. Jiaojiao Yao (姚娇娇), love every one of your jokes! Zihua Yang (杨子华), it was nice chatting with you.

Thanks to the gang I met at Cité Universitaire when I first came to Paris! It was a lot of fun cooking, making dumplings with you [12], and playing Switch.

In the last few months of my PhD, I had the pleasure of participating in the Development at Ubisoft project. Vincent, thanks for teaching me a lot from industry perspectives and give me many advice about my code [13]. Estelle, thanks for inviting me to Le Nouvel Institut for a drink! It was fun to chat over drinks with you, Maxine, and Emilie.

Merci à tous les professeurs et professeures de français de la Sorbonne. Mehdi, merci pour votre enseignement strict. André, merci d'avoir organisé des activités comme manger des Flammekueches et jouer Exploding Kittens, j'aime Apfelsaftschorle beaucoup ! Pauline, j'apprécie vos cours de français, qu'il s'agisse d'un cours de cinéma ou d'un cours de préparation au DELF. Votre passion est toujours inspirante ! Margaux, merci pour votre petite histoire intéressante.

At the same time, I would like to thank the friends I met during the Cannes Film Festival, Berlin Film Festival and Venice Film Festival. It was a pleasure to discuss films, actors and directors with you. Xinyu Zhou (周欣渝), thanks for letting me stay at your house when I was in London, your skylights are my favorite! It was fun to draw together on Agnes Varda's poster [14]! I hope we can go to film festivals together in the future!

To the members of Mexico Summer School, the summer school was an eye-opening experience for me. The week we spent studying at the Museo Nacional de Arte de México benefited me a lot. It was fun to taste Mexican food and tequila with you!

A shout-out and thank you to all my friends in China. Thanks for always being there for me. Qiuyi Li (李秋仪), thanks for polishing my papers for me since I was pursuing my master, for putting up with many of my grammatical errors. And the "assemble!" (集合！) message you sent to start the group phone chat. It was a pleasure chatting with you on the phone for a few hours. Yuqing He (贺雨晴), cat savior and animal lover, we can chat for thousands of lines on Wechat, sometimes we are just competing to see who can re-send a meme to the end. It was fun to write you letters to Japan and receive postcards from you. Xiaojuan Zhu (朱晓娟), a die-hard fan of the law of symmetry in photography, thank you for being the coordinator in the group every time, when the arguing of three of us is out of control. Hope we can travel together in the future [15]. Shan Jiang (姜珊), thanks for always being there no matter what I say, sharing stupid memes and jokes with me, sending wake-up messages in the morning everyday [16]. I hope we can jump around in the livehouse together one day [17]. Siyu Zhou (周思羽), thanks for chatting on and off for three years, and your Disneyland employee ticket (even though I haven't gone yet). Mengyi Mei (梅梦怡) , thanks for giving me so many recommendations when I went to Korea, I loved the food! Jing You (由静) , thanks for your greetings, and glad you like the photos I sent you!

Finally, thank you to my family, who, though most of the time were an ocean away, have continued to support me through every twist and turn. 感谢我的母亲和父亲，给予我物质与精神上最大程度的自由。同时也是我最好的伙伴，倾听我在学业和生活上的点点滴滴，给予我关心与支持。感谢大姨，从小到大对我无微不至的关爱。 感谢的我妹妹，尽管跨越大洋的遥远距离使我们无法分享同一片天空，感谢你对我的倾听与支持。

Three years of PhD life is like a dream, thank you all for being a part of it!

[11] Leave food at my door.
[12] Our goal is to make hundreds of dumplings at a time!
[13] Remember your code should be good prose.
[14] Un jour sans voir un arbre est un jour foutu.
[15] 一起吵吵闹闹看这花花世界。
[16] Of course never woke me up.
[17] 看你mosh！

Multi-scale simplification and visualization of large 3D models

Abstract

This thesis focuses on the creation and interactive visualization of surface meshes derived from massive data, such as those obtained from the use of laser remote sensing scanners or multiple-view image captures. Recent advancements in the development, miniaturization, and widespread accessibility of these capture devices aim to provide a detailed representation of reality.

The raw data resulting from these captures, whether in geographic information, archaeology, or medicine, often pose challenges due to their voluminous and disorganized nature. To address this challenge, the first step in a processing pipeline typically involves the creation of a high-resolution mesh. The chosen solution must effectively mitigate measurement noise and accurately reproduce the topology of the target object.

As capture devices have advanced in resolution, visualization hardware has improved in memory capacity and processing speed. However, it has become impractical to interactively visualize high-resolution meshes, containing tens of millions or even billions of elements. Handling such data flow by hardware and transferring it between pipeline components present significant challenges.

The subsequent step involves constructing a hierarchy of simplified meshes derived from the high-resolution mesh, with each simplification introducing controlled detail loss. While it might appear that processing capacity comes at the expense of detail, for visualization purposes, only the apparent level of detail matters. Portions distant from the point of observation require a lower absolute level of detail than those nearby. This enables the subdivision of different mesh levels into subparts, dynamically selecting and recombining them to adapt to changes in viewpoint, ensuring an optimal apparent level of detail.

In the first part of the thesis, we introduce an approach aimed at addressing the two main challenges inherent in the simultaneous management of multiple pieces of meshes with different resolutions: mesh junction issues (known as "mesh cracks") and temporal discontinuity problems resulting from resolution changes (referred to as "LOD popping"). This approach, situated at the intersection of techniques based on mesh morphing and those rooted in multi-triangulations, continues a long lineage of research in these areas. We accompany it with both a reference implementation and a lightweight visualization prototype serving as a proof of concept. The preprocessing phase, responsible for constructing the hierarchy from the high-resolution mesh, stands out with a processing capacity on the order of one million triangles per second per processing core. This represents a substantial order of magnitude improvement compared to the existing solutions we have had the opportunity to evaluate.

In the second part of the thesis, we dig into the construction of high-resolution meshes from a point cloud, specifically in the case of airborne LiDAR. We demonstrate that the reliability of conventional methods for normal estimation can be significantly enhanced by taking into account data specific to such surveys, including point timestamps and laser beam angles. This improvement has a crucial impact on the quality of subsequent surface mesh reconstruction using the Poisson method. The Poisson method derives the mesh as the level surface of a scalar function, itself obtained by solving a Poisson problem in which the oriented point cloud contributes to defining the source term. We apply these methods to recent open LiDAR data from Swiss (SwissSurface3D) and French (Lidar HD) LiDAR programs, demonstrating that they enable a more faithful reproduction compared to classical methods based on regular altitude grids (digital elevation models), particularly in mountainous areas.

Keywords: mesh simplification, multi-resolution meshes, 3d reconstruction and visualization, real-time rendering

Laboratoire Jacques-Louis Lions
Sorbonne Université – Campus Pierre et Marie Curie – 4 place Jussieu – 75005 Paris – France

Résumé

Cette thèse s'intéresse à la création et à la visualisation interactive de maillages surfaciques issus de données massives, telles que celles obtenues suite à l'emploi de scanners à télédétection laser ou de prises de vues multiples. Ces dernieres années ont en effet vu l'accélération du développement, de la miniaturisation et de la démocratisation de ces dispositifs de capture visant une représentation fine du réel.

Les données brutes issues des captures, par exemple en géographie, archéologie ou en médecine sont difficiles à exploiter car volumineuses et désordonnées. Pour faire face à ce défi, la première étape d'une chaîne de traitement consiste classiquement en la création d'un maillage très haute résolution. La solution retenue doit palier au mieux les bruits de mesure et reproduire la topologie de l'objet étudié.

L'augmentation de la résolution des dispositifs de capture, des capacités de mémoire et de traitement du matériel de visualisation rend souvent impraticable la visualisation interactive des maillages haute résolution. Ces maillages de dizaines de millions voire plusieurs milliards d'éléments sont un défi pour le matériel et pour le transfert de données entre les composants de la chaîne.

L'étape suivante implique la création d'une hiérarchie de maillages simplifiés dérivant de l'original, les pertes de détail à chaque simplification étant contrôlées. Bien que cette approche semble contourner le probleme au lieu de le résoudre, seul le niveau de détail importe en visualisation. Ainsi, il devient possible de découper les niveaux de maillage en sous-parties, les sélectionner et les recombiner à la volée pour suivre les mouvements du point d'observation en optimisant partout le niveau de détail apparent.

Dans la première partie de la thèse, nous proposons une approche qui s'attache à résoudre les deux difficultés principales inhérentes à la gestion simultanée de plusieurs morceaux de maillages de résolutions différentes : les défauts de jointures ("mesh cracks"), et les problèmes de discontinuité temporelle suite aux changements de résolutions ("LOD popping"). Cette approche, dans la longue lignée de travaux ayant abordé ces questions, s'inscrit au carrefour entre les techniques basées sur le morphing entre maillages et celles basées sur les multi-triangulations. Nous l'accompagnons d'une implémentation de référence, ainsi que d'un prototype de visualiseur léger servant de preuve de concept. La partie pré-traitement, qui construit la hiérarchie à partir du maillage haute résolution, se distingue par une capacité de l'ordre du million de triangles par seconde et par coeur de calcul, ce qui représente un ordre de grandeur de gain par rapports aux solutions existantes que nous avons pu tester.

Dans la seconde partie de la thèse, nous abordons la question de la construction du maillage haute résolution à partir d'un nuage de points dans le cas spécifique du Lidar aéroporté. Nous montrons que la fiabilité des méthodes classiques d'estimation de normales peut être nettement améliorée en tenant compte de données propres à de tels relevés : horodatage des points et angle du faisceau laser notamment. Cette amélioration a une incidence cruciale sur la qualité de la reconstruction ultérieure d'un maillage surfacique par la méthode dite de Poisson. Cette dernière obtient le maillage comme surface de niveau d'une fonction scalaire, elle-même obtenue par la résolution d'un problème de Poisson pour lequel le nuage de points orientés intervient dans la définition du terme source. Nous appliquons ces méthodes aux données libres récentes des programmes Lidar suisse (SwissSurface3D) et français (Lidar HD), et montrons qu'elles permettent une reproduction plus fidèle que les méthodes classiques basées sur les grilles à pas régulier (modèle numériques de terrain), d'autant plus dans les zones montagneuses.

Mots clés : simplification de maillage, maillages multi-résolution, reconstruction et visualisation 3d, rendu temps réel

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 3D data acquisition

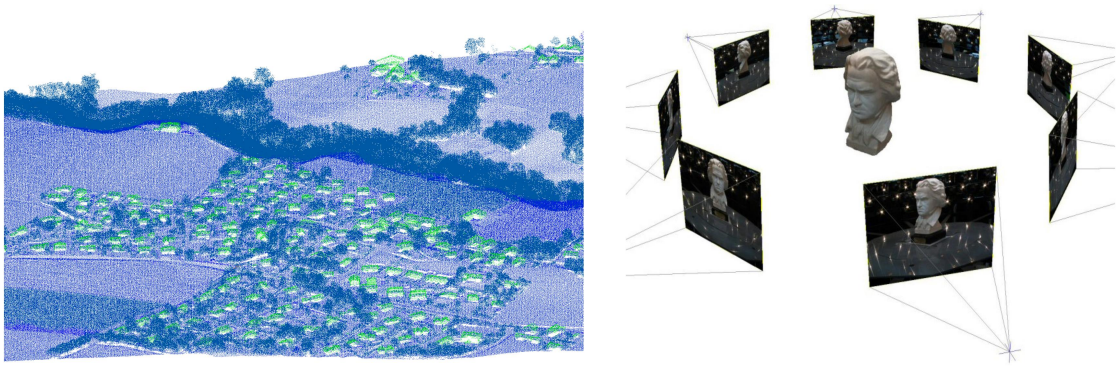
Three-dimensional (3D) data refers to information that represents objects or scenes in 3D space, capturing their spatial attributes and geometry. In recent years, the storage size and spatial volume of 3D data have experienced remarkable growth, primarily driven by the increasing accessibility of affordable 3D sensors, coupled with the proliferation of data generation and utilization across a diverse array of industries and applications.

The 3D sensor market has undergone rapid development in recent years [1]. Several common types of 3D sensors are widely used, in particular Light Detection And Ranging (LiDAR) and depth-sensing cameras, allowing for the capture of 3D data from the real world with unprecedented levels of resolution. They have been widely adopted in various applications, including land monitoring, archaeology, robotics, automotive, gaming, virtual reality (VR), and augmented reality (AR). They are also installed on consumer devices such as smartphones [2]. Two common forms of 3D data generated by these sensors are point clouds (Figure 1.1 (a)) and multi-view photographs (Figure 1.1 (b)).

The rapid advancement of 3D sensors has significantly expanded their applicability across various fields, further accelerating the generation of 3D data. In the automotive industry, 3D data plays a critical role in autonomous driving in terms of perception and navigation, which is fulfilled by LiDAR systems generating vast quantities of point cloud data. Within the game industry, there is a high demand for visually stunning and interactive media. This demand for immersive experiences has been a driving force behind the growth of 3D content creation. Moreover, in the realm of Geographic Information Systems (GIS) and geospatial mapping, 3D sensors (such as airborne LiDAR) are commonly used to capture high-resolution elevation data

[1] https://www.fortunebusinessinsights.com/industry-reports/3d-sensors-market-100104

[2] Some smartphone manufacturers have integrated LiDAR technology into their smartphones.



(a) Point cloud

(b) Multi-view photograph

Figure 1.1: 3D data representation: (a) Screenshot of using CloudCompare [Clo23] to visualize point cloud data from swisstopo [Top23b]. (b) Taken from [Osw15].

of the Earth's surface. It is crucial for creating accurate digital elevation models (DEMs) and 3D terrain models. Furthermore, cultural heritage and archaeology research have harnessed 3D scanning and modeling techniques to preserve and study historical artifacts, archaeological sites, and cultural heritage in 3D.

1.1.2 3D data processing

3D data offers a wealth of information in terms of the geometry of objects and scenes. Once 3D data is acquired, it necessitates a series of processing and transformation steps to meet the requirements of specific applications. These steps can be divided into the following categories [BW10]:

Data filtering. The raw 3D data often contains noise data or invalid data. By removing noise, outliers, or irrelevant data, the overall quality of 3D data can be improved.

Feature detection. The relevant features or attributes from the 3D data, such as edges, keypoints, or surface normals can be identified and extracted. These extracted features are valuable for a wide range of tasks and analyses, including object recognition, segmentation, registration, and surface reconstruction.

Data Registration. 3D data can be captured from various viewpoints to obtain a comprehensive representation of an object or scene. Aligning and transforming multiple 3D scans or multi-view photographs is essential to ensure they share the same coordinate system, allowing for a coherent and accurate composite view.

Data segmentation. The objective of segmentation is to separate different parts or objects within the 3D data. In the context of geospatial data, segmentation can involve dividing the data into distinct regions or categories, such as terrain, buildings, vegetation, and water bodies.

Surface reconstruction. Creating a continuous and smooth surface representation from point clouds or multi-view photography is fundamental for many tasks. Surface reconstruction from

point cloud sets involves converting a set of individual points in 3D space into a mesh representation, which consists of vertices, edges, and faces defining the surface of an object or scene. The triangle mesh model can be generated by using techniques such as Poisson reconstruction (see Figure 1.2) or ball-pivoting algorithm [Ber+99]. Once surface reconstructions from the point cloud are completed, the resulting 3D mesh may undergo refinement to improve its quality, remove artifacts, and optimize for real-time rendering or other specific requirements.

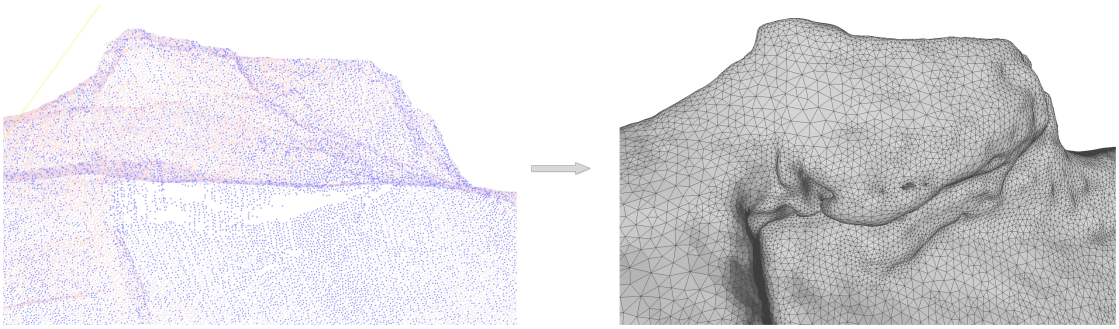


Figure 1.2: Poisson surface reconstruction [KBH06]. Left: points sampled on the overhang landform terrain in the Säntis region of Switzerland using airborne LiDAR [Top23b]. Right: reconstructed surface mesh generated by our pipeline which will be introduced in Chapter 5.

1.1.3 3D data visualization

3D data visualization plays a crucial role in various fields and has important applications in 3D data processing. However, the proliferation of 3D data also presents significant challenges when it comes to visualizing large-scale 3D models. While both point clouds and surface models, especially surface mesh models, can be employed for visualization, meshes are often preferable due to their efficiency in representing intricate geometry and their compatibility with a wide range of rendering techniques.

Rasterization and ray tracing are two commonly used techniques for rendering images of 3D scenes in computer graphics. Rasterization is a real-time rendering technique that uses an object-oriented graphics pipeline to generate real-time rendering results, but it can be challenging to achieve photo-realistic results. Ray tracing is a rendering technique that can generate realistic images by simulating the behavior of light in a scene. However, the intensive computation makes it challenging to achieve the same real-time performance as rasterization.

Concurrently, the evolution of both general-purpose computing processors and modern graphics processors has been enhancing the rendering capabilities of massive 3D models. The expansion in computing power has enabled users to create increasingly complex datasets. However, memory bandwidth and data access speed have not grown at the same accelerated rates as processing power. This processor-memory bottleneck problem is also called a memory wall problem [ECT17]. As Figure 1.3 shows, processors have been improving in terms of clock speed and

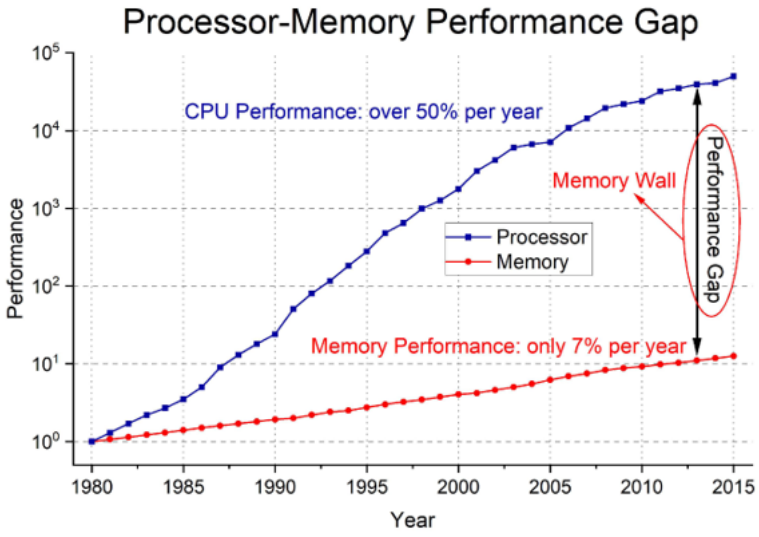


Figure 1.3: Processor-memory performance gap, image taken from [YKC23].

computational power at a much faster rate than Dynamic Random-Access Memory (DRAM) memory has been improving in terms of access speed. As a result, data accessing has become a significant bottleneck rather than computation when handling massive datasets. Consequently, the massive data cannot be efficiently rendered through brute force methods. To address this problem, many researchers developed different rendering technologies, such as data reduction techniques, cache-coherent layout methods, and data compression [GKY08].

1.2 Challenges

This thesis primarily addresses two aspects that have been previously mentioned:

- The reconstruction of terrain surface models from airborne LiDAR point clouds.
- The construction and optimization of continuous Level of Detail (LOD) multi-resolution meshes designed for real-time visualization.

3D reconstruction based on terrain airborne LiDAR point clouds may encounter the following obstacles. First, airborne LiDAR systems can generate a massive amount of point cloud data, especially when large areas need to be covered. The management and processing of such voluminous datasets can be computationally intensive and time-consuming. For example, swissSURFACE^{3D} [Top23b] represents all the natural and built features of the surface of Switzerland as classified point clouds, it contains several millions of points per km^2. Moreover, real-world terrain surfaces exhibit intricate characteristics, including cliffs, overhangs, caves, and dense vegetation. In this case, there sure are substantial difficulties in achieving more accurate reconstruction to accommodate such diverse landforms of terrains.

Real-time visualization of large-scale 3D surface models represents an ongoing challenge, due to the data access speed limitations. One possible way to solve this problem is to rely on simplified models combined with texture maps (normal, bump, displacement, etc.) to enhance apparent detail. This method has been widely used in the industry so far. However, achieving the same level of detail as high-poly models using this method can be a demanding task. Another solution is to generate a multi-resolution model that includes different levels of detail (LODs) and supports view-dependent rendering through mesh refinement or coarsening processes. Nonetheless, it can be difficult to dynamically adapt the resolution without introducing rendering artifacts, such as popping [3] during LOD switching or cracks at the boundary between different LOD parts, due to varying LOD resolutions.

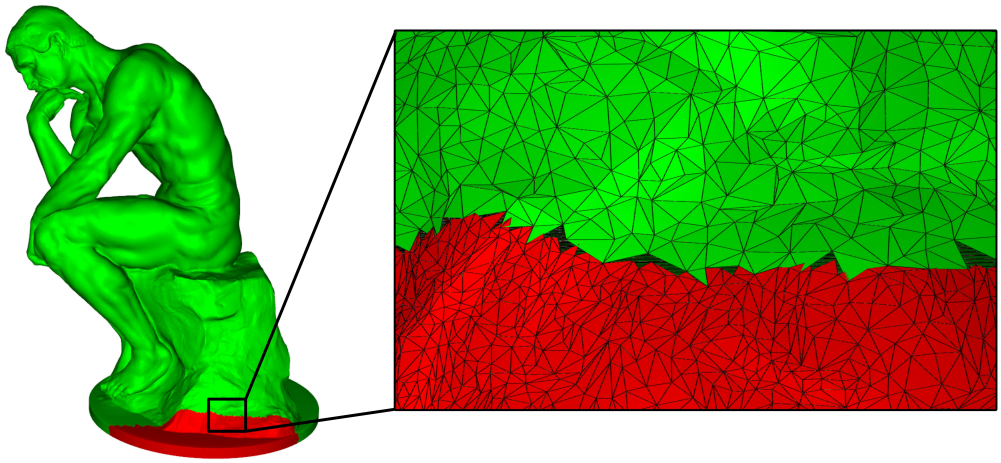


Figure 1.4: Mesh cracks between different LOD parts. Left: Rendering results of the thinker [Rod14] by colorizing the different LODs, the red LOD with higher resolution, and the green LOD with lower resolution. Right: Partial enlargement of mesh crack.

As a thoroughly researched field, many, many frameworks have been proposed to solve this problem. There is a growing demand for more streamlined algorithms that can be readily and seamlessly integrated. Especially for a computer with a basic configuration, massive 3D models can often not be rendered in standard 3D viewers due to their size and complexity. While some high-end rendering engines propose solutions to this problem, they require a steep learning curve, high processing power, and advance hardware devices.

1.3 Contribution

Focusing on real-time rasterization of large-scale 3D surface models and real-world terrain model reconstruction from airborne LiDAR point cloud, the thesis has the following contributions:

[3] https://youtu.be/KfeFcZDjCRg?si=RA1xTzmlnRURpWcp

(1) A multi-resolution construction method allows for generating hierarchy representation with different LODs. A key property is tracking a child-parent dependency between vertices of successive LODs. The child-parent relation between vertices of successive hierarchy LODs is designed to be compatible with parallel preprocessing. The implementation of multi-resolution construction achieves competitive levels of the order of 1 million input triangles per second per core on a mainstream computer.

(2) A view-dependent rendering algorithm achieves the interactive visualization of a 3D large-scale model without boundary cracks and LOD popping, which is combined with suitable constraints in the construction stage, and based on the vertex interpolation between the LODs to guarantee that the actual vertex that is visualized depends continuously on the viewpoint position. The implementation of the viewer could render scanned models of multiples tens of million triangles at optimal visual quality and interactive frame rate on a computer with integrated GPU or discrete GPU.

(3) A pipeline for terrain surface reconstruction from airborne LiDAR point cloud is proposed. This process allows the preservation of landform features, such as overhangs or cliffs, which can maintain more realistic details compared to regular grid terrain models. By capturing the geometrical detail of the terrain, some mountain models can achieve highly realistic rendering results even without the utilization of texture maps.

1.4 Thesis overview

The rendering framework proposed in this thesis is based on the multi-resolution model method and belongs to the rasterization category. The thesis includes 6 chapters, chapter 2 introduces the basic concept of 3D triangle mesh, the quick review of 3D object representations, mesh simplification, multi-resolution model structure, and rendering techniques. Chapter 3 focuses on the construction process of the multi-resolution model and the implementation details. Chapter 4 presents a rendering pipeline that can avoid LOD-popping and boundary mesh cracks, along with its implementation details. Chapter 5 introduces the terrain model generation algorithm based on the DEM or LiDAR point cloud, showcases the adaptation and application of our visualization framework to the large terrain models, and provides an in-depth discussion of design choices, implementation details, and experimental results. Chapter 6 concludes the thesis and outlines future work for the presented framework.

Chapter 2

Terminology

2.1 3D Object representations

3D object representations are used to model the shape, appearance, and geometry of objects in a 3D space. These representations play an important role in computer graphics, computer-aided geometric design, visualization, and robotics [Fun03]. 3D object representations can be divided into 3 primary categories: raw data, surfaces, and volumetric representation.

2.1.1 Raw data

The raw data can be point cloud, range image, and polygon soup. Among them, the point cloud is widely used in autonomous driving and Virtual reality (VR). It is commonly obtained by LiDAR, photogrammetry, and depth sensors, which provide simple yet efficient 3D data representations. A point cloud is a discrete set of data points in the 3D coordinates system, and it can represent a 3D shape or object in 3D space. Each point position is represented by coordinates (x, y, z). The explicit representation allows us to achieve fast transformation and rendering, however, an enormous number of points can result in substantial memory consumption. A common process is to convert this data into a surface representation using reconstruction techniques.

The range image is a 2D image where each pixel represents the distance from the sensor to the surface of the object at that pixel location. Range images are typically obtained using depth cameras or LiDAR sensors, which can be used for rendering. Although there are challenges, the range image is often noisy and incomplete, it has limited field of view and spatial resolution, which can result in a loss of detail and context. In order to solve these problems, range images can be combined with other 3D representations, such as point clouds, to provide additional geometric and texture information.

Polygon soup is an unstructured set of polygons, typically triangles, before any structuring operation. They can be of different sizes and shapes and not connect to each other, which can be used in game engines, simulation, and virtual reality applications, where real-time rendering

is required. Because polygon soup can be easily rendered using modern graphics hardware, such as GPUs, and it can be easily modified and updated, allowing for dynamic changes to the shape and appearance of objects in the scene.

2.1.2 Surface representation

The representation of surfaces can take various forms, including mesh, subdivision surfaces, parametric surfaces, and implicit surfaces. Among these options, a mesh is a frequently used choice. It consists of vertices, faces, and edges, providing a discrete representation of the surface. Meshes offer the advantage of presenting sharp edges and corners in their representation. They are easily processed by modern graphics hardware and compatible with a wide array of rendering techniques, including texture mapping and normal mapping.

A subdivision surface is a curved surface represented by the specification of a coarser polygon mesh and produced by a recursive algorithmic method, which is often used for organic and complex shapes that require a high level of detail and precision. It is a widely used modeling tool for computer gaming and animation [ZOR00]. However, it can be computationally expensive to render due to the necessity of subdividing the control mesh into smaller meshes.

A parametric surface is a surface in the Euclidean space $\mathbb{R}^3$, which is defined by a parametric function with two parameters. Parametric surfaces are often used for objects such as cars or human faces that require high geometry details. They can be computationally intensive to render due to their complex mathematical functions.

An implicit surface is a surface defined in Euclidean space by $F(x, y, z) = 0$. There are different representations for implicit functions, such as algebraic surfaces, radial basis functions, and discrete voxelizations [Bot+10]. Implicit surfaces offer a convenient representation for complex shapes and allow seamless modification through mathematical operations like scaling or rotation. It is often used for objects such as clouds or fire that do not have a well-defined surface geometry. It can be computationally expensive to render due to evaluating the implicit function at every point in space.

2.1.3 Volumetric representation

Volumetric data represents 3D scenes as a volume of data, rather than a surface mesh. It is commonly used to visualize and simulate natural phenomena such as smoke, fire, clouds, and other forms of atmospheric effects. Voxel is a term used for the discrete volumetric samples that form a voxel-based model. Unlike pixels, which are 2D squares, voxels are 3D cubes that make up a uniform grid. A voxel-based 3D model can be thought of as an assembly of 3D pixels, with each voxel representing a point in 3D space. The current rendering pipeline is optimized for rendering polygons, and there is no pipeline to render high-resolution voxels efficiently. However, its structured layout of a voxel grid can be implemented for classification tasks through a 3D convolutional neural network [Ioa+17]. By processing voxels in a similar way to 2D convo-

lutional neural networks processing 2D images, a 3D convolutional neural network can extract meaningful features from the voxel-based data and use them for classification or other tasks. This makes voxel-based data an important tool for applications such as medical imaging, scientific simulations, virtual reality, and gaming.

2.2 Mesh

2.2.1 Definition

Triangle meshes are one of the most commonly used methods for representing 3D objects in computer graphics. A triangle mesh M is composed of both geometric and topological elements. The geometric elements consist of vertices, edges, and faces, while the topological elements provide connectivity and adjacency information. It can be represented by:

1. A set of vertices (ie. points in $\mathbb{R}^3$)

$$\mathcal{V} = \{v_1, ...v_V\}. \tag{2.1}$$

2. A set of triangular faces

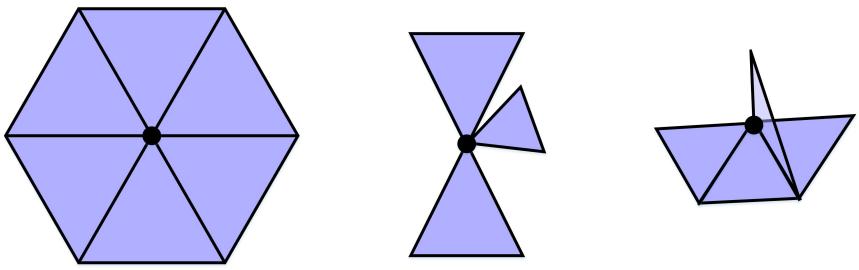
$$\mathcal{T} = \{t_1, ...t_T\}, t_i \in \mathcal{V}^3. \tag{2.2}$$

Sometimes, it is more efficient to represent the connectivity of a mesh by using the edges.

3. A set of edges

$$\mathcal{E} = \{e_1, ...e_E\}, e_i \in \mathcal{V}^2. \tag{2.3}$$

Then we write $M = (\mathcal{V}, \mathcal{T}, \mathcal{E})$.



(a) Manifold mesh (b) Non-manifold vertex (c) Non-manifold edge and non-conforming vertex

Figure 2.1: Vertex type and edge type of mesh.

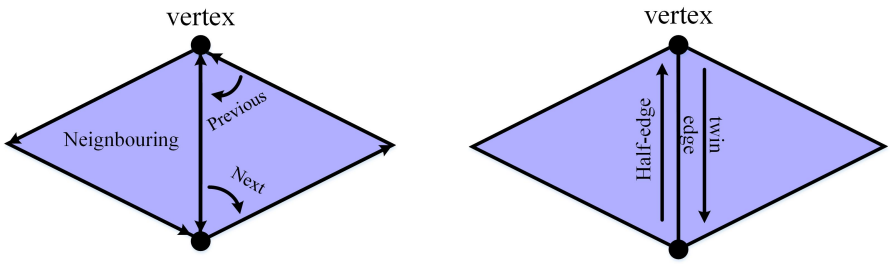
A triangle mesh is said to be manifold if it contains neither non-manifold edges nor non-manifold vertices nor self-intersections, such as in Figure 2.1 (a), where Figure 2.1 (b) is not manifold. A triangle mesh is said to be conforming means that at most two faces meet along

an edge, such as (c) is a non-conforming vertex and non-manifold edge. A manifold mesh can be thought of as a well-behaved, non-self-intersecting surface, which makes it widely used in computer graphics, 3D modeling, and computational geometry.

2.2.2 Mesh data structure

The data structure of mesh data has a direct impact on the efficiency and memory consumption of algorithms in geometry processing. The choice of a mesh data structure should take into consideration factors such as construction time, query time, memory consumption, and redundancy.

Triangle soup is the simplest representation of surface meshes, which store a set of triangles represented by their positions. However, this representation is often regarded as inefficient due to the duplication of vertices. This redundancy can be avoided by indexing the face data, which stores an array of vertices and the indices of triangles. This approach reduces memory consumption by allowing vertices to be shared among multiple triangles. It is a straightforward and efficient method for data storage, which is why many file data formats like OFF, PLY, and OBJ use this data structure. Similarly, indexed triangles are used in OpenGL for rendering, because they reduce the amount of vertex data that the GPU must process. However, the lack of connectivity makes it inefficient for searching and other specific operations.



(a) Winged-based (b) Half-edge data structure

Figure 2.2: Edge-based mesh data structure.

Edge-based mesh is another data structure to represent 3D meshes. The most common structure is the winged-edge [Bau72] data structure, which is depicted in Figure 2.2 (a). Each vertex stores its position and the reference to its incident edge, each face stores the edge reference, and each edge stores the reference to its two vertices and its two adjacent faces, and to its next and previous edges within the incident faces. The half-edge data structure can represent arbitrary meshes and maintain the incidence information of vertices, faces, and edges. Each edge will be decomposed into two half edges with opposite orientations, it can be seen in Figure 2.2 (b). This data structure allows other attributes such as texture coordinates or normals to be stored in the same position value with different other values of attributes. It is a powerful and flexible way to handle complex topologies and support efficient operations on the mesh. The directed-edge data

structure [CKS98] is a memory-efficient edge-based data structure, which is specifically designed for triangle meshes. Each face is presented with 3 directed edges, each edge has a starting vertex, a target vertex, and the previous, next, and neighboring edges.

2.3 Mesh simplification

Mesh simplification is a process of decimating the number of vertices, edges, and surfaces while preserving the topology and geometry features as much as possible. Given a complex 3D mesh model consisting of a large number of triangles, sometimes it is essential to do the simplification for further operation, such as multi-resolution model construction or efficient rendering. There are numerous mesh simplification algorithms that have been proposed by researchers in the field of computer graphics and computational geometry.

2.3.1 Error metric

An error metric is a measure of the quality of a simplified mesh compared to the original mesh. Guaranteeing the geometric error during simplification is the most important requirement for most proposed algorithms, different simplification methods correspond to different error metrics. Given two mesh models M_a and M_b, where M_b is the simplification approximation of M_a. Normally, the geometric distance is used to measure the similarity between two meshes. However, computing the exact geometric distance between two meshes is computationally expensive [CRS98]. Hausdorff distance and quadric error are two commonly used error metrics in mesh simplification.

Single-sided Hausdorff distance

The Hausdorff distance measures the distance between two subsets of a metric space, which is defined to be the maximum minimum distance. The single-sided Hausdorff distance $\mathcal{H}(M_b, M_a)$ is defined as

$$\mathcal{H}(M_b, M_a) = \max_{v_b \in M_b} d(v_b, M_a), \tag{2.4}$$

where d is Euclidean distance. In general, $\mathcal{H}(M_a, M_b) \neq \mathcal{H}(M_b, M_a)$, so the correct error metric will be the distance $\mathcal{H}(M_b, M_a)$ from the simplified model M_b to the original model M_a.

Hausdorff distance is an efficient metric to measure the difference between two meshes. There are many related methods have been proposed [BF05] [Coh+96].

Quadric Error Metric

Another useful error measurement is the Quadric Error Metric (QEM). The quadratic error can be used for the simplification method based on pair contraction [GH97]. For a pair contract operation $(v_1, v_2) \to \overline{v}$, the optimal position of $\overline{v}$ can be found by following:

$$\overline{v} = \arg\min_{v} \sum_{p \in plane(v_1) \cup \in plane(v_2)} d(v,p)^2 \tag{2.5}$$

where $plane(v_i)$ represents the planes related to v_i, and $d(v,p)^2$ is the square distance from a vertex to a plane.

Let $p = [a,b,c,d]^T$ represents the parameters of a general plane equation $ax+by+cz+d=0$, and $v = [x,y,z,1]^T$, the $d(v,p)^2$ can be represented as

$$d(v,p)^2 = (v^T p)^2 = v^T p p^T v = v^T K_p v, \tag{2.6}$$

where $K_p = pp^T$, then Eq. (2.5) can be written as:

$$\overline{v} = \arg\min_{v} v^T \sum_{p \in plane(v_1) \cup \in plane(v_2)} K_p v, \tag{2.7}$$

which can be also written as:

$$\overline{v} = \arg\min_{v} v^T (\sum_{p \in plane(v_1)} K_p + \sum_{p \in plane(v_2)} K_p) v. \tag{2.8}$$

Let $Q_i = \sum_{p \in plane(v_i)} K_p$, then there is:

$$\overline{v} = \arg\min_{v} v^T (Q_1 + Q_2) v. \tag{2.9}$$

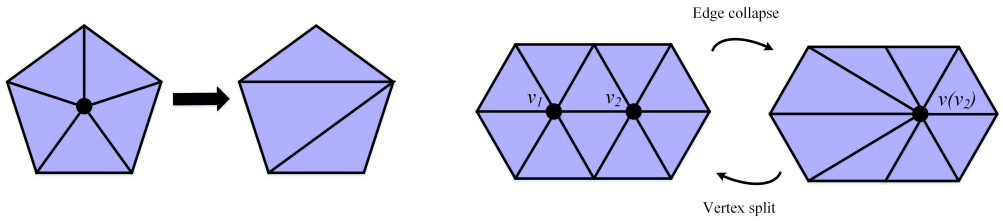
$\overline{Q} = Q_1 + Q_2$ can approximate the error at $\overline{v}$. The major advantage of QEM is memory consumption and computing efficiency. Since each vertex has only one 4×4 matrix Q, the cost can be computed efficiently. And no matter how many planes are associated with the vertex, the error can be represented by one single matrix Q. QEM-based simplification can produce high-quality simplified meshes with smooth surfaces and preserved features. However, it can be computationally expensive for large meshes.

2.3.2 Simplification method

Most of the simplification methods are iterative. The basic operations are vertex removal, face removal, and edge collapse, these operations are just for local modification. Because of the local modification, this kind of method can produce high-fidelity simplification to the original mesh model, which can be used to produce different level-of-details (LODs) representations.

Vertex removal

Vertex removal was first proposed by [SZL92], which removes the vertex and its surrounding faces (Figure 2.3 (a)). After the removal of one vertex, it can decimate one vertex and two faces. Each vertex is assigned to a different classification, and with each iterative step, the geometric



(a) Vertex removal

(b) Edge collapse

Figure 2.3: The basic operation of mesh simplification.

deviation introduced by vertex removal is computed. The main advantage of vertex removal is its simplicity and efficiency, however, it might lose important geometric and topological features of the mesh.

Edge collapse

[Hop+93] proposed edge collapse for mesh decimation, which is the most commonly used operator. As you can see in (Figure 2.3 (b)), this operator removes one vertex, three edges, and two triangles. With each iteration, the geometric error is computed based on the edge, and the edge with minimal error is collapsed first, and then the rest of the edge pairs are sorted. The final simplification is affected by the position of the target vertex.

Half-edge collapse

The half-edge collapse is the simplified method of edge collapse [RR96] [KCS98]. The possible collapse will be $e = (v_1, v_2) \rightarrow v_1$ or $e = (v_1, v_2) \rightarrow v_2$. Its advantage is that the simplified mesh M_b is a subset of the original mesh M_a, which can be used to further efficient progressive operations.

Vertex clustering

Vertex clustering was originally proposed by [RB93], first subdividing the bounding box of the mesh model into 3D cubes in order to cluster vertices. The vertices of cubes are grouped into a cluster, the representative vertex is chosen for each cluster based on an error metric, and then each vertices cluster is reduced to this target vertex. All vertices, edges, and triangles will degenerate to a vertex, so the complexity of the model is reduced. However, vertex clustering may lead to the loss of fine details and sharp features in the mesh. There are many variants of this algorithm, such as floating-cell clustering [LT97], approximated centroidal Voronoi Diagram [VC04], and its improved version [VKC05]. Vertex clustering is capable of simplifying entire mesh models globally in a fast and efficient manner, and it is relatively straightforward to implement. However, it is important to note that vertex clustering can introduce unpredictable changes to the model's topology.

2.3.3 Out-of-core mesh simplification

Because of the rapidly growing size of the 3D data models, for very large data sets, it is difficult to fit entirely into the main memory. The out-of-core algorithm allows us to do the mesh simplification without maintaining the entire data set in the main memory. The general process consists in partitioning the mesh into different clusters and each cluster is simplified independently in-core, storing the simplified tile back to disk and unloading it from memory. Until the entire mesh has been simplified, all simplified meshes of different clusters are joined together to create a simplified version of the entire mesh.

Lindstrom [Lin00] presented an algorithm for out-of-core simplification of large polygon datasets which is based on vertex clustering. Linstrom and Silva [LS01] improved this approach by removing the requirement for the output model to fit into the main memory while preserving surface boundaries that do not use connectivity information. The shortcoming of this approach is that the overall simplification has a constant feature size, the resulting mesh may lose detail in areas where the feature size is smaller than the specified size.

Wu and Kobbelt [WK03] presented a streaming approach to the out-of-core mesh simplification based on edge collapses and QEM. However, the boundary meshes of the original mesh cannot be decimated, since the stream algorithm cannot distinguish true mesh boundaries, this can be a problem for meshes with many boundaries.

Cignoni [Cig+03b] presented a data structure called Octree-based External Memory Mesh (OEMM), which can support the management of complex meshes, loading partial meshes into the memory. The OEMM enables the loading of partial meshes into memory for processing and supports various operations such as mesh simplification, mesh editing, and visualization.

Shaffer and Garland [SG01] proposed an adaptive surface simplification algorithm capable of efficiently producing high-quality approximations of massive polygonal models, which combines an out-of-core vertex clustering step with an in-core iterative decimation step based on QEM. Ozaki [OKK15] proposed an out-of-core framework for QEM-based mesh simplification, each sub-mesh can be simplified independently, and the nature of parallel computation makes the whole process time and memory efficient.

2.3.4 Neural mesh simplification

Most traditional simplification methods decimate the input mesh iteratively based on a cost function to preserve the topology. With the development of neural networks in recent years, neural mesh simplification uses neural networks to simplify 3D meshes by reducing the number of vertices and faces while preserving the visual quality and geometry of the original mesh. The technique involves training a neural network on a large dataset of 3D meshes to learn a mapping between a high-resolution mesh and a simplified mesh with fewer vertices and faces. Some researchers have proposed mesh simplification methods based on neural networks [PPZ22] [Álv+07] [LHZ20]. Rafael Álvarez et al. developed a mesh simplification method called GNG3D

which is able to produce high-quality approximations of polygonal models, it consists of an optimization phase and a reconstruction phase [Álv+07]. Yaqian Liang et al. proposed a new method to obtain the optimal simplified 3D mesh models with the minimum approximation error, which includes a feature-preservation edge collapse operation by using Gauss curvature and QEM, a hybrid "undo/redo" mechanism combined with edge splitting and edge collapse operation, which can be used to reduce approximation error [LHZ20]. Rolandos Alexandros Potamias et al. proposed a fast and scalable method that simplifies a given mesh in one pass, compared to traditional methods with a greedy iteration manner [PPZ22]. This architecture includes three components: the point sampler, the edge predictor, and the face classifier. It has advantages of the run-time efficiency and a lightweight structure, which can be directly plugged into any rendering process without introducing any significant overhead. A limitation of this method is the preservation of topology and manifoldness of generated mesh can not be explicitly guaranteed based on the tailor-made loss function.

2.4 Multi-resolution model structure

A multi-resolution model is a structural framework used to represent 3D models at various LODs. LOD [Cla76] was proposed by James H. Clark, using different LODs to represent the model. It plays an important role in real-time applications such as terrain modeling, virtual reality, and rendering [HD04] [FKP05]. The basic element of LOD is a *base* mesh level M_0, which is the coarsest approximation, the mesh at full resolution is called *reference* mesh M. A set of *update* exists between the M and M_0. The *dependency relation* is generated during the *update* operations, which allows combining them to extract consistent intermediate representations [FKP05].

There are several kinds of multi-resolution frameworks: discrete LOD, continuous LOD, hierarchy LOD, multi triangulations, and neural geometric LOD [Tak+21]. The LOD algorithm includes generation, selection, and switching [AHH19]. LOD generation is the process of construction of different representations of models varying in details, mesh simplification is used to generate different LOD. LOD selection is based on some criteria, such as the distance from the viewpoint to the object. LOD switching is changing the current to another LOD that satisfies the LOD slection criteria.

2.4.1 Discrete level of detail

The simplest LOD is discrete LOD, which involves creating several independent approximations of a model with varying levels of detail. The generation of discrete LOD is usually a preprocessing step. However, discrete LOD has its limitations, particularly when it comes to drastic simplification scenarios, such as high-detailed range scan models, terrain flyovers, and massive CAD models. The other drawback is the occurrence of *popping* artifacts during the LOD switching process. This *popping* is caused by the independent nature of the different LOD approximations,

which can lead to noticeable changes in the model's appearance when transitioning between LOD levels.

2.4.2 Continuous level of detail

Continuous Level of Detail (CLOD) offers a more refined approach to LOD management compared to discrete LOD. It involves storing edge collapse and edge split operations, which can be reversed, allowing for on-the-fly extraction of the appropriate LOD at runtime. One of the most commonly used CLOD techniques is the progressive mesh proposed by Hoppe [Hop96].

The key advantage of CLOD over discrete LOD is its ability to preserve more details, providing a higher level of granularity in representing complex objects or scenes. The edge collapse operation in CLOD facilitates smooth transitions between different LODs, reducing the *popping* effect that can occur during LOD switching. Additionally, CLOD can even perform geomorphing, which smoothly interpolates fine-grained simplification operations over several frames, further eliminating the *popping* artifacts [Hop98a].

However, CLOD also comes with its challenges. The detail selection process involves traversing the CLOD structure and updating each vertex and edge in the current LOD, which can become a bottleneck in performance [GKY08]. Moreover, CLOD may not be the most suitable approach for large models due to increased memory requirements and the growing performance gap between GPU and CPU.

2.4.3 Hierarchy level of detail

The hierarchy level of detail (HLOD) algorithm is the generalization of traditional LOD methods to hierarchical aggregations of objects [EM00]. In the context of 3D meshes, the HLOD structure is typically managed using an octree, while for terrain models, a quadtree is commonly employed. The process of generating HLOD involves several key steps. Initially, the mesh of objects is partitioned into the 3D space to form the basis for subsequent simplification. Next, mesh simplification is performed, starting with the full-resolution model and gradually building consecutive LODs with decreasing levels of detail. This step creates a series of different approximations of the original model, each with varying levels of detail. A significant aspect of HLOD construction is the establishment of child-parent dependencies during the LOD generation process. The root of the HLOD tree represents the coarsest approximation of the model, with each subsequent child node offering a higher resolution than its parent node but occupying a smaller spatial area.

HLOD has been proved to be particularly well-suited for rendering massive models and scenes containing small objects. When combined with the view-dependent method, the low overhead on the CPU and efficient traversal of the HLOD tree enables it to achieve efficient rendering of vast and complex models. Additionally, the hierarchical spatial mesh partitioning nature of HLOD makes it straightforward to implement frustum culling and occlusion culling techniques,

further optimizing the rendering process. Moreover, HLOD naturally integrates with out-of-core rendering, allowing for the efficient management and streaming of data from secondary storage, which is essential when dealing with models that exceed the available GPU memory.

Despite these advantages, the HLOD structure introduces a new challenge, known as *mesh cracks*, which occurs between adjacent LOD parts with different resolutions. These cracks (as shown in Figure 1.4) can result in visible artifacts and discontinuities in the rendered model, detracting from the overall visual quality.

2.4.4 Multi triangulations

The Multi Triangulations (MT) is a general framework to describe multi-resolution models, which is introduced in [FPM97], [Pup+96], and [Pup98]. Leila De Floriani et al. proposed a general CLOD management framework, which encodes the partial order describing mutual dependencies between updates as a Directed Acyclic Graph (DAG) [DMP98]. Paolo Cignoni et al. proposed Batched Multi Triangulation (BMT) [Cig+05], which is a robust way to build a well-conditioned MT DAG by introducing the concept of V-partitions. A V-partition is a sequence of coarser and coarser partitions over a mesh that is independent of the geometric dimension of the model or the choice of the primitive used to represent the surface [Pon09a]. Adaptive TeraPuzzles [Cig+04] uses a regular conformal hierarchy of tetrahedral to partition the 3D model spatially. Each tetrahedral contains a simplified version of the original model. This multi-resolution representation is constructed by a fine-to-coarse simplification of the surface contained in sets of tetrahedral cells sharing their long edges. It is possible that only one triangle falls in a tetrahedron, which is a common problem for regular spatial partitioning, whether it is octree or TeraPuzzles.

2.4.5 Neural geometric level of detail

Neural geometric level of detail (Neural GLOD) is used for rendering the 3D data by using deep neural networks to generate simplified representations of the data at different LODs. The basic idea behind Neural GLOD is to train a deep neural network to learn the simplification process for a 3D scene, resulting in multiple LODs that represent different resolutions. Different from the traditional surface representation, implicit surface-based encode geometry in latent vectors or neural network weights, which parameterize surfaces through level-sets [Tak+21], where signed distance functions (SDFs) is a general implicit representation. DeepSDF [Par+19] and Occupancy networks [Mes+19], use a large multilayer perceptron (MLP) to output value conditional on latent vector and position. Thomas Davies et al. proposed OVERFITSDF, a neural network architecture trained to overfit to a single shape signed distance function [DNJ20].

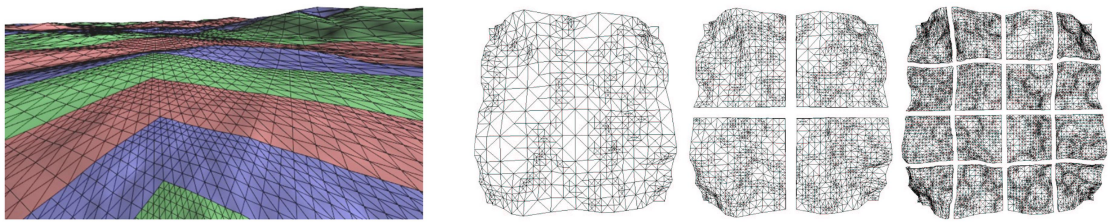
Towaki Takikawa et al. [Tak+21] proposed Neural LOD, which stores features in sparse voxel octree (SVO). In order to build CLOD, the discrete octree LOD blending is executed by linearly interpolating the corresponding predicted distance. The principle of selecting LOD is based on

the distance to the object. This neural LOD structure can achieve state-of-the-art geometry reconstruction quality while allowing real-time rendering with an acceptable memory footprint. However, it is not suitable for large scenes or very thin, volume-less geometry because of the high dependency on the point samples during the training.

Overall, the advantage of Neural GLOD is that it can be applied to a wide range of 3D datasets, and be used to generate simplified representations of the data that are optimized for specific applications, such as video games. However, one of the main challenges is that the Neural GLOD must be trained on large and diverse datasets of 3D scenes. Moreover, the performance and efficiency of Neural GLOD may be influenced by the training data's representativeness.

2.4.6 Multi-resolution terrain model

Multi-resolution terrain is designed to adapt to the characteristics of terrain, it is an important technique for optimizing the rendering of large-scale terrains in real-time applications such as video games, flight simulators, and virtual environments [DVP22]. Many multi-resolution terrain model methods have been proposed, such as Geometry Clipmaps [LH04], Chunked LOD [Ulr02], and Batched Dynamic Adaptive Meshes (BDAM) [Cig+03a], where Geometry Clipmaps and Chunked LOD have been widely used in industry.



(a) Geometry Clipmap (b) Chunked LOD

Figure 2.4: (a) Geometry clipmap, image taken from [LH04]. (b) Chunked LOD, image taken from [Ulr02].

Geometry Clipmap

Geometry Clipmap caches the terrain in a set of nested regular grids centered about the viewpoint. This framework offers several advantages, including visual consistency, uniform frame rates, complexity throttling, and graceful degradation. Additionally, it supports decompression and compositing processes, and is ideal for GPU rendering since it represents terrain as a regular grid of height values. However, it has no screen space error control. There are some other improved versions, such as Shi Li et al. proposed a multi-resolution terrain rendering algorithm that utilizes summed-area tables to facilitate the rendering of terrain with better geometric and shading details [Li+21]. This method is specifically designed as an error-bounded screen-space terrain rendering approach.

Chunked LOD

Chunked LOD, proposed by Thatcher Ulrich, organizes the terrain data structure by quadtree. The basic idea behind Chunked LOD is to divide the terrain into a hierarchy of squared-shaped tiles. The quadtree structure allows for efficient traversal and indexing of the squared-shaped tiles, making it possible to select the appropriate LOD of tiles quickly. It has been widely used in the industry for terrain rendering because of its scalability and the guarantee of screen-space error. The *skirt* (extra mesh ribbon) method was applied to prevent the mesh cracks, and morphing was used to smooth the LOD popping. One of the limitations of Chunked LOD is using more triangles compared to certain other methods when aiming for the same level of screen-space error control. This increased triangle count results from the generation of additional geometry during the rendering process in order to avoid mesh cracks.

BDAM

BDAM, proposed by Paolo Cignoni et al. in 2003, utilizes a right triangle hierarchy stored as a binary tree (bintree) to provide a higher-level representation of data partitioning. In this method, individual triangles are replaced with small mesh patches, serving as the smallest manageable entities. These small patches can be constructed and optimized offline using simplification and tristripping algorithms. An extension of BDAM called Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM), as detailed in [Cig+03c]. P-BDAM is designed for the management of out-of-core data and enables interactive rendering of textured terrain surfaces on a planetary scale.

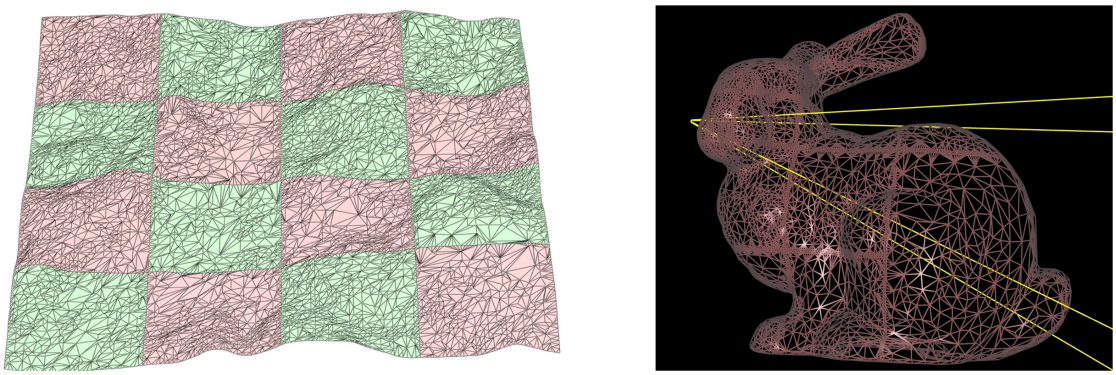
2.5 Visualization of massive models

2.5.1 Rendering based on multi-resolution model

In recent years, many researchers have proposed different frameworks to achieve high-fidelity real-time rendering of large-scale 3D models. The key problems of using multi-resolution structures to render massive 3D models are mesh cracks and LOD popping.

The direct solution [EMB01] [Hop98b] is locking the shared mesh edge of boundary between clusters during simplification, so the edges will always match their boundaries during rendering because they never changed, however, this might lead to the dense accumulation of triangle cruft around the boundary edges (Figure 2.5) at coarse LOD. The presence of triangle accumulation may further cause inefficient rendering and overdraw problems.

In the field of visual rendering of large terrain data, many researchers [PG07] focus on how to achieve smooth and seamless interactive rendering. ChunkLOD is implemented for regular grids, the cracks between different LOD parts are dealt with using "skirts", which might introduce additional hidden geometry and require lots of clipping, resulting in additional draw calls, making it less efficient to apply this method to arbitrary meshes.



(a) (b)

Figure 2.5: Dense triangle accumulation around the boundary edge is caused by the HLOD construction method. (a) The hierarchical PM construction process partitions the model into blocks, recursively simplifies, and combines the blocks [Hop98b]. (b) The demonstration of how HLODs are generated with partitioning approximate view-dependent simplification [EMB01].

The other methods are based on progressive mesh (PM) structure [Hop96], such as dependency-free parallel progressive meshes [DG12], Parallel view-dependent out-of-core progressive meshes [DMG10], and parallel view-dependent level-of-detail control [HSH09]. They are the view-dependent progressive mesh structure. The construction of data structure takes a long time due to the operations of vertices , the data size of those construction and editing are fairly large as well. For [DG12], the reorganization of the vertices is rather expensive and [DMG10] needs an additional memory reduction. These methods can prevent cracks perfectly, but the computation time and complexity are expensive. Moreover, rendering each part of the model in high granularity is not necessary.

The progressive buffer [SM05] did not give many details about the hierarchy multi-resolution construction. This method was designed when GPUs were mostly accessed through the fixed function pipeline. It had in particular double vertex attributes and thus incurs vertex cost. Adaptive TetraPuzzles [Cig+04] partitions the mesh into a hierarchy of tetrahedra, which is a spatial partition because of the limited control over the number of triangles per cluster. It is possible that only one triangle falls in a tetrahedron, which will make it very inefficient to render. The pop buffer [Lim+13] achieves crack-free borders by sorting a small number of protected vertices to the beginning of the vertex buffer, it aims at web-based environments and mobile clients, not for massive model rendering.

BMT [Cig+05] and Quick-VDR [Yoo+05] build the boundary constraints and cluster dependencies during the hierarchical simplification. Quick-VDR represents a 3D model as a clustered hierarchy of progressive meshes (CHPM), it allows the real-time coarse-grained as well as fine-grained refinement based on the viewpoint position. However, the cluster dependencies of Quick-VDR that necessitate additional cluster-split operations might cause popping artifacts.

BTM [Cig+05] uses a DAG to build the multi-resolution representation, which can be efficiently extracted and batched to the GPU by simply combining precomputed patches during the rendering. These methods eliminate cracks and smooth the popping artifacts by controlling the screen space error associated with a patch.

Nexus [Vis20] is a library for the creation and visualization of a batched multi-resolution 3D model structure. It combined the feature of BMT [Cig+05] [Pon09b], Adaptive TetraPuzzles [Cig+04], and BDAM [Cig+03a]. Nexus is divided into 3 parts: multi-resolution model construction, multi-resolution model editing, and multi-resolution rendering.

Nanite [KSW21] is a new and improved virtualized geometry system that uses a new internal mesh format and rendering technology to render pixel scale detail and high object counts. It is a hybrid implementation of [Cig+05] and [Yoo+05]. However, building and rendering Nanite structures in real time can require significant processing power and computing resources, and achieving expected performance on general mainstream computer platforms can be challenging.

2.5.2 Out-of-core rendering

Since the complexity and size of the model are increasing gradually, out-of-core rendering is a technique used to render large data sets that do not fit entirely into the memory of the GPU. It works by loading only a portion of data into the GPU memory at a time and swapping in and out different parts of the data as required. This allows for the rendering of massive data without a prohibitively large amount of memory. Several out-of-core rendering algorithms have been proposed [Lin03] [GK04] [DMG10] [VM02] [VM02] [FS93] [RL01] [WDS05] [Sar+22] [Bax+02] [Bor+05]. The general process of constructing a multi-resolution model representation involves several steps, including partitioning the mesh into different clusters, out-of-core mesh simplification, and LOD construction. The multi-resolution model is constructed in an out-of-core manner, where the data is too large to fit into the main memory, so the data is stored on disk and loaded into memory when it is needed. During the multi-resolution construction, a tree structure for managing mesh partitions is generated, which is usually stored in the main memory. The final multi-resolution model is stored on the hard disk in a file format for later use. In the rendering phase, a data tree traversal (octree or quadtree) is performed based on detail selection criteria, and the renderable data is then streamed from the disk to the main memory. However, the streaming operation can be the bottleneck of the rendering performance, especially for real-time applications that require high frame rates. To address this issue, the data streaming thread and the rendering thread are normally separated in order to achieve better performance.

2.5.3 Neural rendering

Neural rendering uses deep learning models to generate images from 3D data, and the ability to handle large datasets is often more related to the hardware and software used to perform the rendering. There are some researchers who focus on rendering implicit surfaces based on

neural networks [Tew+20] [Kat+20]. It simulates the physics of a camera capturing the scene, which is different from the classic rendering methods: rasterization and ray tracing. Shichen Liu et al. [Liu+19] proposed soft rasterization, which is a differentiable rendering framework that is able to render a colorized mesh in the forward pass. It can consider a variety of 3D properties, including mesh geometry, vertex attributes (color, normal, etc.), camera parameters, and illuminations, and is able to flow efficient gradients from pixels to mesh vertices and their attributes, but it can not handle shadows and topology change. Overall, neural rendering can be used to generate high-quality rendering images from large 3D datasets, but it needs to be trained on sufficiently large and diverse training data. In fact, one of the advantages of neural rendering is that can generate photorealistic images even from incomplete or noisy data, which can be useful when working with large datasets that contain errors or may not be fully captured. However, in practice, the size and complexity of the large-scale data sets can require significant computational resources, such as powerful CPU and GPU.

2.5.4 Real-world terrain model reconstruction

The terrain model is a crucial component of many computer graphic applications, creating surface representations of terrains with realistic landscapes is important for many applications [Gal+19] [Nat+13]. The methods can be divided into 2 categories: procedural terrain generation (data-free) and data-driven terrain reconstruction by measured data, such as LiDAR point cloud and Digital Terrain Model (DEM).

Procedural terrain generation uses mathematical functions and algorithms to generate terrain, without relying on measured data from the real world. Fractal landscape modeling and physical erosion can be used for surface creation [BF01]. Szymon Stachniak et al. used fractal models to create terrain models, but users have low control of the resulting models [SS05]. Sketch-based terrain generation is often used in computer graphics and game development to quickly create realistic terrain. Jonathon Doran et al. combines procedural modeling with user constraints and the terrain model creation is based on a height map [DP10]. Erosion-based terrain generation is a type of terrain generation method that simulates natural erosion processes [BF01]. Ondřej Šťava et al. uses hydraulic erosion, which is implemented on GPU [Šta+08]. Adrien Peytavie et al. can create complex landscapes such as overhangs, arches, and caves [Pey+09].

Data-driven terrain reconstruction uses measured geologically measured data, such as LiDAR point cloud [Top23b] and DEM [Top23a]. It can provide a more accurate representation of real-world terrain and can be useful for simulating terrain changes and predicting future landscape patterns. Compared to procedural terrain generation, real-world terrain data can add significant value to games, virtual reality, films, and the metaverse. For example, it can enhance the player's immersion and overall gaming experience, save time and effort in the development process, and discover real-world locations within the game world. The increasing diversity and size of geospatial data have opened up new possibilities for terrain modeling.

2.5.5 Real-world terrain model rendering

We divide real-time terrain rendering into the following three categories: heightmap-based terrain rendering, Triangulated Irregular Network (TIN) based terrain rendering, and volumetric terrain rendering.

Heightmap-based terrain rendering

Some terrain visualization methods are developed for regular grids, and the multi-resolution model is based on quadtree [Paj98] [LP02]. The other group of methods is based on the bin-tree area [Cig+03a] [Cig+03c]. All these methods construct the LOD on the terrain model through split-merge operations or use quadtree structures to manage mesh hierarchy in object space. Some methods adjust the level of detail based on the viewpoint and adapt their borders to match the resolution of neighbor chunks [LKE09] [LH04] [Ulr02]. Geometry Clipmap and Chunked LOD are widely used in industries, such as game engines and virtual globes. The OGRE engine [Eng23] and OpenGlobe [CR11] are virtual globes, which are based on Geometry Clipmap. The *Far Cry 5* of Ubisoft [Moo18] and *Serious Sam 4: Planet Badass* [Lad19] were developed based on ChunkLOD. There are also improved visualization methods based on the Geometry Clipmap and Chunked LOD, such as CDLOD (Continuous Distance-Dependent Level of Detail) [Str09], which tried to overcome the disadvantages of Geometry Clipmap and Chunked LOD by stitching the different LOD parts and using morphing to eliminate the LOD popping. Dupuy [Dup20] introduces the concurrent binary tree (CBT), a novel concurrent representation to build and update arbitrary binary trees in parallel. A large-scale regular grid terrain renderer based on CBT is implemented, which includes efficient parallel algorithms that can run entirely on the GPU. The above methods developed for height-map rendering are primarily designed for terrain models represented using a regular grid structure. When dealing with terrain models composed of irregular triangles, applying these methods can pose challenges.

TIN-based terrain rendering

TIN-based terrain refers to a type of terrain representation that uses a TIN to model the surface of the terrain. For the terrain mesh hierarchy which contains irregular triangles [CL96] [Hop98a], Daniel Cohen-Or et al. presented an algorithm that establishes a correspondence between two given polygonal objects in an off-line process, which can achieve a blending of two objects online [CL96]. Hugues Hoppe proposed a mesh hierarchy based on view-dependent progressive mesh (VDPM), which can eliminate mesh cracks between different LOD parts [Hop98a]. BDAM can also manage the TIN-based terrain model with its proposed structure [Cig+03a].

Volumetric terrain rendering

Volumetric terrain rendering is a technique used to create realistic 3D terrain models by representing the terrain as a volume of 3D data rather than a surface or a mesh. It can simulate natural

phenomena such as erosion, weathering, and vegetation growth, resulting in terrain that looks more realistic and natural. Manuel Scholz et al. presents a practical and GPU-friendly LOD for large-scale volumetric terrain that is designed for real-time rendering algorithm [SBD13]. However, volumetric terrain rendering may not be able to capture the same level of detail as other terrain rendering techniques, such as height maps or TINs.

2.6 Conclusion

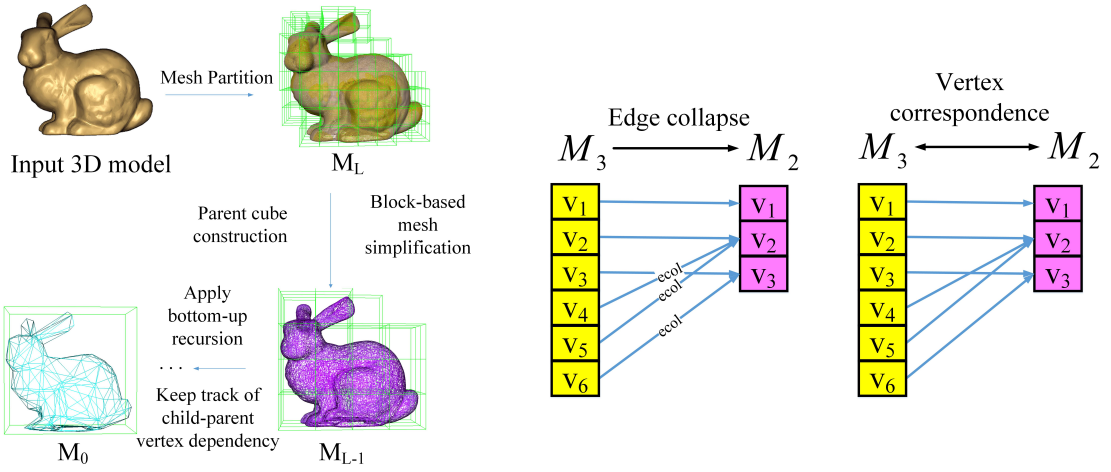
In this chapter, a review is provided on mesh simplification, the construction of multi-resolution models for arbitrary triangle mesh models and terrain models, and rendering methods for both in-core and out-of-core scenarios based on these multi-resolution models. The next chapter will focus on the structure and construction approach of our proposed multi-resolution model.

Chapter 3

Multi-resolution model construction

3.1 Overview of multi-resolution model

The multi-resolution construction method allows for generating hierarchy representation with different LODs, which is based on HLOD and can be used in a view-dependent manner. This structure can be used to avoid LOD popping and mesh cracks in the following real-time rendering. It includes mesh partitioning, cube-based mesh simplification, and LOD construction, with the following features:



(a) HLOD construction

(b) Child-parent dependency

Figure 3.1: (a) The overview of hierarchical multi-resolution model construction. (b) The child-parent dependency between two successive LODs, where the number represents the index of the vertex, *ecol* represents v_6 collapse to v_3. For example, the v_3 of M_2 is the parent vertex of v_3 and v_6 in M_3.

(1) *Transparent and controllable vertex dependency.* During the fine-to-coarse mesh simplification, the dependencies between each vertex at different LODs are established, which is the simple child-parent relationship between vertices. It allows us to manipulate the dependency of vertex belonging to different LODs easily.

(2) *Fast HLOD construction.* A multi-resolution representation can be constructed at a rate of the order of 1M triangles per second per core on a mainstream desktop PC, for a model with multiples of tens of millions of triangles. Fast HLOD construction can save a lot of time and improve performance, for example, it helps researchers inspect mesh quality quickly during data collection, such as field archaeological data collection.

(3) *Memory efficient.* The HLOD structure generates LOD sequences, resulting in an increase in the number of vertices and triangles in the final HLOD model representation. The memory consumption required by our HLOD representation is less than or around twice that of the original model, which is memory efficient.

The overall construction process is shown in Figure 3.1. We will introduce the following 3 sections: in-core multi-resolution model construction, out-of-core multi-resolution model construction, and the implementation and test results of these two parts related algorithms.

3.2 In-core multi-resolution model construction

3.2.1 Mesh and mesh simplification

Multi-resolution model construction proposed in this thesis focuses on triangle meshes. The geometry of a triangle mesh M is described in the sequel by a set of vertices $V = \{v_i \in \mathbb{R}^3 | i \in \{1, ..., m\}\}$ and a finite set of triangles $T \subset V^3$, we then write $M = (V, T)$. We say that two triangle sets $T_1, T_2 \subset V^3$ are equivalent, and we then write $T_1 \simeq T_2$, if each triangle in any of them is either degenerate (i.e. at least two of its vertices coincide) or present in the other one, up to a possible (orientation preserving) permutation of its vertices.

Given a mesh set $M = (V, T)$ and a mesh set $M' = (V', T')$, we say that M' is a simplification of M, and we then write $M' \leq M$, if there exists a surjective map $P : V \to V'$ such that:

$$T' \simeq P(T) := \{(P(v_1), P(v_2), P(v_3)) \; with \; (v_1, v_2, v_3) \; \in T\}.$$

For $v \in V$, we refer to $P(v) \in V'$ as the parent of v. As shown in Figure 3.2, after the mesh simplification of M, we get $P(v_1) = v_4$. For a triangle $Tri = (v_1, v_2, v_3) \in T$, we use the same notation $P(Tri)$ to refer to the triangle $(P(v_1), P(v_2), P(v_3))$. The *key point* for the understanding of the sequel is this: although the mesh that will effectively be rendered each frame shall be constituted of different parts of different LODs of the original mesh, it can be proved to be a true simplification of the original mesh in the above meaning. This by itself is sufficient to warrant that no crack can be present in it if there is no crack in the original mesh.

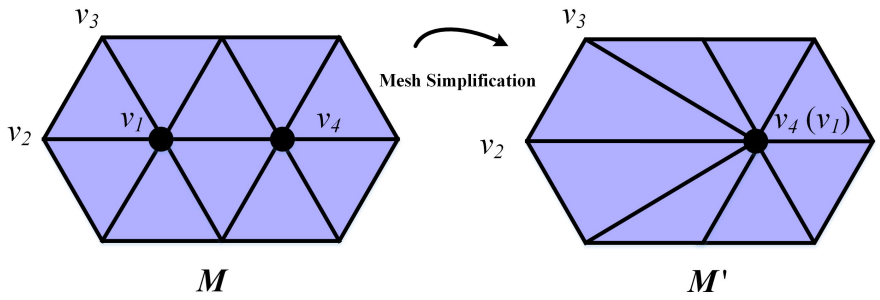


Figure 3.2: After mesh simplification (half-edge collapse), the child-parent vertx relationship is established.

For the representation of distance d mentioned later, where d is not a Euclidian distance, but the max distance, given two vertices $v_1(x_1, y_1, z_1)$ and $v_2(x_2, y_2, z_2)$, the

$$d(v_1, v_2) = max(|x_1 - x_2|, |y_1 - y_2|, |z_1 - z_2|).$$

The reason for the utilization of maximum norm is because the cube is the unit ball for the L^∞, and therefore well adapted to octree.

Mesh simplification algorithms based on edge collapse are therefore particularly adapted (edge flips would not). An advantage of edge collapse is that it can establish a vertex correspondence between simplified mesh and original mesh. This correspondence ensures that vertices in the simplified mesh can be associated with their original vertices in the higher-detail mesh, forming a child-parent vertex relationship.

In our implementation, we employ a variant of the QEM-based half-edge collapse algorithm, as outlined in [GH97], [GH98], and [Gar99]. To further enhance efficiency, we integrate an efficient hash function [VCP19] and a quick vertex array sorting algorithm [Tes+03]. These optimizations can improve the efficiency of vertex searching and sorting edge collapse pairs with quadric errors, which plays a crucial role in the QEM-based edge collapse algorithm.

Moreover, meshoptimizer [Kap20] has observed that processing speed can be substantially improved by using an approximate sorting algorithm for QEM-based edge collapse (radix sort), while the difference in quality with respect to exact sorting is not noticeable.

In our case, the target number of simplified vertices and the quadric error threshold (ϵ) are user-defined preset values. These values provide control over when the mesh simplification process should terminate. Once the edge adjacency information is generated, the quadric data for each vertex is initialized. This data includes both the face quadric. The face quadric represents the summation of the quadric matrices of the planes of the triangles that converge at this vertex. For the boundary edge or seam edge, the quadric of its vertices also includes the edge quadric, the edge quadric represents the summation of the quadric matrices related to the edges that meet at this vertex.

During the initialization of the face and edge quadrics, weight values are computed based on various factors. For the face quadric, the weight is calculated using the area of the triangle. Similarly, the weight of the edge quadric can be determined based on the length of the edge. To ensure that the border edges of the model are preserved, a high-weight value can be assigned to them. This weighting strategy is employed to maintain the silhouette and important features of the model during simplification.

In order to achieve mesh simplification more efficiently, in the operation of building a unique position map, we use the hash coding method in [VCP19] to encode the vertex position. For a vertex v with position (x, y, z), the vertex position is casted into unsignd integers, and then the following hash function is applied:

$$hash(x, y, z) = (\textbf{cast<int>(x)} \cdot p_1 \textbf{ xor cast<int>(y)} \cdot p_2 \textbf{ xor cast<int>(z)} \cdot p_3) \textbf{ mod } n,$$

where p_1, p_2, and p_3 are three large prime numbers, these three values are 73856093, 19349663, 83492791, and n is the table size. This function has been evaluated in [VCP19], which is very efficient and produces a comparatively small number of hash collisions for small hash tables.

Many models incorporate additional surface properties, such as texture coordinates or color, where texture coordinates are used to map textures onto the mesh's surface. When simplifying the mesh with texture mapping, it is important to preserve the texture coordinates to maintain the appearance of the texture on the simplified mesh. For the textured model, each vertex denoted as $v = [x, y, z, u, v]$ represents an n-dimensional point that defines a 2D plane within $\mathbf{R}^n$. So the squared distance of the arbitrary vertex $v \in \mathbf{R}^n$ to this plane can be the quadric matrix, and the derived squared distance D^2 can be rewritten as $D^2 = v^T \boldsymbol{A} v + 2\boldsymbol{b}^T v + c$. In this generalized quadric, $\boldsymbol{A}$ is an $n \times n$ matrix, and $\boldsymbol{b}$ is an n-vector. Given that the quadric error maintains the same structure as before, it can be integrated into existing algorithms without major modifications.

3.2.2 Multi-resolution model

Starting from the input mesh model M, the highest resolution mesh level M_L will be constructed first, where maximum level $L \geq 0$ will appear from the sequence, and capture details at a scale of order 2^{-L}. We consider a sequence of simplifications of M:

$$M_L \geq M_{L-1} \geq \cdots \geq M_0.$$

This sequence can be seen as a representation of a multi-resolution model, where l presents the LOD of different levels. Typically, M_L represents the highest-resolution mesh, and M_l represents a simplified version of the mesh with a progressively lower level of detail as l decreases.

We assume the following simplification constraint, where $\sigma_0 > 0$ is a model parameter:

1. Each edge in M_L has a length of at most $\sigma_0 2^{-L}$.

2. For each $1 \leq l \leq L$ and for each $v \in M_l$,

$$d(v, P(v)) \leq \sigma_0 2^{-(l+1)}. \tag{3.1}$$

A consequence of these two conditions is that for $1 \leq l \leq L$, each edge in M_l has a length of at most $\sigma_0 2^{-l}$. In most cases, there's no need to explicitly enforce this distance constraint. The value of σ_0 can be understood as a threshold that measures the uniformity of edge lengths in our method. Longer edges may need to wait for several LODs to be generated before being simplified (indeed, $\sigma_0 2^{-(l+1)}$ grows by a factor of 2 between each LOD level).

3.2.3 Full-resolution LOD construction

We assume the mesh of the input model M is represented as a list of triangles based on indices and a set of vertices, where each vertex may contain position, normal, and texture coordinates (or color). With the preset highest LOD level L and input model M, the full-resolution LOD M_L can be constructed as follows:

At the top level L, we split the Euclidean space according to the axis-aligned regular voxel grid G_L of size 2^{-L} centered at the origin, and partition the triangle set T_L according to

$$T_L = \bigcup_{C_L \in G_L} T(C_L),$$

where

$$T(C_L) = \{Tri \in T_L, \text{ s.t. } bary(Tri) \in C_L\},$$

and where $bary(Tri)$ refers to the barycenter of Tri. Whenever the barycenter falls exactly at the border between adjacent cubes, we use an arbitrary tie rule so that the former decomposition of T_L is an actual partition in the mathematical meaning. In particular, we do not recut triangles according to the grid. As a result, some triangle parts will fall outside the cube.

3.2.4 Block-based mesh simplification

After partitioning the original model into the cube C_L to create M_L, the subsequent step involves performing mesh simplification to generate the next coarser LOD M_{L-1}. Given the immense size of the model, simplifying the entire model directly is impractical due to memory constraints. Therefore, the approach is to select a block within the cube, effectively forming a mesh cluster, and then execute the mesh simplification process on this cluster.

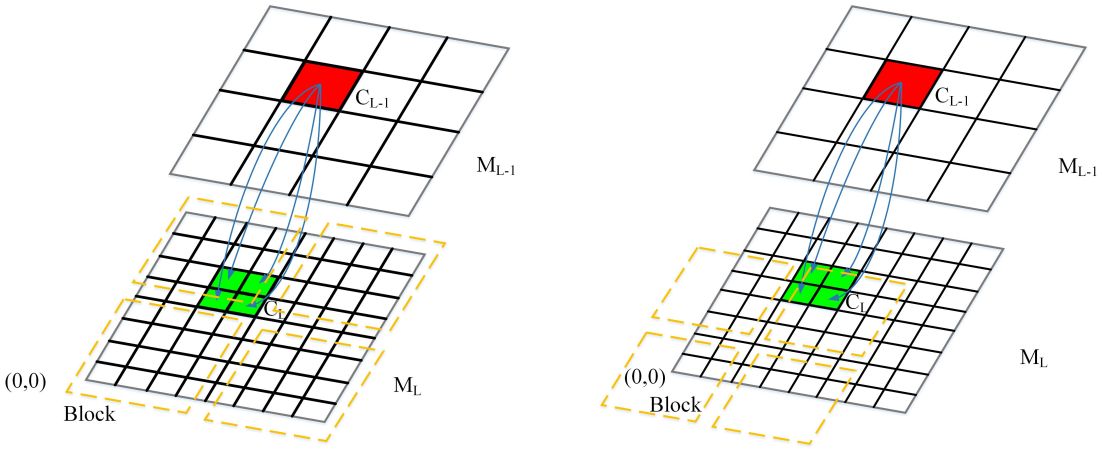
We select $n \times n \times n$ cubes to form a block, mesh simplification is executed on the mesh cluster related to this block, where n represents the number of cubes along each axis (X, Y, and Z), the maximum value is 2^L, and usually $n > 2$. The mesh simplification process is terminated when the number of triangles reaches a preset value, or when the target error threshold ϵ (representing the error relative to the range that the model can tolerate) is reached. The preset target error

ϵ is the overall error for the whole object, for each selected block, we will adjust ϵ according to the block size and the length of the cube, so the quadric error ϵ_{block} is computed as follows:

$$\epsilon_{block} = \frac{\epsilon}{nL_c 2^{-l}}, \tag{3.2}$$

where n represents the block size, and L_c corresponds to the length of C_0, which is the length of the bounding box of the entire object.

The boundary of each mesh partition related to a block is preserved during mesh simplification in order to maintain the conformality of the resulting mesh. However, when using a small value of n and a poor choice of block selection, some boundaries may be preserved throughout the entire hierarchy. This can result in an inefficient utilization of boundary triangles and pose limitations on simplification, especially for very large models. Although it is possible to simplify the mesh of an entire model to eliminate the boundary unchanging problem, this approach may not be feasible for very large models due to memory limitations. In such cases, the block method can offer a more efficient and scalable solution, allowing for parallel execution and addressing the challenges associated with preserving boundaries in a hierarchical simplification process.



(a) Block shares the same origin as the model

(b) The center of block is the model origin

Figure 3.3: In the 2D schematic diagram of the block selection process, the orange dotted line represents the selected block including 4×4 C_L of M_L. After simplification, the unit of the next LOD is constructed by splitting simplified meshes into 2×2 C_L, where $(0, 0)$ is the origin of the model. The two different block selection methods: (a) The bottom-left block shares the same origin as the model. (b) The center of the the bottom-left block is the model origin.

Different block selection methods lead to different model approximation results. The 2D schematic of the first block selection method is shown in Figure 3.3 (a), in a three-dimensional case, the origin of the model is used as the center of the bottom-left block including $n \times n \times n$ cubes. If the bottom-left block origin coincides with the model origin, the mesh belonging to borders remains unchanged during the mesh simplification of different LODs, resulting in mesh

aggregation at the edge of the block (Figure 3.4 (b_2)).

The second selection method, as illustrated in Figure 3.3 (b), centers the bottom-left block around the origin of the model. With this approach, the boundary of the block at the current LOD level becomes the interior of the block in the next LOD. Consequently, the boundary meshes of the current level will undergo simplification in the subsequent LODs, ensuring that they do not remain unchanged over multiple LODs. Therefore, this block selection method prevents the accumulation of dense triangles at the boundary of the block, addressing potential issues related to boundary preservation during mesh simplification.

The comparison of the two LOD selection methods is listed in Figure 3.4, under the same block size selection, in Figure 3.4 (b_2), there are triangles accumulated at the cube boundary during the mesh simplification, but using the second block selection method can greatly reduce the accumulated triangles (Figure 3.4 (a_2)) at the border of the cube, which can improve the rendering efficiency while improving the result of mesh simplification. So we choose to use the second block selection method to perform mesh simplification.

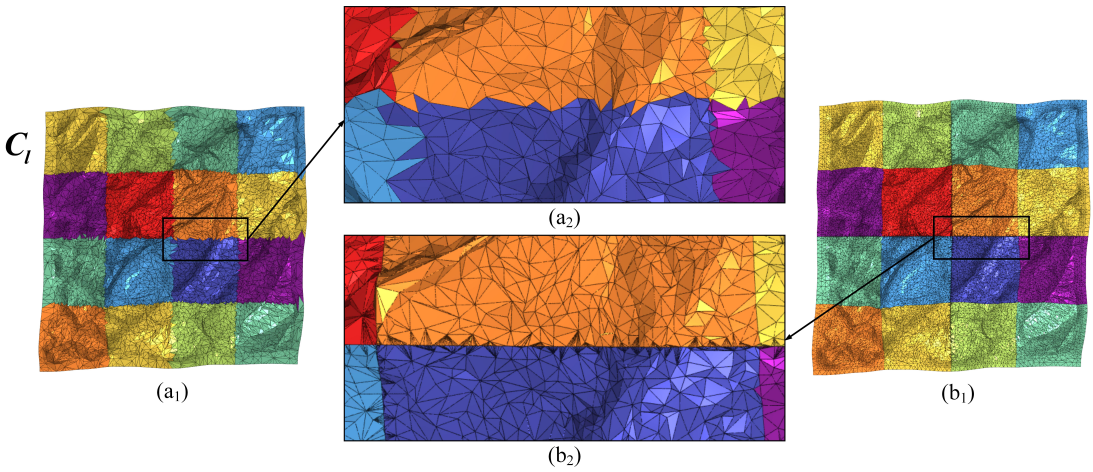


Figure 3.4: LOD construction results based on different block selection methods, (a_1) based on the first method, (b_1) based on the second method. One block contains $4 \times 4 \times 4$ cubes. (a_2) and (b_2) are partially enlarged pictures of the border of the cube.

3.2.5 Hierarchical level of detail generation

After the cube-based simplification, the resulting simplified meshes are split into $2 \times 2 \times 2$ cubes (for example, the green cubes in Figure 3.3 (b)) at a time to build a parent cube C_{L-1} (the red cube in Figure 3.3 (b)), which is equivalent to splitting the Euclidean space according to the axis-aligned regular grid G_l of size 2^{-l} centered at the origin. For each level $0 \leq l < L$ in the multi-resolution to be constructed, a cube C_l of the grid at level $l \geq 0$, we denote by $P(C_l)$, the only cube in the grid at level $l-1$ that contains C_l.

The partitions of the lower resolutions triangle set T_l $(0 \leq l < L)$ combined with cube-based simplification are then built in a recursive way:

$$T_l = \bigcup_{C_l \in G_l} T(C_l),$$

where

$$T_l \simeq \bigcup_{C_{l+1} \in G_{l+1},\ P(C_{l+1})=C_l} P(T(C_{l+1})).$$

Note that we write $\simeq$ instead of $=$ above because triangles that get collapsed by P are removed to yield T_l. Note also that in general, this partition may slightly differ from the one that would be obtained by splitting T_l according to the grid at a level as for T_L, it is done by splitting the simplified meshes. It is important to do it this way, the child-parent dependencies (Figure 3.1(b)) can be obtained and stored directly by recording the vertex correspondence during mesh simplification and simplified mesh splitting. In the practical implementation, the correspondence between child and parent vertices is maintained and stored as a vertex attribute. The value of this attribute corresponds to the index of the parent vertex associated with its mesh cluster.

For each $T(C_l)$, we call $V(C_l)$ the subset of vertices in V_L that are referred to by some triangles in $T(C_l)$, we let $M(C_l) := (V(C_l), T(C_l))$, and we refer to $M(C_l)$ as the mesh cluster associated to the cube C_l.

Since we have not recut triangles along with the grids, some vertices in $V(C_l)$ may fall outside of C_l, but as a consequence of our simplification constraint in the previous subsection, it follows that:

$$\forall v \in V(C_l), \quad d(v, C_l) \leq \sigma_0 2^{-l}. \tag{3.3}$$

The process is repeated iteratively to generate a sequence of coarser LODs until the desired level of detail is reached. The pseudo-code of HLOD generation is as follows:

```
while (l = L and l ≥ 0){
    while (C_l in M_l){
        B_N ← SelectBlock²(n × n × n);
    }
    do in parallel
    while (B_n in B_N){
        M'(B_n) ← MeshSimplification(M(B_n));
        while (C_l in B_n){
            B' ← SelectBlock¹(2 × 2 × 2);
            C_{l-1} ← BuildNewCube(M'(B'));
        }
    }
    l − −;
```

},

where N represents the total number of blocks, SelectBlock^1() represents applying the first block selection (Figure 3.2 (a)) method to build the block, and SelectBlock^2() uses the second block selection method (Figure 3.2 (b)). BuildNewCube() is the process of splitting the simplified meshes into $2 \times 2 \times 2$ cubes, with the goal of building a new cube that belongs to the coarser LOD. Therefore, the selection of the value for n is frequently a multiple of 2, which aligns more effectively with the mesh simplification process.

3.2.6 HLOD tree generation

The multi-resolution model construction process generates an octree, and each leaf node of the octree represents a cube C_l. C_l is marked by its i, j, k coordinates at this $level$, these $(level, i, j, k)$ are called logical coordinates (Figure 3.5), i, j and k vary between 0 and $2^{level} - 1$ with $(0, 0, 0)$ being the lower-left corner. A cube contains a mesh cluster that includes indices, vertex positions, normals, texture coordinates, colors, parent-child remaps, or any other data for rendering and representing the 3D model. The root cube $(0, 0, 0, 0)$ contains the data corresponding to the whole model at the coarsest resolution, and each level of the octree represents a higher resolution of the model. The cubes at each level of the octree contain only a part of the model, and the size of the cube decreases as the level increases, resulting in a higher level of detail.

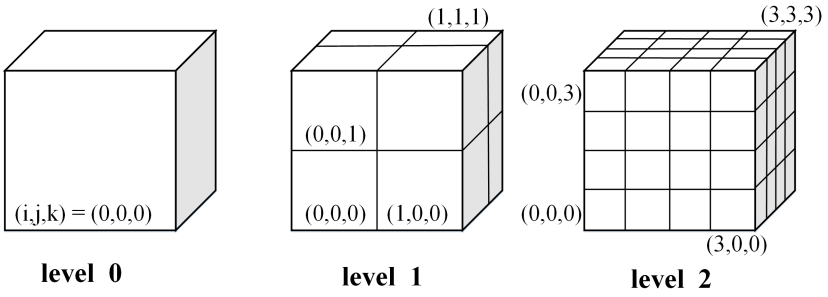


Figure 3.5: Logical cube coordinate system $(level, i, j, k)$, where $0 \leq i \leq 2^{level} - 1$, $0 \leq j \leq 2^{level} - 1$, $0 \leq k \leq 2^{level} - 1$.

In addition to the level and index of each cube in the octree, we also set physical coordinates for each cube C_l (Figure 3.6). The (O_x, O_y, O_z) is the coordinates of the lower-left corner of the cube at level $l = 0$ in a fixed reference frame. The physical coordinates correspond to the actual model and represent both the position and dimensions of the cube within the 3D world space. The size of the cube is in some length unit (e.g. meter) and decided by the size of the model (L_x: the max length in the x-axis direction, L_y: the max length in the y-axis direction, L_z: the max length in the z-axis direction). We take $L_c = max(L_x, L_y, L_z)$ as the length of C_0. By assigning physical coordinates to each cube, the HLOD representation becomes more versatile and can be used for various purposes such as computing the distance from the viewpoint to the cube.

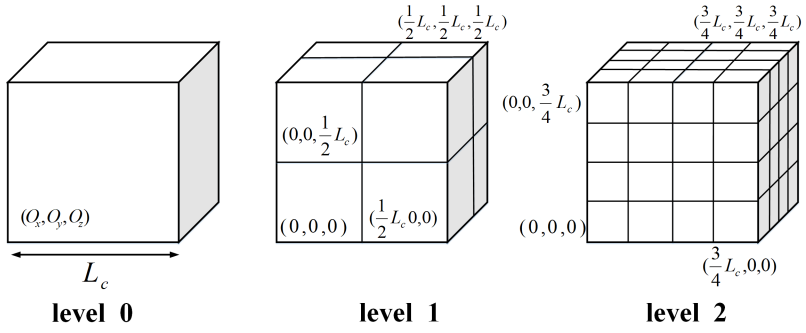


Figure 3.6: Physical cube coordinate system (O_x, O_y, O_z), the initial coordinate is $(O_x, O_y, O_z) = (min_x, min_y, min_z)$, where $O_x = min_x + L_c/2^{level}$, $O_y = min_y + L_c/2^{level}$, $O_z = min_z + L_c/2^{level}$.

The logical coordinates are used to identify the position of a node in the octree hierarchy, which can be used to traverse the octree with its child-parent cube relationship. The physical coordinates correspond with real data because the mesh partition of M_L is based on the barycenter of the triangle, the physical boundary of data might be smaller or bigger than its associated logical cube. These two coordinate systems are beneficial for us to realize block-based mesh

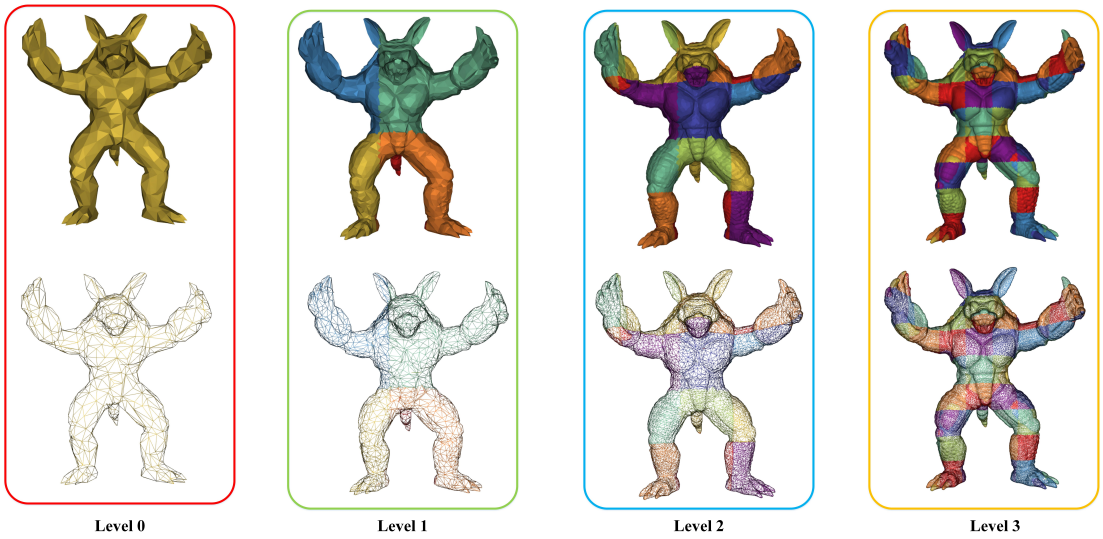


Figure 3.7: Multi-resolution model structure of Armadillo.

simplification, multi-resolution model construction, and octree-based real-time view-dependent visualization. The independence of data associated with each cube C_l allows for parallel construction, which can significantly improve the efficiency of the HLOD construction process.

The multi-resolution model of the Armadillo is illustrated in Figure 3.7. To visualize the relationship between child and parent cubes and to represent the varying levels of detail (LOD)

within the multi-resolution model, each cube in different LODs is colorized differently.

3.2.7 Data structure of multi-resolution model

The proposed multi-resolution scheme maintains a static *VertexBuffer* holding vertex attributes, and an *IndexBuffer* holding 3 indices per triangle face. These data structures, as detailed in Table 1, are stored on the GPU using sequential buffers. Our data structure uses a total of $4t + 31v$ or $4t + 36v$ bytes, where v and t are the numbers of vertices and indices of the multi-resolution model.

Multi-resolution models can be memory-intensive. Nevertheless, our method minimizes the storage size of the multi-resolution structure through simple vertex correspondence. Assuming the input model has p vertices and q indices, the simplification ratio $1/s$, which is between the target number of indices and the original number of indices. We can have the number of vertices and indices for each LOD as follows:

$$M_L \to (p, q), M_{L-1} \to (\frac{1}{s}p, \frac{1}{s}q), M_{L-2} \to (\frac{1}{s^2}p, \frac{1}{s^2}q), \cdots . \tag{3.4}$$

So the number of vertices v for the multi-resolution model can be calculated as follows:

$$v = p + \frac{1}{s}p + \frac{1}{s^2}p + \cdots = p\sum_{k=0}^{\infty}(\frac{1}{s})^k = \frac{s}{s-1}p. \tag{3.5}$$

Similarly, the number of vertices t can be calculated as $t = \frac{s}{s-1}q$. This implies that the memory requirements of our method are less than twice that of the original model.

Table 3.1: Data structure and buffer structure of multi-resolution model

Buffers	Elements	Memory (bytes)	Memory after quantization (bytes)
IndexBuffer	Index	$4t$	$2t$
VertexBuffer	Position	$12v$	$6v$
	Normal	$12v$	$4v$
	Color / Texcoord	$3v/8v$	$3v/4v$
	Child_Parent_Map	$4v$	$2v$
Total		$4t + 31v/4t + 36v$	$2t + 15v/2t + 16v$

When the memory capacity of the system is sufficient to accommodate the model, the multi-resolution representation can be loaded into the main memory at one time, which can lead to better interactive performance. When the size of the model exceeds the available memory, loading the entire multi-resolution model into the main memory may not be possible, our approach can be extended to an out-of-core method. The multi-resolution model data can be stored on the disk, and the geometry data of the required cube will be streamed when needed, while the parent cube needs to be loaded at the same time in order to achieve the popping-free and crack-free visualization result. More information regarding out-of-core construction and rendering will be provided in section 3.3 and section 4.3.

3.2.8 Vertex quantization

Vertex quantization (Figure 3.8) is a technique employed to decrease the storage demands of 3D models by representing vertex attributes as integer or half-precision float-point values rather than floating-point values. This approach is designed to achieve a reduction in storage space without significantly compromising the visual quality of the model. The reduced data is valuable for efficiently storing meshes in CPU/GPU memory and facilitating the faster transfer of meshes from RAM to VRAM [Mag+15] [LCL10] [Cal02]. Vertex quantization is achieved by scaling the vertex attributes (such as position, normal, and color) to fit within a fixed range and then rounding the resulting values to the nearest integer.

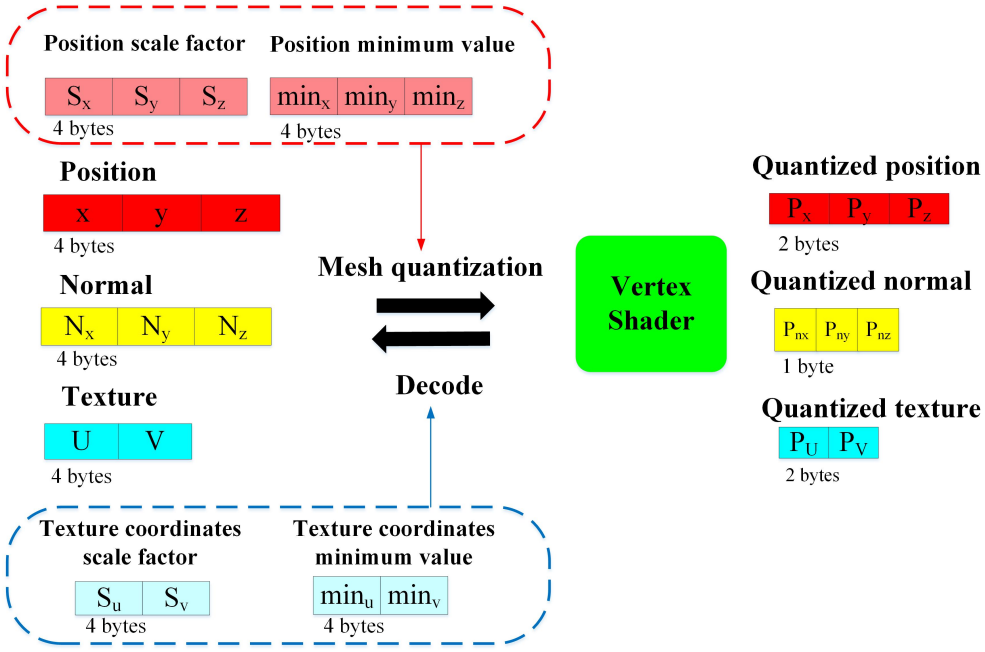


Figure 3.8: Vertex attributes coding and decoding of quantization process.

The steps of vertex quantization are as follows:

(1) Choose a quantization range, which is typically determined by the precision requirements of the application and the available memory.

(2) Scale the vertex attribute value to fit within the quantization range.

(3) Round the scaled attribute value to the nearest integer to obtain the quantized value.

It is important to choose an appropriate quantization range and scaling factor to ensure that the level of precision is sufficient for the application. Additionally, we need to avoid quantization artifacts, such as value distortion or vertex popping, which can occur when the quantized vertices are rendered.

Position quantization

Position quantization involves the local quantization of its value within mesh clusters that have been generated as part of the multi-resolution construction process. For each mesh cluster $M(C_l)$ of different LOD M_l, 16-bit quantization for vertex position (32-bit float) (x, y, z) is performed. The position quantization steps are as follows:

(1) Calculate the maximum position value (Max_x, Max_y, Max_z), the minimum position value (Min_x, Min_y, Min_z) of current mesh partition $M(C_l)$.

(2) Calculate the scale factor of the current mesh cluster (S_x, S_y, S_z).

$$S_x = 65535/(Max_x - Min_x),$$
$$S_y = 65535/(Max_y - Min_y),$$
$$S_z = 65535/(Max_z - Min_z).$$

(3) Combine the scale factor and the minimum value of this partition to get the quantized position (P_x, P_y, P_z) in the range of [0, 65535].

$$P_x = round(S_x * (x - Min_x)),$$
$$P_y = round(S_y * (y - Min_y)),$$
$$P_z = round(S_z * (z - Min_z)).$$

(4) Save the minimum value and the scale factor associated with the cluster, which will be used in the decoding phase. For the texture coordinates, the quantization process is similar.

Normal quantization

In the case of normal attributes represented as (n_x, n_y, n_z), An 8-bit quantization method is employed. The steps are as follows:

(1) Normalize the normal value by scaling them to fit within the range of $[-1.0, 1.0]$.

(2) Quantize the normal value into an 8-bit unsigned integer by following:

$$P_{nx} = \begin{cases} int(n_x * 127.5), n_x \geq 0, \\ int(n_x * 127.5 - 1.0), n_x < 0. \end{cases}$$

$$P_{ny} = \begin{cases} int(n_y * 127.5), n_y \geq 0, \\ int(n_y * 127.5 - 1.0), n_y < 0. \end{cases}$$

$$P_{nz} = \begin{cases} int(n_z * 127.5), n_z \geq 0, \\ int(n_z * 127.5 - 1.0), n_z < 0. \end{cases}$$

(3) Pack the (P_{nx}, P_{ny}, P_{nz}) into 32-bit unsigned integer.

Decoding

After transmitting the geometry data from the main memory to graphic card memory in encoded form, i.e., a 16-bit integer for the position, a 16-bit integer for texture coordinate, and an 8-bit integer for normal. The scale factor and minimum value of different partitions are also loaded into VRAM. The decoding process is conducted in the vertex shader, for example, the position decoding process is as follows:

$$\begin{aligned} x &= P_x/S_x + Min_x, \\ y &= P_y/S_y + Min_y, \\ z &= P_z/S_z + Min_z. \end{aligned}$$

The normal decoding is as follows:

$$n_x = \begin{cases} P_{nx}/127.5, P_{nx} \geq 0, \\ (P_{nx} + 1.0)/127.5, P_{nx} < 0. \end{cases}$$

$$n_y = \begin{cases} P_{ny}/127.5, P_{ny} \geq 0, \\ (P_{ny} + 1.0)/127.5, P_{ny} < 0. \end{cases}$$

$$n_z = \begin{cases} P_{nz}/127.5, P_{nz} \geq 0, \\ (P_{nz} + 1.0)/127.5, P_{nz} < 0. \end{cases}$$

The decoding process is nicely incorporated with the standard rendering pipeline. The only major overhead is the scale factor and minimum value per cluster at rendering time, which can be considered negligible. It is relatively inexpensive in terms of computation and memory usage, especially compared to the benefits of using quantization to reduce the size of the data being rendered.

It is often beneficial to reduce the amount of memory required to store the mesh, either for data fetching or to optimize memory bandwidth. For our HLOD scheme, memory optimization can be done by vertex quantization. Under the assumption that the HLOD structure comprises v vertices and t indices: the 32-bit float normal value can be encoded to an 8-bit integer and combined to one 32-bit integer, resulting in $4v$ bytes; for the position, 16-bit normalized integers are encoded the position relative to the physical coordinates of a cube, its memory usage can be reduced to $6v$; the number of indices per cube can be restricted to maximal value 2^{16}, which reduce its memory usage from $4t$ to $2t$; with the number limitation of the index, the child_parent_map (the index of its parent vertex) can be stored as a 16-bit integer, resulting in $2v$ memory usage. The final reduced total memory usage will be $2t + 16v$ or $2t + 15v$ bytes.

3.2.9 Overall implementation

After getting the data from the original 3D model, the input model is partitioned to build the highest resolution LOD (M_L) based on section 3.2.3. Then block-based mesh simplification is implemented recursively to build new LODs until the preset highest level L is reached. In order to accelerate HLOD construction, parallel mesh simplification and parallel LOD construction are utilized. For each LOD, each thread will perform parallel simplification on the selected block, the simplified meshes are used to build the parent cube while simultaneously updating child-parent dependency of each vertex.

Algorithm 1 Multi-resolution construction (In-core)

Input : The highest multi-resolution level: L;
The full-resolution model: M_{origin};
The cube-block size: n;
M_{origin} ←ReadfromFile();
M_L ← MeshPartition(M_{origin}, L);
while $l = L$ *and* $l \geq 0$ **do**
  while C_l *in* M_l **do**
    $B_N \leftarrow \text{SelectBlock}^1(n \times n \times n)$;
  end
  do in parallel
  while B_n *in* B_N **do**
    $M'(B_n) \leftarrow$ MeshSimplification($M(B_n)$);
    while C_l *in* B_n **do**
      $B' \leftarrow \text{SelectBlock}^2(2 \times 2 \times 2)$;
      $C_{l-1} \leftarrow$ BuildNewCube($M'(B')$);
    end
  end
  $l - -$;
end
Output : The multi-resolution model: $M_L \sim M_0$.

The pseudocode of the construction process is in Algorithm 1: The function ReadfromFile() reads data from the standard 3D model files, such as PLY and OBJ; MeshPartition() constructs full-resolution LOD based on the preset maximum LOD level L and input model M_{origin}; MeshSimplification() simplifies the mesh clusters of the current block based on QEM-based edge collapse; BuildNewCube() constructs a parent cube by splitting the simplified meshes into $2 \times 2 \times 2$ cubes, and also generates a compact index and vertex buffer; SelectBlock^1() and SelectBlock^2() correspond to different block selection methods in section 3.2.4.

3.3 Out-of-core multi-resolution model construction

3.3.1 Octree-based Out-of-core Mesh data structure

For a gigantic-sized model, it is difficult to read its geometric data and perform mesh simplification due to the constraint of limited core memory. We have designed an out-of-core multi-resolution model construction pipeline, enabling the efficient handling of large 3D mesh models that cannot be accommodated entirely in memory. With this approach, only the necessary portions of the model are loaded into memory as required, while the remaining mesh data is stored on disk.

The Octree-based Out-of-core Mesh data structure (OOM) is based on a hierarchical geometric partition of the data set with no vertex replication and consistent vertex indexing between cubes that share a reference to the same vertex. During the multi-resolution construction, only the hierarchical structure of the octree is maintained in the main memory: each octree leaf (cube) holds the external memory address of the corresponding portion of the model. Even for very large models, it is not memory-consuming to maintain the whole octree in the main memory due to its relatively small size.

OOM leaf node: Each leaf l of the octree corresponds to a cube C, which includes the logical coordinates of the cube, the number of vertices, the number of indices, the maximum and minimum coordinates of the cube, and the file path. The file path stores a pointer to a secondary memory which contains:

- Vertices: all the vertices contained in the cube; for each vertex v, it has the position, normal, texture coordinate/color, and child-parent dependency.
- Indices: all the indices contained in the cube.

The OOM data structure encoding each OOM node in memory and hard disk is as follows:

```
OOMNode {
   unsigned int   ijk;   // The logical coordinates of the cube
   unsigned int   nv;    // The number of vertices of the cube
   unsigned int   ni;    // The number of indices of the cube
   float  max[3];        // The maximum position value of the cube
   float  min[3];        // The minimum position value of the cube
   string file;          // The file path
};
DiskNode {
   float* position;
   float* normal;
   float* uv;
   char*  color;
   unsigned int*  child_parent_remap;
```

```
    unsigned int*  index;
};
```

These structures allow us to load data into the main memory and modify (update) any parts of the data. The cube data file on disk is stored in binary format, which offers several benefits, including reduced file size and faster IO operations. By traversing the OOM tree and iteratively loading, updating, and saving nodes, we can apply geometric algorithms that rely on local updates to very large meshes.

Traversal

In order to apply geometric algorithms over an OOM structure, we define a traversal strategy such that all the vertices and triangles are loaded into the main memory to be operated. Loading only a leaf at a time does not allow for getting full information on the associated mesh portion and operating the triangles in an efficient way. For further mesh simplification and LOD construction, the OOM traversal follows these rules:

- Start from the root cube C_0 down to the leaf nodes of the octree.
- Load all the leaves selected for the current operation.
- Load the minimal set of leaves such as the number of vertices and the number of indices.

A geometric algorithm can traverse the OOM tree to choose the different leaves depending on the characteristics of the processing.

Loading leaves

Loading a selected set of leaves $S = \{l_1, l_2, l_3, ...\}$ into the main memory means reconstructing indexed mesh representation from the selected loaded leaf nodes. This operation will generate the new index presentation to all the loaded vertices. Vertices re-indexing can be done in linear time based on the hash-coded of each vertex.

We first hash each vertex in the selected leaf and assign it a unique identifier to form a redundant-free vertex buffer. This process will be accompanied by generating a remap table from the input vertex buffer to the compact vertex buffer. Then, we allocate memory space for the compact index buffer and generate them based on the remap table.

Once the new indexed mesh representation is generated, geometric algorithms such as mesh simplification and LOD construction can be applied efficiently to the selected set of leaves in the main memory.

Saving leaves

After simplification or generation of a new mesh for the selected set of leaf nodes $S = \{l_1, l_2, l_3, ...\}$, the data has to be written back to the corresponding files on secondary memory. This is done

by mapping the modified or generated data to the corresponding leaf files and writing the data to those files. The modification is then considered permanent and will be used for further LOD construction or visualization of the model:

- Update the information of existing leaf nodes, such as parent-child correspondence. This operation only appends the new attributes to the existing original leaf node file.
- Generate a new leaf node with the compact vertex buffer and index buffer without redundancy. Write the optimized buffers to the appropriate file on secondary memory. Subsequently, free the data stored in memory to avoid using unnecessary memory resources.

3.3.2 Construction of Octree-based Out-of-core Mesh data structure

The input mesh can be an indexed structure, but it also can be unindexed triangles, such as triangle soups. We describe OOM construction with the worst-case input, but if the input is indexed meshes, some steps can be skipped. The OOM based on the input meshes can be constructed in two steps:

(1) The initial construction phase consists of defining the structure of the OOM with a preset maximum level L of a multi-resolution model. Subsequently, a thorough scan of all meshes is performed to obtain the maximum and minimum position values within the model. The second scan of triangles assigns all the triangles into cubes based on the barycenter of the triangle, the triangles will not be cut during the mesh partition. This results in the generation of the raw OOM. The complexity of the partitioning algorithm based on octree is $O(nlog(n))$.

(2) The second phase is based on the raw OOM, whose meshes of each cube are unindexed. The process begins by generating a compact vertex buffer for each cube, followed by updating the indices of each vertex within that cube. Subsequently, after the reindex of meshes of each cube, all the data will be written into an independent binary file.

In order to apply mesh simplification and rendering over an OOM, the visiting strategy is based on the traversal rule mentioned previously, when this cube is required, its corresponding data is loaded into the main memory, such as the vertex attributes, indices, and the parameters of the cube.

The construction of OOM is accompanied by the generation of the bottom level of the octree structure, the whole octree structure will be written into second memory during the multi-resolution model construction, the structure of the OOM tree is as follows:

```
OOMTree{
    /* LOD information */
    unsigned int   l_total;            // The level of LOD
    unsigned int   l;             // Current LOD level
    unsigned int   num_cube;      // The total number of cubes
```

```
    int     grid_size;  // The maximum number of cubes along one axis
    float   min[3];     // The minimum position value of the model
    float   max[3];     // The maximum position value of the model
    float   length[3];  // The length of the model

    /* The information of each cube in the current LOD */
    /* The first cube */
    int      i, j, k;          // The logical coordinates of the cube
    unsigned int64  ijk_64;
    // 64-bit encoding of logical coordinates
    unsigned int  num_vertex;   // The number of vertices of the cube
    unsigned int  num_index;    // The number of indices of the cube
    float    min[3];        // The maximum position value of the cube
    float    max[3];        // The minimum position value of the cube
    string   file_path;     // The data file related to the cube
    /* The second cube */
    ...
}
```

Maintaining the OOM tree structure in the main memory does not lead to a memory bottleneck since its memory size is relatively small compared to the vast size of the mesh model. To give an example, the multi-resolution structure of St. Matthew consumes 12.4 GB of storage space, and the size of the OOM tree is only 3.4 MB. The algorithm's complexity is closely tied to the octree structure, with time and space complexities scaling logarithmically in relation to the size of the input data.

Along with the construction of the OOM from the original data, we obtain the highest resolution LOD, where each data file serves as the fundamental unit, denoted as C_L, within this LOD level. The logical coordinate and physical coordinate of each C_L are distributed as part of the construction process, forming the bottom level of the OOM tree.

3.3.3 Out-of-core simplification and LOD construction

During the out-of-core simplification and LOD construction, the overall mesh simplification process is similar to the in-core algorithm, in order to select the cube block to simplify meshes and build the parent cube, the OOM tree is maintained in the main memory all the time. The whole construction process is shown in Figure 3.9.

Based on the highest resolution LOD and the OOM tree obtained in section 3.3.2. We select $n \times n \times n$ C_L to build the blocks which are the execution unit for mesh simplification. The block selection method (Figure 3.3 (b)) is used to form the blocks in order to achieve a better mesh approximation result. After the selection of blocks, the cube information (such as the numer of vertex and index) of each block is obtained by searching the OOM tree according to the logical coordinates of each cube. Memory allocation for the data associated with each block

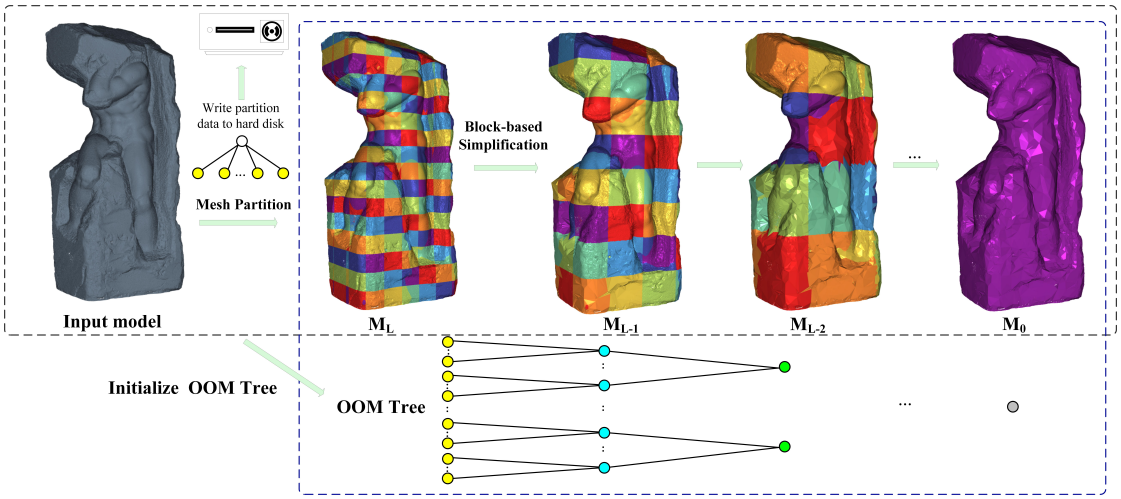


Figure 3.9: Out-of-core multi-resolution construction, the upper part of the picture is the overall construction process, where different colors represent the different mesh partition C_l of the current LOD level. The part in the purple dot box is the OOM tree structure. Different dots represent the leaves of the OOM tree, and they are related to the different cubes.

is determined by the total number of vertex and index, then loading the mesh data related to the currently selected block from the disk into the memory. Based on the preset quadric error ϵ, mesh simplification based on edge collapse is performed in each block separately, and the vertex remaps are initialized to further child-parent relationship establishment.

After obtaining the simplified meshes, we need to initialize memory space for the parent cube to allocate geometry data first and then split the simplified geometry into $2 \times 2 \times 2$ cubes to build a new cube C_{L-1}, which is the parent cube $C_{L-1} = P(C_L)$. Once the initial geometric data for each parent cube is obtained from the simplified data, compact and non-redundant vertex attribute buffers and index buffers are generated. For each vertex, the child-parent dependency information is updated and written into the OOM leaf file based on the vertex remap of mesh simplification and parent cube construction. The independence of each OOM leaf makes it easy to implement in parallel, which can accelerate the out-of-core multi-resolution construction.

3.3.4 Overall implementation

The out-of-core multi-resolution model construction process is similar to the in-core method, except for the data streaming part including cube data file reading and writing. The entire model is initially divided into axis-aligned cubes by employing an octree structure. This partitioning leads to the creation of the highest-resolution LOD and initiates the OOM tree structure. During the multi-resolution construction process, the subparts are grouped hierarchically and are simplified together. The OOM tree data structure file is also updated, which can be used for octree traversal further for other applications, such as the subdivision of real-time rendering.

Algorithm 2 Out-of-core multi-resolution model construction

Input : The highest multi-resolution level: L;
The full-resolution model: M_{origin};
The cube-block size: n;
$M_{origin} \leftarrow$ ReadfromModel();
$M_L \leftarrow$ MeshPartition(M_{origin}, L);
WriteCubeDataToDisk(C_L);
while $l = L$ *and* $l \geq 0$ **do**
  while C_l *in* M_l **do**
    $B_N \leftarrow$ SelectBlock$^1(n \times n \times n)$;
  end
  do in parallel
  while B_n *in* B_N **do**
    ReadFromDisk(B_n);
    $M'(B_n) \leftarrow$ MeshSimplification($M(B_n)$);
    while C_l *in* B_n **do**
      $B' \leftarrow$ SelectBlock$^2(2 \times 2 \times 2)$;
      $C_{l-1} \leftarrow$ BuildNewCube($M'(B')$);
      WriteCubeDataToDisk(C_{l-1});
      UpdateCubeDataToDisk(C_l);
    end
  end
  $l - -$;
end
WriteOMMTreeToDisk(OOM tree);
Output : The multi-resolution model: $M_L \sim M_0$ file, OOM tree file.

In the pseudocode of Algorithm 2, functions such as ReadfromFile(), MeshPartition(), MeshSimplification(), SelectBlock^1(), SelectBlock^2(), and BuildNewCube() perform functions similar to those in the in-core multi-resolution construction process. However, certain functions have specific purposes. WriteCubeDataToDisk() is responsible for writing the geometry data related to C_l to the disk. UpdateCubeDataToDisk() appends the child-parent remap information to the cube data file, and WriteOMMTreeToDisk() is utilized for saving the OOM tree to the disk.

3.4 Results

The multi-resolution model construction was done on a desktop machine (without discrete GPU) equipped with an Intel UHD Graphics 630 and Intel i7-8700 CPU (16GB). The software library and preprocessing codes have been implemented on Linux using C++.

The in-core test result is based on 11 models: two fractals models, Happy Buddha, Egyptian

tomb, Egyptian wall painting, David at 1mm resolution, Terrain ($1km \times 1km$), Night sculpture, Gurnah temple mural, Lucy and Egyptian temple mural. The out-of-core test result is based on St. Matthew and Atlas. The terrain is generated by ourselves (the terrain reconstruction pipeline will be introduced in Chapter 5), Happy Buddha, Lucy, David, Night, St. Matthew, and Atlas are from Stanford University [Lev+00]. The Egyptian tomb, Egyptian wall painting, Gurnah temple mural, and Egyptian temple mural are from Laboratoire d'Archéologie Moléculaire et Structurale of Sorbonne Université [Alf+18].

3.4.1 In-core multi-resolution model construction

Table 3.2 shows the quantitative results of our in-core multi-resolution construction process, the number of vertices and indices of the original model M_L, the coarsest LOD level M_0 and the multi-resolution model are listed. In our implementation, the QEM-based edge collapse mesh simplification is executed on each selected block, a block consists of $4 \times 4 \times 4$ cubes, and the number of target vertices for each block grid simplification is a quarter of the original number. For textured models, the extended QEM mesh simplification is used. Our mesh simplification code implementation is based on [Kap22].

Table 3.2: Quantitative results of multi-resolution model construction

Model	M_L		M_0		Multi-resolution Model		level L	time (s)
	n_v	n_t	n_v	n_t	n_v	n_t		
Fractal_1	498156	2999997	124010	736839	1119908	5957673	4	1 .168
Happy Buddha	543652	3263148	2111	12732	770319	4344996	4	1.065
Egyptian tomb	1000175	5986128	222493	799761	2156105	9895032	4	1.950
Fractal_2	1416596	8507589	7312	32676	1983953	11328471	4	2.724
Terrain	7249423	43495242	449911	2698212	11190546	65179494	5	11.625
Night	11050083	64711983	847063	4045962	18664770	101075478	6	15.579
Lucy	14027872	84167226	663812	3982755	22003224	126401145	6	24.082
Egyptian wall painting	22548642	135170106	2606598	7637196	45155593	208790802	6	28.675
Gurnah temple mural	28364417	164999997	533415	3002493	39692236	228610629	6	27.187
David (1mm)	28184526	168691029	177595	652767	40469139	226659180	7	49.717
Egyptian temple mural	30001155	180000000	356265	2018868	43632901	247109496	7	53.238

The construction process time includes mesh partition of the original model, block-based mesh simplification, and LOD construction. We built hierarchy LODs for each model, and the construction time is related to the size and complexity of the models and whether they contain other attributes such as texture coordinates. As the number of triangles increases, the construction time increases accordingly. We designed CPU multi-core parallel mesh simplification and LOD construction implementation based on OpenMP [Cha+01], which speeds up multi-resolution model construction. The quantitative results show that n_v of the multi-resolution model is close to $4/3n_v$ of the original model, and the memory required by our multi-resolution structure is less than or around twice that of the original model, which is memory efficient.

Different types of models are chosen, such as the colored fractal model, it is a suitable choice for testing the multi-resolution construction algorithm because of its complex geometry and intricate details. The algorithm's ability to preserve the original model's overall shape and geometric

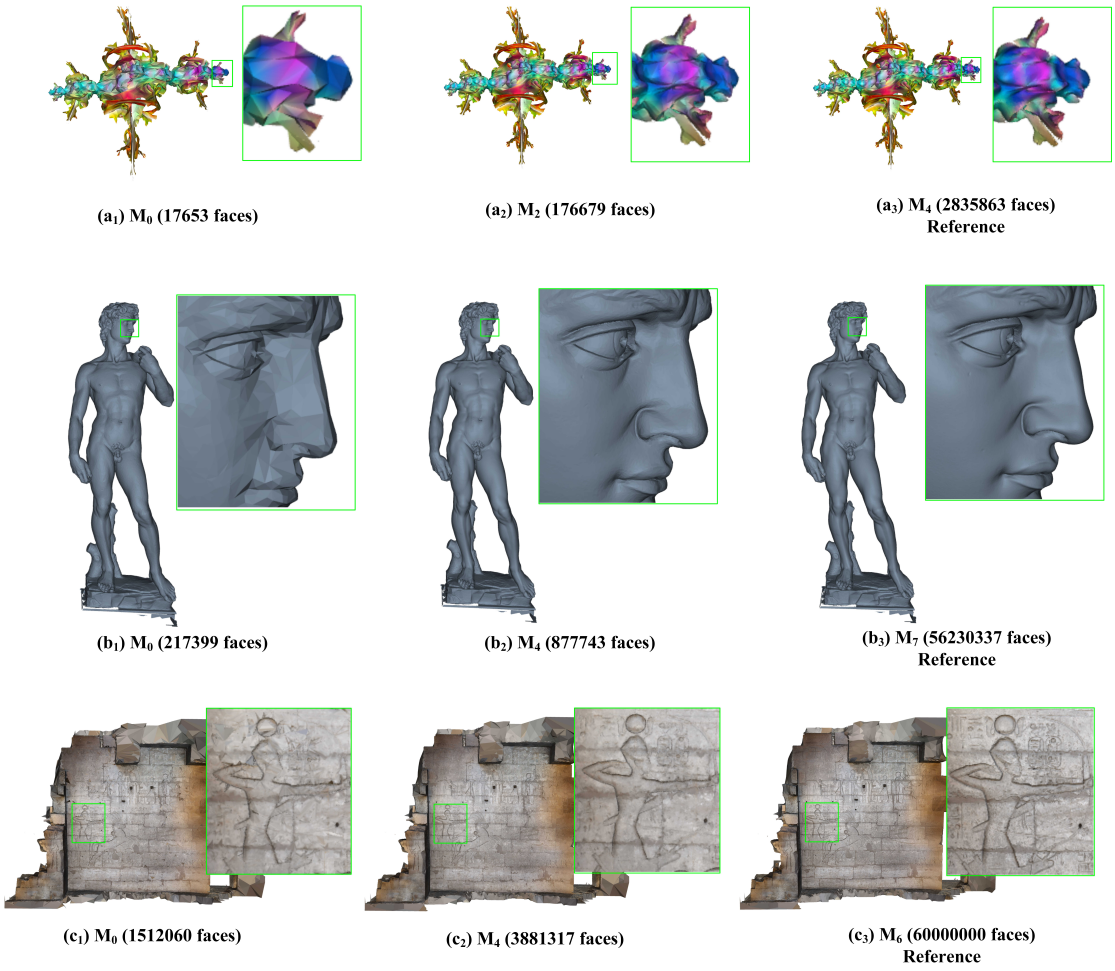


Figure 3.10: Rendering results of different LODs for different models: Fractal_2, David, and Egyptian temple. The last column is the rendering result of the full-resolution model.

features can be evaluated using this type of model. We also test the textured models, such as Egyptian tomb, Egyptian wall painting, and Egyptian temple mural, the texture coordinates information for different LODs are preserved. For the model with the color attribute, such as Fractal_2 and Gurnah temple mural, their color characteristics are maintained across LODs. For models with large planes such as Egyptian wall painting, Egyptian temple mural and Gurnah temple mural, the edge collapse error limit of two vertices on the plane is added to limit the large collapse of the plane vertex. In Figure 3.10, we showcase the rendering results of various LODs for different models, including detailed partial enlargements.

The mesh quantization is done after the multi-resolution construction, and the size comparison of the original model and quantized model is shown in Table 3.3. The model size has been reduced to at least 2/3 of the original model, where the Egyptian tomb, Egyptian wall painting,

Table 3.3: Original model and quantized model size comparison.

Model	Memory usage (MB)	
	Original model	Quantized model
Egyptian tomb	99.63	66.42
Lucy	965.91	775.73
Egyptian wall painting	2216.99	1490.03
Gurnah temple mural	1855.57	1492.63
David(1mm)	1991.17	1601.11
Egyptian temple mural	2972.98	1992.41

and Egyptian temple mural are the models with textures.

3.4.2 Out-of-core multi-resolution model construction

The out-of-core multi-resolution construction was tested on St.Matthew and Atlas (Figure 3.11). Table 3.4 shows the quantitative results of the out-of-core resolution construction. During the block-based mesh simplification, $4 \times 4 \times 4$ cubes are chosen to do the simplification at one time, and the target number of the simplification is 1/4 of the original number. It can be seen from the table that the final number of vertices and indices is 4/3 of the input model. The out-of-core multi-resolution representation is stored in the second memory, and the size of the multi-resolution model is less than two times that of the original 3D model file with adding the child-parent dependency of each vertex.

Table 3.4: Quantitative results of out-of-core multi-resolution model construction

Model	M_L		M_0		Multi-resolution Model		size (GB)		time (s)	level L
	n_v	n_t	n_v	n_t	n_v	n_t	input	output		
St.Matthew	186984410	372767445	77040	454833	256373513	498588415	7.8	13.2	348.72	7
Atlas	255035497	507512682	87848	505545	353681293	682153837	10.7	18.1	520.31	7

The out-of-core construction time is much longer than our in-core algorithm, because the IO operations, the reading or writing of the mesh partition file is time-consuming. The out-of-core construction process can be accelerated by high-performance storage devices such as Solid State Drives (SSDs) to provide fast access to data stored on disk. However, our method only takes less than 10 minutes to build the multi-resolution structure of Matthew and Atlas by using a Hard Disk Drive (HDD).

3.4.3 Comparison

We compare the multi-resolution construction time of our method and Nexus [Vis20] by processing 11 models generating the same level of LODs. For this comparison, we utilize the module nxsbuild from this open-source library, which can create a batched multi-resolution 3D model structure. The comparison results are shown in Table 3.5. Nexus uses the VGC library [Vis17]

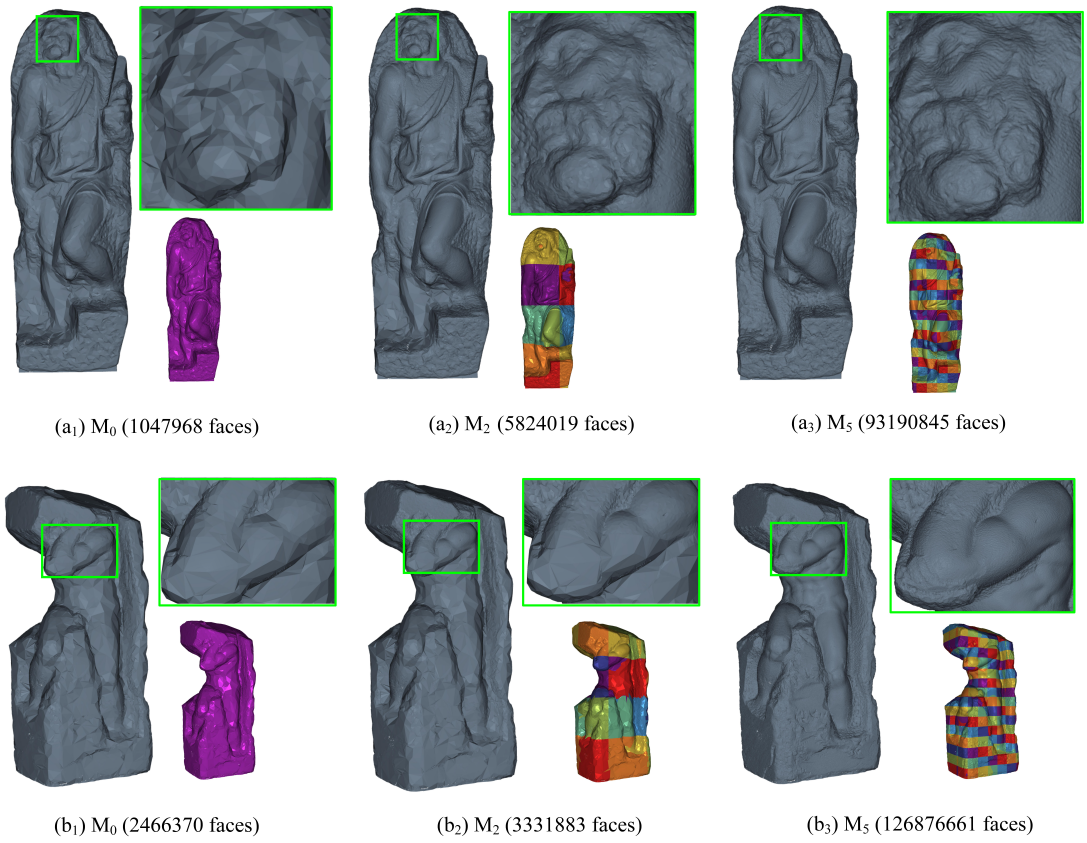


Figure 3.11: Rendering results of different LODs for St.Matthew and Atlas. The upper right corner of each picture is a partially enlarged image, and the lower right corner is the cube coloration based on the mesh partition.

simplification algorithm, which is the QEM-based edge collapse method. Our method can process about 1 million triangles per second per core, the construction time increases with the model complexity.

For the out-of-core construction of St. Mattew and Atlas, the proposed method took about 6 minutes to build the multi-resolution structure of St. Matthew, and 9 minutes of Atlas. We also tried to use Nexus to build the multi-resolution modes of the St. Mattew and Atlas models. Under the same computer configuration, Nexus could not complete the construction of St. Mattew, and its nxsbuild module crashed.

Our construction approach differs from the progressive buffer [SM05] in the method of constructing a multi-resolution hierarchy. Our method constructs a resolution hierarchy by grouping high-resolution clusters to build the next coarser level (Figure 3.12 left), the boundary meshes of blocks are not simplified at the current level but they will be simplified at the next level. Conversely, the progressive buffer builds the resolution hierarchy for each initial cluster independently (Figure 3.12 right). It loads adjacent clusters to simplify the boundary meshes of the

Table 3.5: Multi-resolution model construction time comparison between Nexus and our method.

Model	Preprocessing time (s)	
	Our Method	Nexus
$Fractal_1$	1.168	13.535
Happy Buddha	1.065	4.791
$Fractal_2$	2.724	5.037
Egyptian tomb	1.950	43.693
Terrain	11.625	64.969
Night	15.579	100.104
Lucy	24.082	152.520
Egyptian wall painting	28.675	435.392
Gurnah temple mural	27.187	324.912
David(1mm)	49.717	341.671
Egyptian temple mural	53.238	521.307

current cluster. This construction method may lead to scalability issues when the progressive buffer is applied to large-scale models, such as planet-sized models.

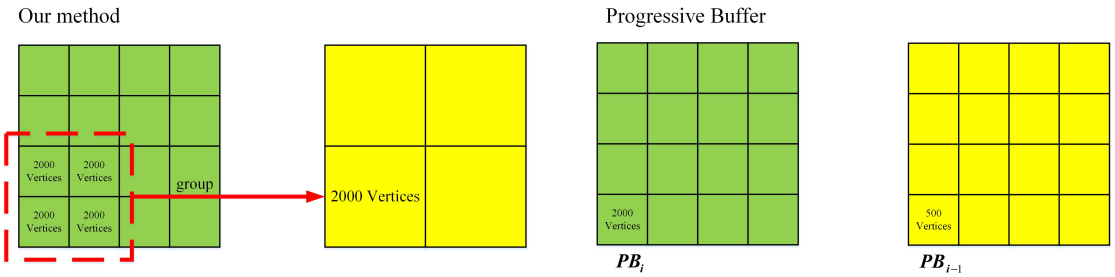


Figure 3.12: Comparison between our method and progressive buffer for resolution hierarchy construction.

By comparison with other methods, our in-core and out-of-core multi-resolution construction methods can generate hierarchy representations with different LODs for the different scale massive 3D model that contains only simple vertex dependency (child-parent relationship), which results in a time-efficient and memory-efficient construction process.

3.5 Conclusion

In this chapter, an efficient multi-resolution structure with vertex hierarchy is introduced, and the whole construction can be accelerated by parallel processing, it can be extended to out-of-core construction mode depending on the scale of the model. In the next chapter, the high-fidelity real-time rendering algorithm based on the multi-resolution structure will be discussed.

Chapter 4

Adaptive real-time rendering

4.1 Overview of adaptive real-time rendering

The real-time visualization of massive models is based on the multi-resolution model, which is generated in the construction stage, the octree-based hierarchy model is traversed coarse-to-fine to select the mesh clusters with the appropriate resolution based on the position of the viewpoint, vertex interpolation between child and parent is used to achieve crack and popping-free rendering. The overview of the view-dependent rendering is shown in Figure 4.1, it has the following contribution:

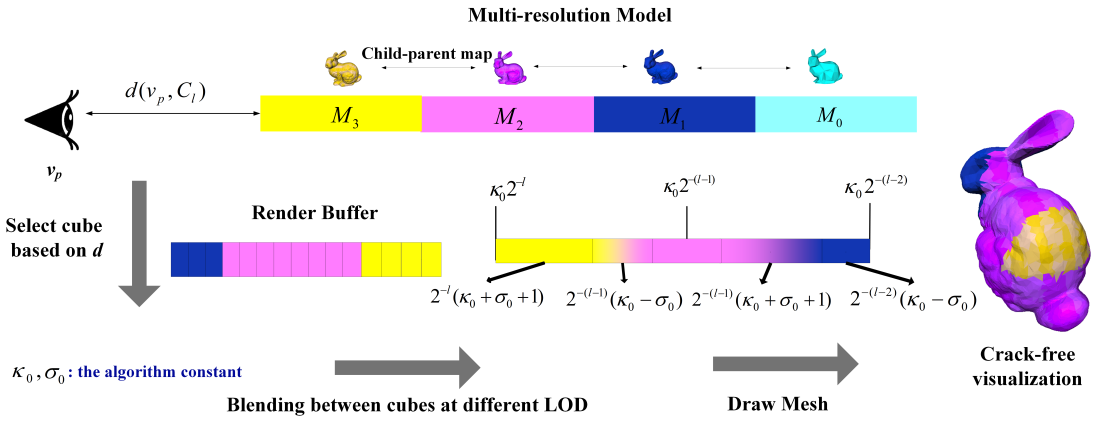


Figure 4.1: The overview of view-dependent rendering algorithm.

(1) *Vertex interpolation between different LODs.* In order to prevent the LOD popping and the boundary cracks, we use an algorithm that interpolates vertices between different LODs automatically. As simple as it may seem, the actual implementation of this (old) idea in a way that does not deteriorate performance or quality requires due care and also imposes

some (mild) constraints (σ_0) during the construction stage.

(2) *GPU-friendly static vertex and index buffers.* Our implementation uses the programmable nature of modern GPUs for the best performance. Vertex attributes are kept minimal (in particular we avoid the standard double vertex cost for interpolation) as we fetch index and vertex buffers (Shader Storage Buffer Object (SSBO) in our OpenGL implementation) directly from the shaders. One vertex buffer is dedicated to the child-parent correspondence between LODs.

4.2 Adaptive multi-resolution rendering

4.2.1 Distance-based subdivision

The multi-resolution model organized by octree is dynamically subdivided based on the current viewpoint, in order to provide more details near the viewpoint. The subdivision is only based on distance from the viewpoint to cubes. The distance-based rule is as follows:

Given a viewpoint v_p, a cube C_l is said to be suitable for visualization if either $l = L$ or

$$d(v_p, C_l) \geq \kappa_0 2^{-l}, \tag{4.1}$$

where $\kappa_0 > 0$ is a second constant in the model. If it is suitable for visualization and either $l = 0$ or its parent cube is not suitable for visualization, so C_{l-1} is subdivided because its distance to the viewpoint is less than $\kappa_0 2^{-(l-1)}$, where d is the maximum distance, the position of the viewpoint is (c_x, c_y, c_z), and the physical coordinates of C_l are (O_x, O_y, O_z):

$$\begin{aligned} &d(v_p, C_l) := max(min(|c_x - O_x|, |c_x - O_x - 2^{-l}|), \\ &min(|c_y - O_y|, |c_y - O_y - 2^{-l}|), min(|c_z - O_z|, |c_z - O_z - 2^{-l}|)). \end{aligned}$$

This distance calculation is valid when the viewpoint is outside of the cube, if the viewpoint is inside the cube, then $d(v_p, C_l) = 0$. For κ_0, increasing its value will result in a higher required LOD level, and the rendering result will exhibit a smaller screen space error. For instance, when a plane is rendered with a texture $S \times S$ pixels, we set the field of view angle to θ and the screen width in pixel to w, so the projected pixel size of texture on the screen will be at most $(w/(2 \times \kappa_0 \times S \times tan(\frac{\theta}{2})))$.

In the sequel, we say that two cubes C_l and C_k belonging to potentially different LOD level l and k are neighbors if none is contained in the other but their intersection is non-empty (that intersection is then necessarily a vertex, an edge, or a face of them).

Lemma 1. *If $\kappa_0 > 1$, C_l and C_k are neighbours and both retained visualization, then $|k-l| \leq 1$.*

Proof. Without loss of generality, we may assume that $k > l$. Indeed, if they are equal there is nothing left to prove, and if $l < k$ we may just rename them in the opposite way. In particular,

we have $k > 0$ and $l < L$. Since C_l is suitable we thus have

$$d(v_p, C_l) \geq \kappa_0 2^{-l}. \tag{4.2}$$

Also, since $P(C_k)$ is not suitable we have

$$d(v_p, P(C_k)) < \kappa_0 2^{-(k-1)}.$$

On the other hand, since C_l and C_k are neighbours (and so are C_l and $P(C_k)$), by the triangle inequality it follows that

$$d(v_p, C_l) \leq d(v_p, P(C_k)) + \mathrm{diam}(P(C_k)) \leq \kappa_0 2^{-(k-1)} + 2^{-(k-1)}. \tag{4.3}$$

Combining (4.2) and (4.3) and using that $\kappa_0 > 1$ we obtain

$$\kappa_0 2^{-l} < (\kappa_0 + 1)2^{-(k-1)} < \kappa_0 2^{-(k-2)},$$

and this is only possible if $k \leq l + 1$.

4.2.2 Adaptive and continuous level-of-details

Now we consider the situation where two cubes C_l and C_{l+1} ($P(C_{l+1}) \neq C_l$) are neighbors and are both retained for visualization. As illustrated in Figure 4.2, the cube C_l of purple parts corresponds to the coarser LOD M_l, the cube of yellow parts C_{l+1} corresponds to the LOD at higher resolution M_{l+1}. If we were to draw the corresponding $M(C_l)$ and $M(C_{l+1})$ unmodified, this would most likely lead to cracks in the visualized mesh. Our goal is to derive some estimations that will allow us to devise a continuous morphing between them. For that purpose, we consider a vertex:

$$v \in V(C_{l+1}) \quad \text{s.t.} \quad P(v) \in V(C_l).$$

Lemma 2. *Under the previous assumptions we have*

$$d(v_p, v) \geq 2(\kappa_0 - \sigma_0)2^{-(l+1)}$$

and

$$d(v_p, P(v)) \leq (\kappa_0 + 1 + \sigma_0)2^{-l}.$$

Proof. Since C_l is suitable for visualization (and $l < L$ since $l + 1$ is involved), we have

$$d(v_p, C_l) \geq \kappa_0 2^{-l}.$$

The vertex v may not be inside C_l, but we know that $P(v) \in V(C_l)$ and therefore from Eq. (3.3)

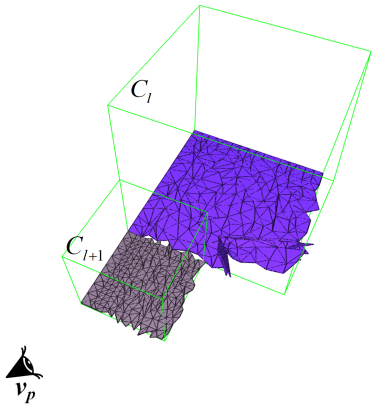


Figure 4.2: Some cracks between C_l and C_{l+1} can be seen due to the difference in mesh resolution.

and our simplification constraint it follows that

$$d(v, C_l) \leq d(v, P(v)) + d(P(v), C_l) \leq \sigma_0(2^{-(l+2)} + 2^{-(l+1)}) \leq \sigma_0 2^{-l}.$$

Therefore, by the triangle inequality

$$d(v_p, v) \geq d(v_p, C_l) - d(v, C_l) \geq (\kappa_0 - \sigma_0)2^{-l}$$

and the first statement of the lemma therefore follows. For the second, we use that $P(C_{l+1})$ is not suitable, that is

$$d(v_p, P(C_{l+1})) < \kappa_0 2^{-l}.$$

By the triangle inequality and the fact that C_l and C_{l+1} are assumed to be neighbors, we then obtain

$$\begin{aligned} d(v_p, P(v)) &\leq d(v_p, P(C_{l+1})) + \operatorname{diam}(P(C_{l+1})) + d(P(v), P(C_{l+1})) \\ &\leq \kappa_0 2^{-l} + 2^{-l} + \sigma_0 2^{-l}, \end{aligned}$$

where we have used Eq. (3.3) to estimate the last of the three terms.

For the visualization of $M(C_l)$ we will not draw an arbitrary vertex $w \in V(C_l)$ at its original position, but at the viewpoint-dependent and LOD-dependent position

$$I(v_p, w, l) := \lambda w + (1 - \lambda)P(w),$$

where the interpolation factor $\lambda \in [0, 1]$ is of the form

$$\lambda = \lambda(w, v_p, l) = b(d(v_p, w)/2^{-l}),$$

for some fixed continuously decreasing blending function $b : \mathbb{R}^+ \to [0, 1]$.

We choose b is such a way that $b(s) = 1$ if $s \leq (\kappa_0 + 1 + \sigma_0)$ and $b(s) = 0$ if $s \geq 2(\kappa_0 - \sigma_0)$ this imposes the condition

$$\kappa_0 > 1 + 3\sigma_0$$

on the two coefficients κ_0 and σ_0 of the model[1]. κ_0 is a real-time constant that can be adjusted during rendering to achieve the target resolution. Initially, we take the value $2.0 \leq \kappa_0 \leq 4.0$. According to the edge length of the model, if the edge length is relatively large, the value of σ_0 can be set larger, and the corresponding value of κ_0 should also increase. With such a choice, in a situation described by Lemma 2 we will have

$$I(v_p, v, l + 1) = I(v_p, P(v), l) = P(v),$$

and therefore no crack will occur at the border between the morphings of the neighbor mesh partition $M(C_l)$ and $M(C_{l+1})$.

Vertex interpolation is carried out in the vertex shader, and the specific process is as follows: for any vertex belonging to the cube retained for visualization, the level l of LOD, along with the two constants κ_0 and σ_0, and the viewpoint's position $v_p(c_x, c_y, c_z)$ are already known. The distance from v_p to the current vertex $\omega(x, y, z)$ can be computed as

$$d(v_p, \omega) = max(|x - c_x|, |y - c_y|, |z - c_z|).$$

Then, an interpolation band based on level l and two constants will be calculated. The vertex interpolation will begin from d_{start} to d_{end}, where

$$d_{start} = (1 + \kappa_0 + \sigma_0)2^{-l},$$

and

$$d_{end} = (\kappa_0 - \sigma_0)2^{-(l-1)}.$$

The interpolation coefficient can be computed as follows:

$$\lambda = clamp((d_{end} - d(v_p, \omega))/(d_{end} - d_{start}), 0.0, 1.0).$$

Thus, the position of viewpoint-dependent and LOD-dependent will be

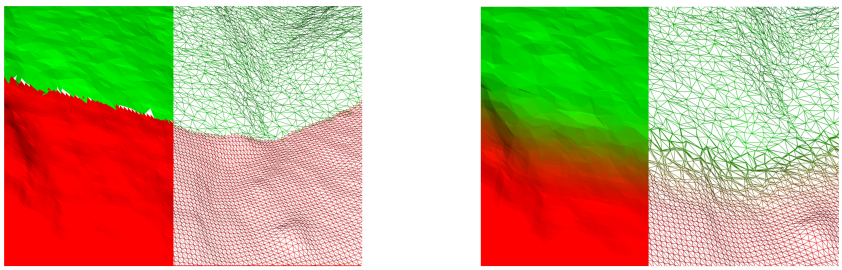
$$\omega_{result} = \lambda\omega + (1 - \lambda)P(\omega).$$

The vertex interpolation is done in real-time by the GPU implemented in the vertex shader, and the pseudo-code of vertex interpolation for ω and $P(\omega)$ to generate new position ω_{result} is

[1]In practice we will take $\kappa_0 = 2.0$ (or maybe between 1.5 and 4.0) and σ_0 much smaller than 1 (and certainly smaller than 0.1).

as follows:

```
d = distance(v_p, ω);
λ = clamp((d_end - d)/(d_end-d_start)),  0.0,  1.0);
ω_result = lerp(ω, P(ω), λ);
```



(a) Original vertex position (b) Vertex interpolation result

Figure 4.3: Comparison of results between our method without vertex interpolation (a) and with vertex interpolation (b), where the green part represents LOD3, and the red part represents LOD4.

If we draw an arbitrary vertex at its viewpoint-dependent and LOD-dependent position, no crack will occur at the border between the morphing of the neighbor mesh partitions $M(C_l)$ and $M(C_{l+1})$ (Figure 4.3 (b)) and no popping during the LOD switching. In principle, vertex interpolation could potentially create mesh self-intersection, but we have not experienced it in practice.

One implication of this morphing function is that when we save a mesh cluster on disk, we must not only write the position of its vertices but also of their direct parent in the hierarchy.

4.2.3 Visibility culling

Visibility culling [AHH19] is a technique used in real-time rendering to reduce the amount of geometry needed by only rendering objects that are visible from the viewpoint or contribute to the final image. This is particularly important in large-scale 3D scenes where the number of triangles can be very high, resulting in poor performance and decreased frame rates. Our approach involves two techniques: frustum culling and back-face culling.

Frustum culling involves determining which cubes are within the viewing frustum of the camera and therefore need to be rendered. It is used to remove the cube which is out of the frustum pyramid. It can be done using simple geometric tests during the hierarchical octree traversal, such as checking if the cube's bounding box intersects with the viewing frustum. If a cube is determined to be outside the frustum, its children's cubes can be skipped as well, since they are contained within the parent cube's bounding box.

Another operation is backface culling, which involves determining whether some portions of the mesh are not visible due to being occluded by other objects. This can be done by precomputing visibility information during the construction of the multi-resolution model structure, or

by using runtime algorithms to dynamically determine visibility based on the current camera position and orientation. The depth buffer test can be used in conjunction with face culling to perform visibility culling for the rendering of our multi-resolution structure, and it is implemented by enabling the GL_DEPTH_TEST function in OpenGL.

4.2.4 Screen space error control

Screen space error (SSE) control is critical for multi-resolution models because it ensures that the level of detail displayed in the model is appropriate for the current viewing distance and angle. The choice of SSE value can significantly impact the visual quality and performance of a rendering system. Combining our view-dependent octree selection rule with error bounds at the mesh simplification stage implies some screen space error bound at run time. First, for each cube C_l, during the mesh simplification phase, we impose that the quadric error ϵ, relative to the cube size, is bounded by some user-defined constant ϵ_q:

$$\frac{\epsilon}{2^{-l}L_c} \leq \epsilon_q. \tag{4.4}$$

Then, if the cube C_l is suitable for visualization, its distance d from the viewpoint satisfies $d \simeq \kappa_0 2^{-l} L_c$, where L_c is the length of C_0. For a lateral field of view angle θ and a screen width S_w, this yields the screen space error bound

$$\epsilon_{SSE} \lesssim \frac{\epsilon_q S_w}{\kappa_0 tan(\theta/2)}. \tag{4.5}$$

In fact, our vertex interpolation method allows smooth LOD transition regardless of the screen space error of the approximation. With the relationship between κ_0 and SSE, we can also use the target SSE to decide the value of κ_0. The κ_0 can be calculatd as follow:

$$\kappa_0 = \frac{\epsilon_{MRE} S_w}{\epsilon_{SSE} tan(\theta/2)}, \tag{4.6}$$

where ϵ_{MRE} represents the mean relative error of the multi-resolution model, which can be calculated by using the following pseudocode:

```
while (l > 0 and l < L){
  while(i = 0 and i < N){
    error += ε_{C_i};
    count++;
  }
}
// Compute the mean error and adapt it to the cube size of the M_L
ε_MRE = (error / count) / (L_c / (1<<L) );
```

In pseudo-code, N represents the total number of the cube for current LOD, and the count

is the total number of cubes in the multi-resolution model.

4.2.5 Textures and other attributes

For models with material properties, particularly textures, a multi-step process is followed. First, generate mipmaps for textures corresponding to LODs. These mipmaps form a sequence of textures, each progressively reducing the resolution of the same image [Cor]. These mipmaps are designed to meet the resolution requirements of each LOD level. Next, during mesh simplification and LOD generation, texture coordinates for each LOD level are computed. Mesh simplification for textured models is achieved using the extended quadric error method, as introduced in [GH98].

For normals, we recalculate normals for the mesh of each LOD during multi-resolution construction, in order to ensure that the shading and lighting effects remain consistent and visually appealing across different LODs. The extended QEM-based edge collapse method proposed in [GH98] is used for simplifying color attributes as well. In our case, new texture coordinates and color attributes are not generated separately, instead, they are considered along with vertex positions when computing the quadric error.

During the rendering, we use the same interpolation coefficient λ to blend the texture coordinate, normal, and color with their parents:

$$t_{result} = \lambda t + (1 - \lambda)P(t),$$

where t_{result} is result value of this current attribute, the t is the value of attribute and $P(t)$ is its parent.

4.2.6 Overall implementation

View-dependent rendering is based on the multi-resolution model. When the visualization condition is satisfied, the coarsest LOD (M_0) is rendered. If this condition is not met, we traverse the octree to obtain a list of visible cubes based on the distance between the viewpoint and the cubes. During this traversal, frustum culling is executed to select the final drawable cubes.

We do not use vertex attributes (which would double vertex cost) but indices into SSBO directly. Using indices SSBO directly instead of vertex attributes can reduce memory overhead since the child_parent_map (the index of its parent vertex) consumes less memory. They are typically used for passing large amounts of data between shaders, such as vertex data or information. In our case, since each vertex attribute is stored in a single SSBO, child vertices can directly access their parent vertices by accessing child-parent dependencies (child_parent_map). This allows for efficient traversal and lookup of vertex data, without the need for additional indexing or memory overhead.

In Algorithm 3, the function LoadChildCube() gets the child cube of C_l in an octree fashion. Draw() is related to the draw call function of OpenGL, glDrawElementsBaseVertex() is used in our implementation. FrustumCulling() returns a bool value. If it is in the frustum, the logical

coordinate of the cube is pushed into the stack. The **stack** retains the logical coordinates of drawable cubes during traversal, and the data updates adhere to a First-In, Last-Out (FILO) order. Normally, the cube placed on the top of the stack is closer to the viewpoint and belongs to the high-resolution LOD, which will be drawn first. The rendering order of the cube aligns well with depth-first rendering, which minimizes overdraw by rendering objects from front to back based on their distance from the viewpoint.

Algorithm 3 View-dependent rendering of multi-resolution models

Input : The multiresolution model: $M_L \sim M_0$;
The position of viewpoint: v_p;

```
if (M_0 suitable) then
    Draw (M_0)
else
    while C_l in M_1 do
        if d(v_p, C_l) ≥ κ_0 2^{-l} then
            stack.push(((l, i, j, k)))
        else
            LoadChildCube (C_l)
            if (IsInFrustum(C_{l+1}) == true) then
                stack.push(((l, i, j, k)))
            else
        end
    end
end
while stack.size() ≠ 0 do
    Draw(C_l);
    stack.pop((l, i, j, k);)
end
```

4.3 Out-of-core rendering

4.3.1 Memory management

In out-of-core rendering, the multi-resolution model data is stored on slower secondary storage due to the limited size of the GPU memory and data transfer overhead. Consequently, cache management assumes a critical role in maintaining optimal performance. The cache replacement policy becomes an essential algorithm used to decide which data should be removed from the cache when it reaches capacity, and new data needs to be loaded.

The Least Recently Used (LRU) strategy is a common approach to optimize memory management in such scenarios. LRU keeps track of the most recently used and least recently used data in the cache. When the cache reaches its memory limit and new data must be loaded, the

data that has been least recently accessed is evicted from the cache to allocate space for the new data. Implementing the LRU strategy in out-of-core rendering systems promotes efficient memory utilization, allowing the GPU cache to store the most critical data for rendering.

LRU structure

To meet the algorithm requirements, the LRU structure is designed as follows: a hash map is employed for rapid querying of whether the current cube is in memory, as it enables accessing a random key in $O(1)$ time complexity. In this design, the key value of the hash map is the logical coordinate ijk of each cube C_l, and the value is a pointer to the elements of the list structure. For the implementation of the LRU, the hash map is realized using the `unordered_map` introduced in C++11. The hash function employed by the `unordered_map` is MurmurHashUnaligned2, as detailed in the SMHasher library [App].

The list is used to store the logical coordinates ijk of each cube C_l and the memory offset Offset_{C_l} of the current memory block. The list structure allows us to remove the last entry in the LRU cache in $O(1)$ time complexity and add/move an entry to the front of the LRU cache in $O(1)$ time complexity.

There are two main operations of LRU structures: *put* and *get*.

- *put* operation set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least recently used item before inserting a new item.
- *get* fetches the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

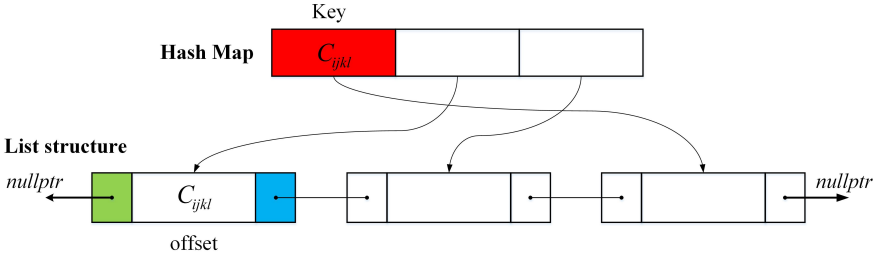


Figure 4.4: LRU structure for cache management.

Memory allocation

The memory allocation is based on the geometry data of the multi-resolution structure. First, we traverse the OOM tree structure to get the maximum vertex number (l, NUM_v) or maximum index number (l, NUM_i) of each cube for each level l of LOD. The slot size of the cache can

be decided by the maximum size of the vertex MAX_v and index MAX_i, the memory allocation factor can be calculated combined with the byte size of these values as follows:

$$Ratio = \frac{N_{attribute} * MAX_v * sizeof(float)}{MAX_i * sizeof(unsigned\ int)},$$

where the $N_{attribute}$ presents the total count of attribute values in the current input model. For instance, in the case where the model consists only of positions and normals, $N_{attribute} = 6$, as positions are represented by 3 float values, and normals also encompass 3 float values. `sizeof()` retrieves the size of a data type in bytes, such as `float`. With the available memory space $Total_{mem}$, the memory occupied by the index buffer is

$$MEM_{index} = \frac{Total_{mem}}{(Ratio + 1.0)},$$

and the memory for vertex buffers is

$$MEM_{vertex} = \frac{Total_{mem} * Ratio}{N_{attribute} * (Ratio + 1.0)},$$

the number of slot N_{slot} for each buffer is calculated by following:

$$N_{slot} = \frac{MEM_{index}}{MAX_i * sizeof(unsigned\ int)},$$

where N_{slot} represents the capacity of the LRU cache. Based on the attribute type of the current model, the MEM_{vertex} is divided into several buffers: position, normal, child-parent remap, and texture coordinates (color) if it exists.

The distribution of vertices and indices can vary significantly between different cubes at various LODs, making it challenging to determine the optimal cache slot size for each buffer. If we were to use the vertex and index data from the second largest cube to initialize the cache slot size, it would lead to significant memory wastage. Typically, cubes at higher-resolution LODs will have a maximum number of vertices and indices that are closer in magnitude, but smaller than those at lower-resolution LODs. However, it is essential to note that the total number of vertices and indices in coarser LODs is much smaller than in high-resolution LODs, as the coarser LODs represent less detail and require fewer vertices and indices to achieve their level of detail.

To make efficient use of GPU memory space and enhance rendering performance, we conduct an analysis of the maximum number of vertices and indices per cube at different LOD levels. Through this analysis, we identify a specific level value, denoted as $level_{upload}$, which represents the threshold for the linear distribution of the maximum vertex and index values per LOD level. When $l \leq level_{upload}$, we upload all the data of related LODs to the GPU.

However, for LODs where $l > level_{upload}$, we employ the LRU method to manage the mesh data related to cubes. The determination of the optimal cache slot size is based on the maximum number of vertices and indices within the LODs (M_l) for which $l > level_{upload}$. Once we have

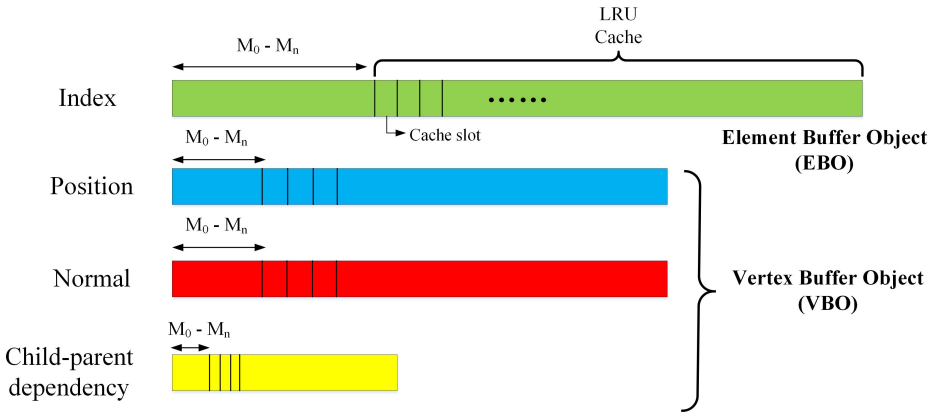


Figure 4.5: Memory distribution for vertex attribute and index.

established the optimal cache slot size, we calculate the maximum slot number (LRU capacity) based on this value.

In Figure 4.5, the memory distribution under the OpenGL framework has been shown, the Vertex Buffer Object (VBO) contains data such as position, normal, and child-parent dependency. An Element Buffer Object (EBO) stores an array of indices, where each index typically corresponds to a vertex in a VBO. These indices determine the order in which vertices are processed during rendering. The geometry data of M_0 to M_n is uploaded into the GPU directly, where $n = level_{upload}$. For the rest of the LOD, the LRU cache strategy is used to manage the cube data streaming and rendering.

4.3.2 Rendering

Scene traversal

Out-of-core rendering utilizes a multi-resolution representation in conjunction with precomputed static LODs. With the OOM tree structure generated during out-of-core multi-resolution construction, the rendering algorithm starts its traversal from the root cube in each frame. Upon reaching cube C_l, if it satisfies the visibility condition (4.2), C_l is added to the draw call list $Stack_{Draw}$. However, in the context of out-of-core rendering, accounting for data availability in RAM/VRAM is crucial for real-time performance, considering the data streaming time. Therefore, an additional conditional statement pseudocode is provided as follows:

```
if(P(C_l) is not suitable){
    if(all the children are loaded){
        Stack_Draw.push(children);
    }
    else{
        Stack_IO.push(children);
```

```
        Stack_Draw.push(P(C_l));
    }
else{
    Stack_Draw.push(P(C_l));
    }
}
```

By applying the constraints mentioned above, we ensure a clear separation between rendering data and data uploading. This separation helps prevent real-time rendering from experiencing stuttering issues caused by delays in processing data required for rendering. Although our method may introduce a slight delay in achieving the desired resolution, it effectively mitigates performance issues such as stuttering.

During the traversal of the OOM tree, our algorithm relies on the OOM tree itself, which contains essential connectivity information about cubes, such as child-parent relationships and additional data structures related to cube information. As the viewpoint changes, our traversal algorithm selects the appropriate cube at different LODs by combining the frustum culling based on the viewpoint's position. We utilize two stacks: $Stack_{Draw}$ for storing cubes to be rendered and $Stack_{IO}$ for storing cubes to be uploaded. Both of these stacks operate on a First-In, Last-Out (FILO) basis.

Rendering loop

During rendering, the logical coordinate of each cube can be obtained with the *pop* operation of the stack. With these logical coordinates, we can access the geometry data, parent cube information, and the offset of the vertex buffer and index buffer. To prevent LOD popping and mesh cracks, when drawing the current cube C_l, we also require the geometry data of its parent cube C_{l-1} at the same time.

Due to our scene traversal rule, parent cubes at LODs with coarser resolutions are already streamed from disk to RAM. Therefore, during the draw call stage, we use the memory buffer offset to locate the geometry data of parent cubes, and execute vertex interpolation between child and parent in the same manner as in the in-core rendering stage. The pseudocode for this process is as follows:

```
for (C_l in Stack_Draw){
    if(C_l in VRAM){
        Offset_{C_l} ← LRUCache.get(C_l);
        Offset_{P(C_l)} ← LRUCache.get(P(C_l));
    }
    else{
        Offset_{C_l} ← LRUCache.put(C_l);
        Offset_{P(C_l)} ← LRUCache.put(P(C_l));
    }
}
```

```
glDrawElementsBaseVertex(GL_TRIANGLES, indices.size(),
                         GL_UNSIGNED_INT,
                         Offset_{C_l}.index, Offset_{C_l}.vertex);
```

In the pseudo-code above, LRUCache.get() corresponds to the LRU cache *get* operation, LRUCache.put() corresponds to *put* operation, and they return the memory offset of the current cube. glDrawElementsBaseVertex() represents the draw call function of OpenGL.

4.3.3 I/O streaming

I/O streaming refers to the process of streaming geometry data from disk to the CPU memory (RAM), and from CPU memory to GPU memory (VRAM). The speed of data transfer from disk to RAM is influenced by the reading speed of disk or RAM. On average, the speed of a modern hard disk drive is around 100 MB/s to 200 MB/s, while the data transfer speed of a solid-state drive (SSD) can be as high as 500 MB/s or more. However, the speed of data transfer from RAM to VRAM is faster than the speed from disk to RAM. RAM and VRAM can access data quickly and efficiently without the need for mechanical read/write operations. Additionally, modern graphics cards are designed to support high-speed data transfer between RAM and VRAM. For discrete GPU, PCI-Express enables the transfer of data from RAM to VRAM, with transfer rates ranging from 4GB/s to 16GB/s for PCIe 3.0.

In our case, I/O streaming involves both disk-to-RAM and RAM-to-VRAM data transfers. Data streaming from disk to RAM, due to its relatively low transfer speed, is executed separately in a dedicated thread that does not affect real-time rendering. On the other hand, data streaming from RAM to VRAM is integrated into the rendering loop. When a cube is initially considered for visualization, its geometry data is uploaded from RAM to VRAM. The total amount of data transferred from RAM to VRAM per frame can have an impact on real-time rendering performance.

To further improve data bandwidth, during the out-of-core construction phase, the data can be quantized and stored in a file format. Quantization parameters are determined based on a cube-based mesh partition. The approximate data bandwidth of the rasterization pipeline is illustrated in Figure 4.6.

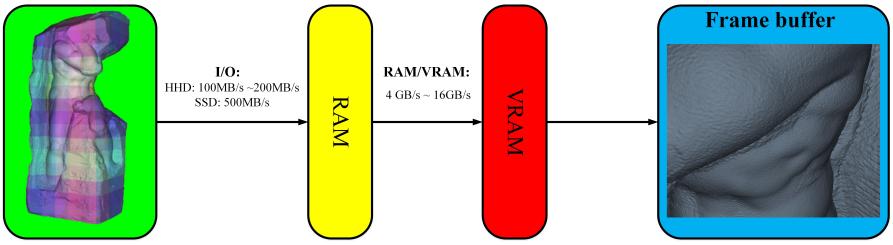


Figure 4.6: Approximate bandwidth of rasterization pipeline.

4.3.4 Parallelization

In the out-of-core rendering process, employing two threads can be highly beneficial as it can enhance rendering performance, reduce latency, and improve efficiency. In our implementation, we utilize two threads: an I/O thread and a render thread (Figure 4.7).

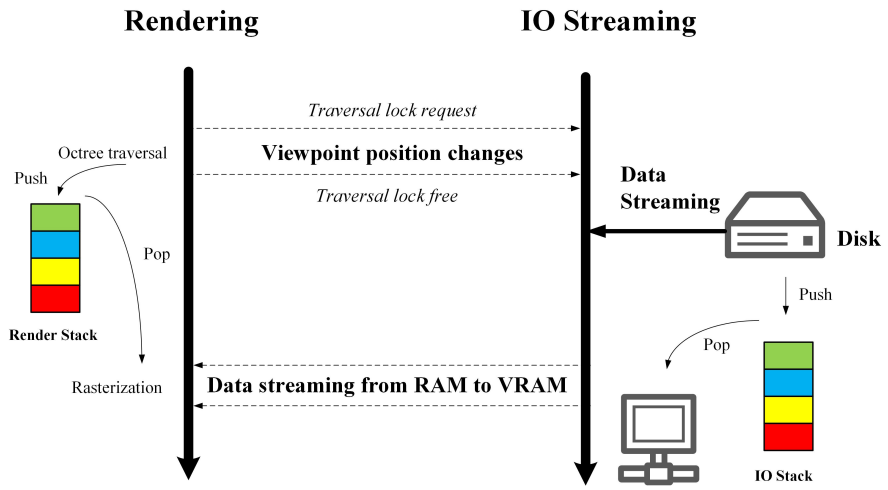


Figure 4.7: Parallel rendering and streaming thread.

The I/O thread is dedicated to handling I/O operations and reading data from a disk. It gets the logical coordinates of the cube that needs to be loaded by getting the data from $Stack_{IO}$, and if it has not been loaded, the data of this cube will be loaded into the main memory.

```
for (C_l in Stack_IO){
    if(C_l not in RAM){
        LoadCube(C_l);
    }
}
```

While the render thread manages rendering operations, including scene traversal and the rendering loop, scene traversal is typically prioritized during the parallelization of I/O streaming and rendering. This is because the $Stack_{Draw}$ and $Stack_{IO}$ need to be updated to determine which parts of the 3D scene require rendering or uploading. Initially, if the viewpoint is far from the object and lower-resolution LODs have been uploaded to VRAM, only the $Stack_{Draw}$ is updated. As the viewpoint approaches the object, both stacks are updated based on the criteria in section 4.3.2. Subsequently, the rendering loop reads the $Stack_{Draw}$ to obtain the logical coordinates of the cubes that need to be rendered. If the data is already in VRAM, the offset of its data in memory is read. If the data is not in VRAM, the LRU method is used to load the cube data into VRAM.

4.3.5 Overall implementation

In Algorithm 4, the pseudocode of overall implementation is listed as following:

Algorithm 4 Out-of-core real-time rendering

Input : The multiresolution model file on disk: $M_L \sim M_0$;
The OOM tree structure file $File_{tree}$;
The position of viewpoint: v_p;
OOMTree ← Read&InitializeOOMTree($File_{tree}$);
if *($l \leq level_{upload}$)* **then**
  Upload ($M_0 \sim M_l$) ;
else
  Initialize(LRU);
end
if *(M_0 suitable)* **then**
  Draw(M_0);
else
  while C_l *in* M_1 **do**
    if $d(v_p, C_l) \geq \kappa_0 2^{-l}$ **then**
      $Stack_{Draw}$.push((C_l));
    else
      if *children of* C_l *in RAM* **then**
        $Stack_{Draw}$.push((the children of C_l)) ;
      else
        $Stack_{Draw}$.push(($P(C_l)$));
        $Stack_{IO}$.push((the children of C_l));
      end
    end
  end
end
`// Rendering thread`
while $Stack_{Draw}.size() \neq 0$ **do**
  Draw(C_l);
  $Stack_{Draw}$.pop((l, i, j, k));
end
`// IO streaming thread`
while $Stack_{IO}.size() \neq 0$ **do**
  LoadFromDisk(C_l);
  $Stack_{IO}$.pop((l, i, j, k));
end

Where the function Read&InitializeOOMTree() initializes the OOM tree from the file. Up-

load() loads the LOD data from disk to GPU. Initialize() decides the LRU slot size and the maximum slot number based on the statistics of index and vertex count distribution per cube. Draw() relates to the draw call function of OpenGL. LoadFromDisk() streams data from the disk to RAM.

The out-of-rendering process commences by reading the OOM tree structure file. Following the initialization of the OOM tree structure, the fundamental information of each cube at different LODs is updated, such as the number of vertices and indices. Based on the statistics of the vertex and index counts, the memory allocation factor $Ratio$ and $level_{upload}$ are determined, and the LRU cache structure is initialized accordingly. For the cubes in M_l $(0 \leq l \leq level_{upload})$, the data is uploaded from the disk directly to VRAM. For the remaining LODs, the cube is uploaded if it suits for visualization. During the rendering loop, the scene traversal updates $Stack_{draw}$ and $Stack_{IO}$. These updates serve the purpose of identifying the cube list that requires data uploading and the cube list that is ready for rendering. After the octree traversal, the IO thread and the rendering thread run parallel to achieve real-time rendering.

4.4 In-core rendering results

All the tests were done on a desktop machine (without discrete GPU) equipped with an Intel UHD Graphics 630 and Intel i7-8700 CPU (16GB) and a laptop (Intel i7-12850HX CPU 32 GB) with discrete GPU (Nvidia RTX A4500 Laptop GPU 16GB). The test result is based on 12 models: two fractals models, Happy Buddha, Egyptian tomb, Egyptian wall painting, David (1mm resolution), Vanil Noir mountain region ($8km \times 8km$), Night sculpture, Gurnah temple mural, Lucy, Locarno region ($16km \times 16km$) and Egyptian temple mural. At the same time, we conduct a performance comparison of our method with Nexus [Vis20]. For this comparison, we utilize the module nxsview from this open-source library, which can render a batched multi-resolution 3D model structure. For all the results shown next, we use a window size of 1280×800.

4.4.1 Method profiling

To analyze the performance of different parts in our rendering pipeline, specifically, octree traversal and rendering loop time per frame. We recorded the elapsed time for these aspects during the real-time rendering of David, progressing from afar to a near viewpoint. In Figure 4.8, as the viewpoint moves closer to the object, the octree traversal time increases due to the desired LOD being situated at deeper levels within the octree. Additionally, the region of interest shrinks, resulting in reduced rendered triangle numbers with the increased draw call (cubes) numbers. Our method incorporates vertex interpolation and does not significantly increase the draw call time when compared to directly drawing the original position of the vertex, even though the interpolation process in vertex shader includes distance calculation, interpolation factor calculation, and LOD-dependent position calculation.

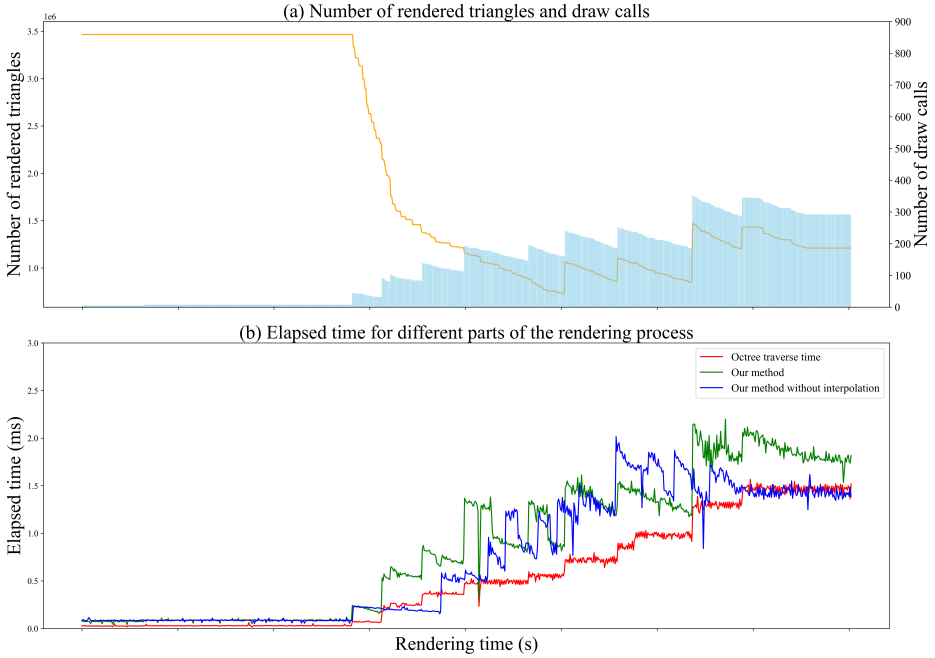


Figure 4.8: Method profiling of different parts in our rendering pipeline.

4.4.2 Comparison with full-resolution model rendering

The rendering performance of the original model and its multi-resolution representation are compared. Table 4.1 lists the frame time of rendering each full-resolution model and its multi-resolution model with a dynamic viewpoint. The frame time data were obtained by controlling the movement of the viewpoint to the model from a distance, so it has a certain range of volatility. As the viewpoint approaches the model, the level of detail in the model increases, leading to a larger number of rendered triangles and, consequently, an increase in frame time. Conversely, as the viewpoint moves farther away from the model, the level of detail decreases, resulting in a decrease in the number of rendered triangles and a decrease in frame time. The proposed multi-resolution structure outperforms rendering the original full-resolution model, especially for larger models. The challenge with real-time rendering of large models lies in the sheer size of the model, which can easily overwhelm available system resources. Processing and storing a vast amount of data in memory can lead to slow frame rates, extended load times, and potentially system crashes.

Table 4.1: Frame time comparison of rendered multi-resolution models and original full-resolution models

Model	Frame time (ms)	
	Full-resolution	Multi-resolution
$Fractal_1$	9.42-16.21	2.09-13.89
Happy Buddha	5.72-11.52	1.34-8.45
$Fractal_2$	6.43-15.45	1.43-8.03
Egyptian Tomb	9.96-19.86	3.21-13.10
Vanil Noir	14.10-18.12	2.47-6.73
Night	51.26-59.12	5.16-9.05
Lucy	111.07-112.36	5.01-13.03
Locarno region	120.07-132.52	3.48-8.65
Egyptian wall painting	321.85-323.81	8.89-15.05
Gurnah temple mural	138.56-161.17	4.33-12.30
David(1mm)	129.34-138.628	3.91-10.31
Egyptian temple mural	459.66-463.01	3.14-14.70

Figure 4.9 shows the view-dependent rendering results of David. The full-resolution model contains 56 million triangles and is rendered at 32 fps (Figure 4.9 (a)) on a mainstream desktop platform. Figure 4.9 (b) and Figure 4.9 (c) are the rendering results of the multi-resolution model, and the local magnified image depicts the different LODs by colorization. By comparing the two local magnified images, it shows that our method can avoid mesh cracks by implementing vertex interpolation.

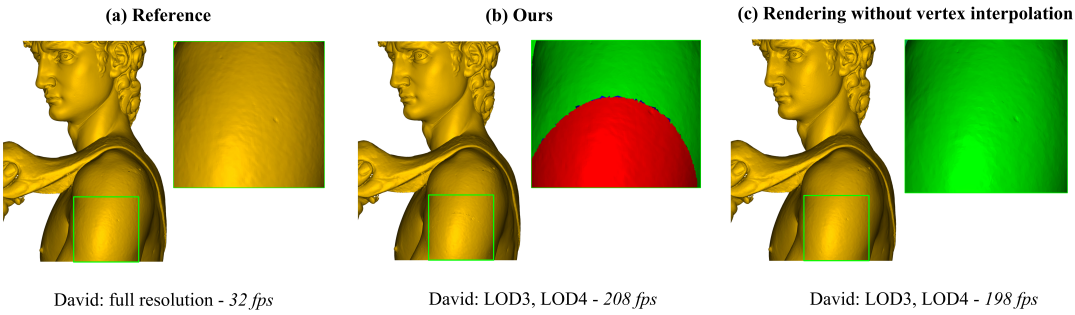


Figure 4.9: View-dependent rendering results and comparison of David.

4.4.3 Comparison with other methods

We present real-time rendering results of several 3D models, namely Terrain, David, Egyptian wall painting, and Gurnah temple mural. Figure 4.10 shows the rendering results of our method,

our method without vertex interpolation, Nexus, and a reference rendering result. Both Nexus and our method can avoid cracks compared to our method without vertex interpolation, but Nexus generally has a lower FPS compared to our method.

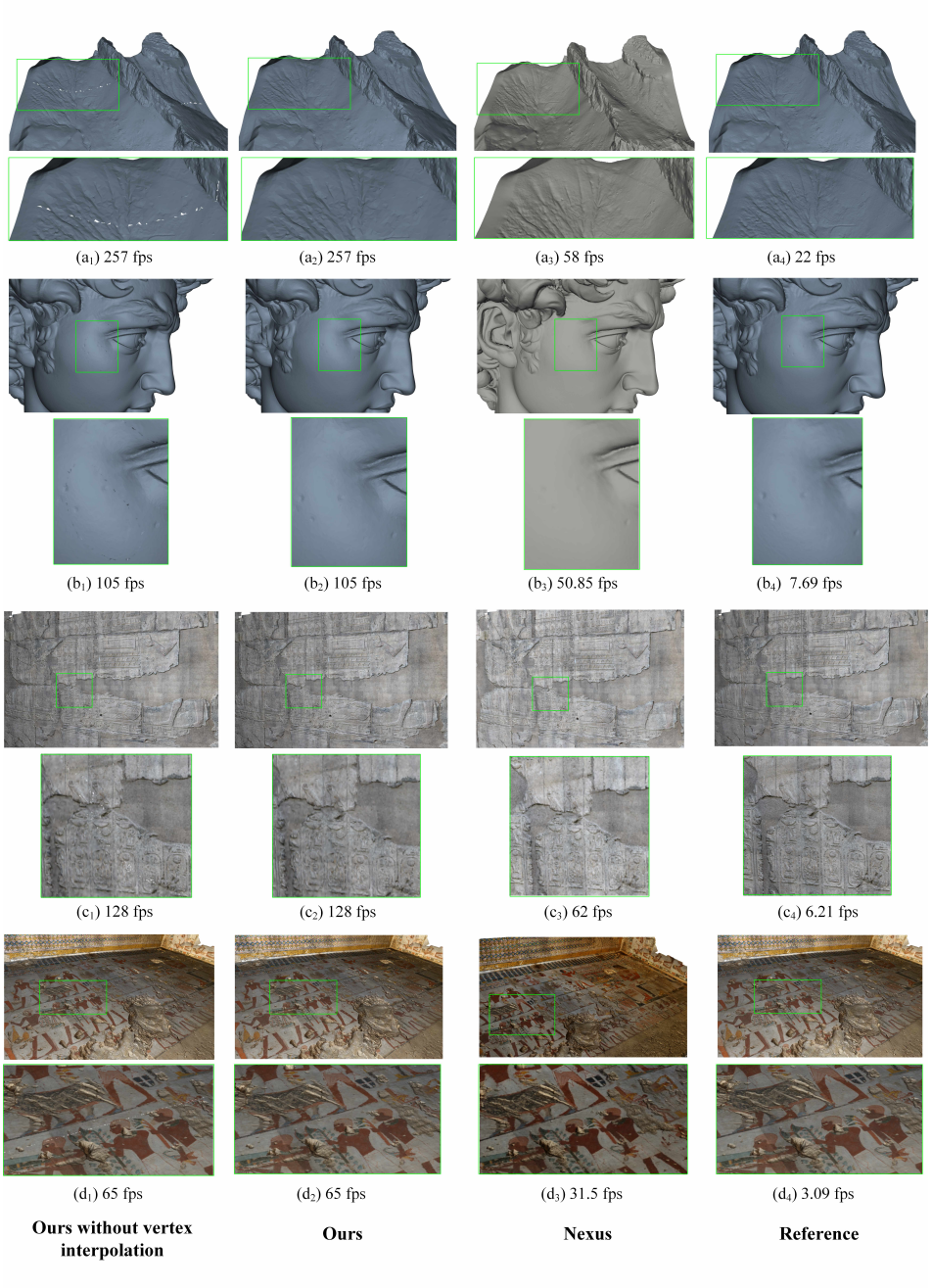


Figure 4.10: The comparison of rendering results of different methods.

In order to compare the geometric change process of the two methods in the LOD transition, we control the viewpoint moving from far to near, and the plane model rendering resolution transitions from LOD3 to LOD4, the number of triangles is recorded for each view (Figure 4.11). For Nexus, it eliminates cracks and smooths the popping artifacts by controlling the screen space error associated with a patch, with a small triangle budget or a larger target screen space error, it might lead to the LOD popping. However, in our method, each vertex is transitioned smoothly by interpolating with its parent, thereby eliminating LOD popping.

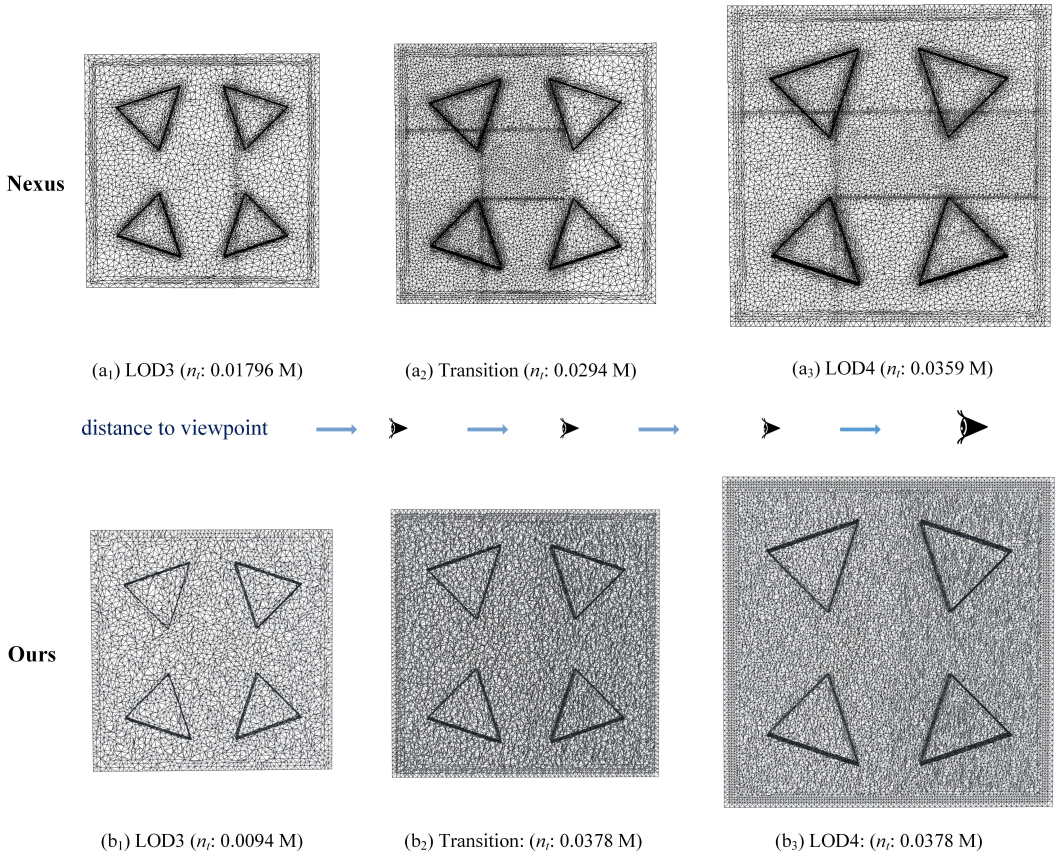


Figure 4.11: The LOD transition comparison between our method and Nexus, where n_t represents the number of rendered triangles. (a_1), (a_2) and (a_3) are the LOD transition phases rendered by Nexus. (b_1), (b_2) and (b_3) are the LOD transition phases rendered by our method.

We performed real-time rendering tests on 3D models under different rendering frameworks such as Nexus. The test models are Vanil Noir, Lucy, Locarno, Night, David, Egyptian wall painting, Gurnah temple mural, and Egyptian temple mural. In the case of rendering the same model, we control the viewpoint move from the overall model to the part of interest, the performance of different methods is evaluated by comparing frame time and the number of rendered triangles per frame. For Nexus, real-time rendering tests are performed based on the files that

were generated under multi-resolution construction, it uses a streaming approach, the loading and unloading buffers based on RAM and GPU budget. For our method, the in-core rendering scheme allocated the multi-resolution model data to the GPU at the beginning.

We analyzed two indicators during the rendering process, namely frame time and the number of triangles rendered per frame. Figure 4.12 (a) illustrates a comparison of the number of rendered triangles per frame for different models under different rendering frameworks. For Nexus, the triangle budget determines the resolution of the initial visualization result, we adjust κ_0 to 8.0 to ensure a similar number of initial rendering triangles during the comparison. As the viewpoint moves to the region of interest, only a small region of the model needs to be rendered, resulting in reduced number of rendered triangles. The reason for Nexus using more triangles could be caused by the non-hierarchical constraints that might be sub-optimal when the whole model is visible but some regions are close to the viewer. However, our method can maintain a relatively stable level of the number of rendered triangles per frame for different models and dynamic viewpoints without losing interactivity.

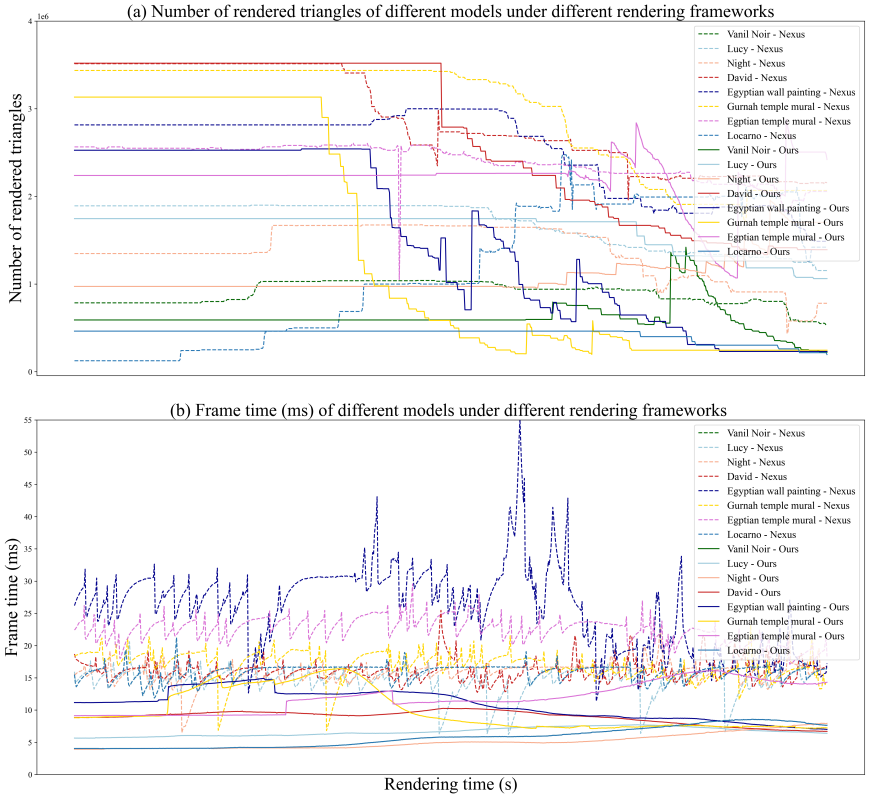


Figure 4.12: Rendering performance comparison of our method and Nexus [Vis20] with dynamic viewpoint.

Figure 4.12 (b) is a comparison of the real-time frame time of different 3D rendering methods,

our method can generate relatively stable frame time for the rendering of different scale models under the dynamic movement of the viewpoint, although the maximum or minimum frame time changes due to different models. For Nexus, both the different scales of the model and dynamic viewpoints have an impact on the frame time. The frame time fluctuates greatly as the viewpoint approaches the model, the Egyptian wall painting's frame time is higher than 56 ms at most.

In the above comparison, we initially used $\kappa_0 = 8$ to render models to match the number of triangles rendered in Nexus. However, in practical implementation, we have observed that using $\kappa_0 = 4$ works well for many types of models. Nevertheless, for Nexus, the smaller initial triangle budget value and the target projected error (in pixels) for large-scale models significantly affect the rendering performance. This indicates that Nexus might require a higher triangle budget to achieve a high-fidelity rendering result.

We also compare our method with SVDLOD [Hop98b] (Figure 4.13) by using terrain ($1km \times 1km$), and the wireframe rendering result is chosen to show the geometry information. Both methods generate multi-resolution models with the same level of LOD ($L = 7$), using edge collapse based mesh simplification. In Figure 4.13 (a_1), SVDLOD locks the shared mesh edge of boundary between clusters during the block-based simplification, which leads to the dense triangle cruft around the boundary (Figure 4.13 (a_2)). Our block selection method can avoid this problem (Figure 4.13 (b_2)). The test was also carried out on Unreal Engine 5, but we only rendered the Happy Buddha (14.71ms-16.39ms) and Egyptian tomb (13.15ms-22.12ms) successfully on our test platform. The main reason is that Unreal Engine 5 is a processor-hungry system, which makes it very difficult to achieve the best performance under the general configuration.

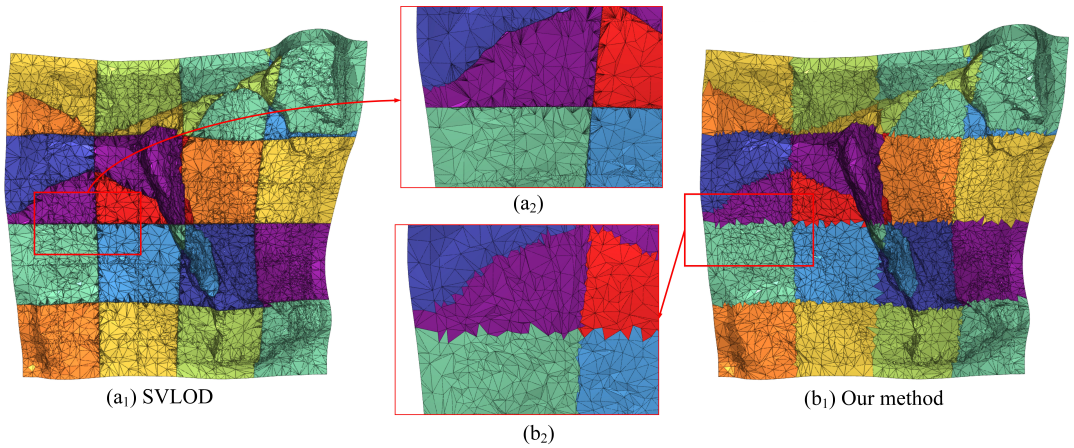


Figure 4.13: The comparison between our method and the SVDLOD [Hop98b] method of the same LOD rendering results.

4.4.4 Overdraw analysis

In Figure 4.3 (b), smaller triangles are concentrated around the edges of the larger triangles, resulting from vertex interpolation. The presence of these artifacts can be attributed to the similarity between vertex interpolation and vertex collapse. The interpolation process is illustrated with a single vertex in Figure 4.14. In practice, as vertices in the blending region are all interpolated, the effect depicted in Figure 4.3 is observed. These smaller triangles may indeed contribute to the problem of overdraw, but it's important to note that this issue is primarily confined to the blending region. Further analysis of overdraw is carried out as follows.

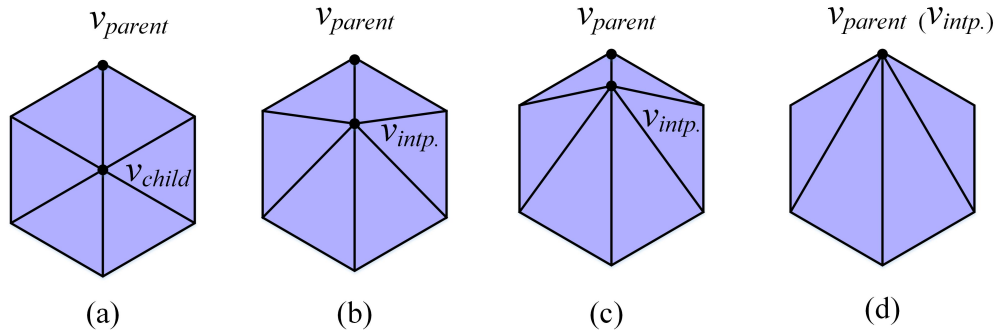


Figure 4.14: Single vertex interpolation process, where $v_{intp.}$ is the interpolation result.

Overdraw is one of the performance bottlenecks in rendering. It occurs when the same pixel is drawn multiple times in a single frame. Overdraw can significantly impact rendering performance because each additional draw call for the same pixel requires extra processing time. This can result in slower frame rates, longer rendering times, and increased power consumption.

We use atomic operations to count the number of times each pixel is drawn. The count value is stored in a 2D image, specifically in the red channel, using a 32-bit unsigned integer format denoted as $r32ui$. This step can be executed in the vertex shader with image atomic operations. The pseudo-code is as follows:

```
// This is a 2D image that is used to count overdraw in the scene
layout (binding = 0, r32ui) uniform uimage2D overdraw_count;
void main(){
    // Atomically add one to the contents of memory
    imageAtomicAdd(overdraw_count, ivec2(gl_FragCoord.xy), 1);
}
```

After counting the number of overdraws, we use 3 colors to indicate the overdraw count value, where the green part means the pixel is not drawn, the blue part means that the number of overdraws is within the normal range, and the red part means that the number of overdraws is high. The overdraw count is converted to color by the following operations in the fragment shader.

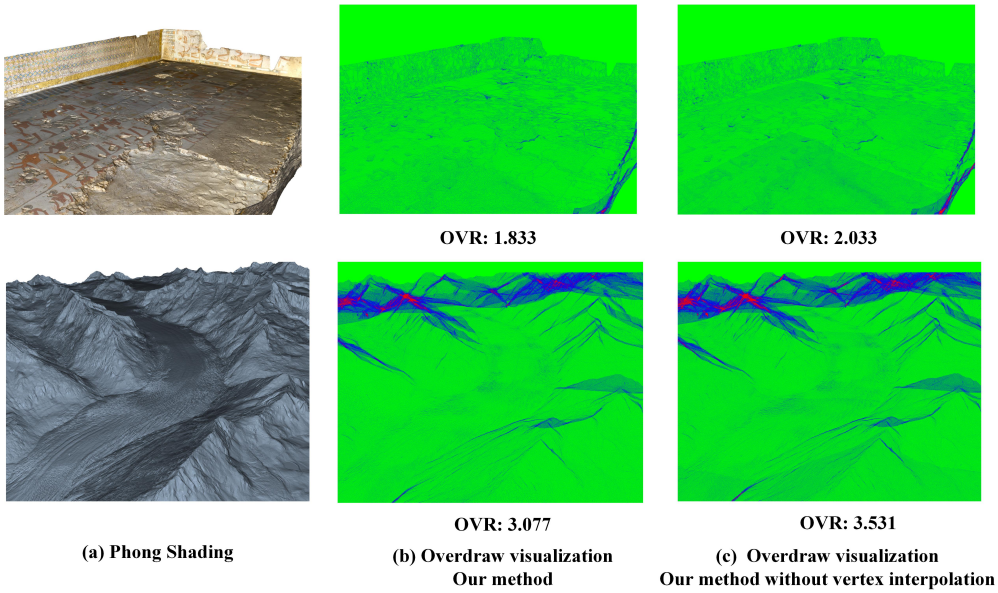


Figure 4.15: The overdraw analysis and OVR calculation of Egyptian wall painting and terrain.

```
out vec4 FragColor;
// Convert the overdraw count to the value in (0.0, 1.0)
float ratio =
      clamp(imageLoad(overdraw_count, ivec2(gl_FragCoord.xy)).x/32.0,
            0.0, 1.0);
// When ratio = 0.0, the pixel will be drawn as green
if (ratio < 0.5) {
  FragColor = vec4(0.0, 1.0 - 2.0 * ratio, 2.0 * ratio, 1.0);
}
else {
  ratio = 1.0 - ratio;
  FragColor = vec4(1.0 - 2.0 * ratio, 0.0, 2.0 * ratio, 1.0);
}
```

The visualization of the overdraw count provides an approximate distribution of the number of overdraws in the rendered scene. In addition, we also calculate the overdraw ratio (OVR). The OVR can be defined as the ratio between the total number of pixels passing the depth buffer and the number of visible pixels, it is calculated by

$$OVR = (sum_{overdraw})/(sum_{draw}).$$

We visualized the overdraw count of Egyptian painting and terrain, and calculated the OVR

for multi-resolution model rendering without vertex interpolation (3.531), and multi-resolution model rendering with vertex interpolation (3.077). The OVR shown in Figure 4.15 is not cache-optimized, which proves that our method does not aggravate the overdraw problem but reduces the overdraw count.

4.5 Out-of-core rendering results

The out-of-core rendering evaluations were done on a desktop machine (without discrete GPU) equipped with an Intel UHD Graphics 630 and Intel i7-8700 CPU (16GB) and a laptop (Intel i7-12850HX CPU 32 GB) with discrete GPU (Nvidia RTX A4500 Laptop GPU 16GB). The test results are based on two models: St.Matthew and Atlas. The multi-resolution model files of St.Matthew and Atlas are stored on the disk with the OOM tree file. For all the results shown next, we use a window size of 1280×800.

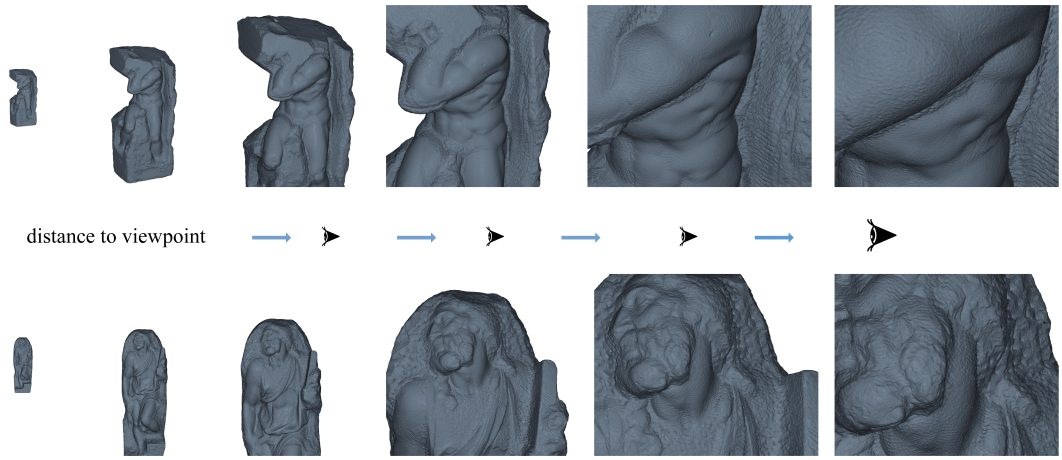


Figure 4.16: Out-of-core rendering of Atlas and St.Matthew with the viewpoint movement from far to the object.

In contrast to in-core rendering, loading models like St. Matthew and Atlas into main memory is challenging due to their size and scale. Therefore, comparing full-resolution model rendering with multi-resolution model rendering is not feasible. To assess the out-of-core performance of a rendering system, it's crucial to measure key metrics such as bandwidth, frame time, and rendered triangles per frame in real-time. In our approach, we recorded these metrics as the viewpoint transitioned from a distant view to close proximity, gradually increasing the resolution as the model was approached (refer to Figure 4.16).

Based on the experimental results (Figure 4.17), it is evident that the real-time rendering performance of the laptop surpasses that of the desktop. The laptop, equipped with an SSD and discrete GPU, maintains stable frame times throughout the evaluation. In contrast, the frame time of the desktop increases as the resolution rises. Compared to the discrete graphic card, the

integrated graphics card has limited video memory space, resulting in fewer available cache slot numbers. Consequently, when the viewpoint is close to the object and higher resolutions with more cubes are required, the integrated graphics card struggles due to the limited cache slots, leading to frequent LRU updates and reduced frame rates.

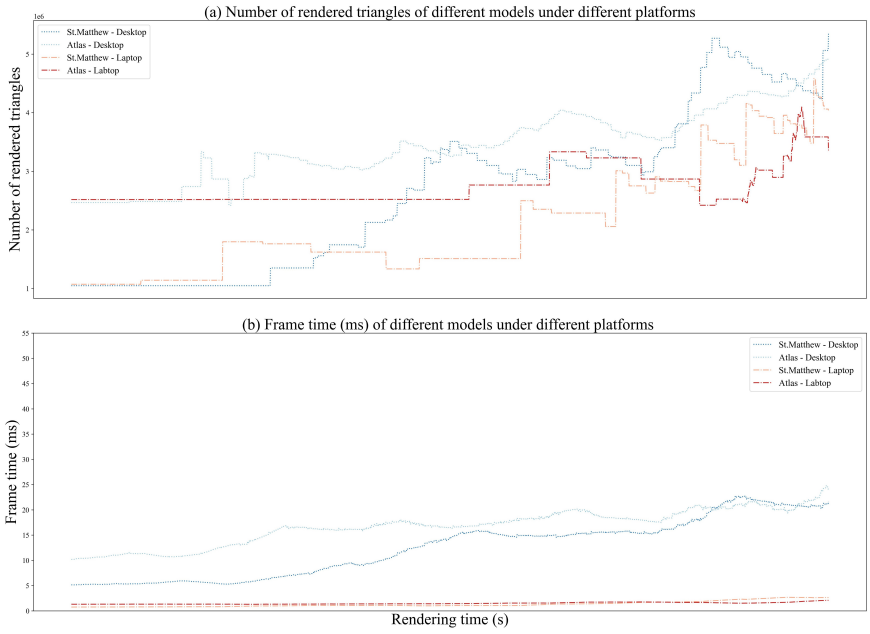


Figure 4.17: Out-of-core rendering performance of Atlas and St.Matthew.

In summary, benefiting from the faster data transmission speed of the SSD and the presence of a discrete GPU, the laptop platform demonstrates superior real-time rendering performance compared to the desktop platform. It maintains stable frame times and minimizes the perceptible delay in rendering higher resolutions, while the desktop experiences increased frame times and a more noticeable delay due to its reliance on HDDs for data storage.

Regarding the number of rendered triangles per frame, initially, both platforms start with the same triangle count as the selected LOD levels are preloaded. However, as the rendering of the remaining high-resolution LOD is managed by the LRU algorithm under limited video memory, the performance of the two platforms differs significantly. The number of triangles rendered on the laptop remains relatively stable, while the number of triangles rendered on the desktop fluctuates. This fluctuation occurs because the data thread on the desktop is delayed behind the rendering thread, and the data transfer speed of HDD is lower than SSD.

We have also recorded the bandwidth from disk to RAM and the bandwidth from RAM to VRAM, which refers to the data transfer rate from the disk to the CPU and from RAM to VRAM, on two platforms: a laptop equipped with an SSD and a desktop equipped with a Hard Disk Drive (HDD). In Figure 4.18 (a), the data transfer bandwidth from RAM to VRAM on

different platforms is illustrated, the speed of the two systems is not much different. Although the average performance of the laptop is better than that of the desktop, since the transmission speed between the CPU and the GPU is at giga bytes level, it will not have a significant impact on the real-time performance of the rendering thread.

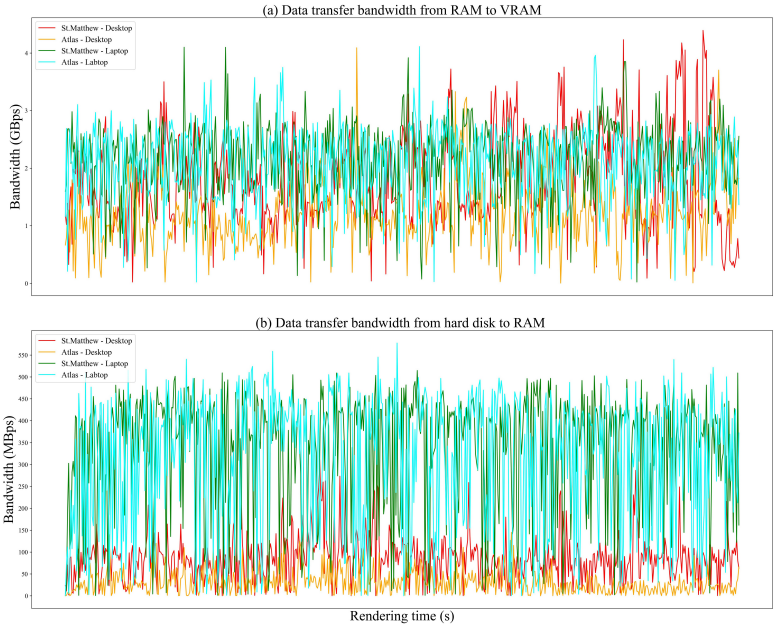


Figure 4.18: Bandwidth data of Atlas and St.Matthew during real-time rendering.

In Figure 4.18 (b), the data transfer bandwidth from disk to RAM was compared on two platforms, the SSD's transfer speed outperforms the HDD. The SSD's high data transfer speed ensures that any visual disruptions are quickly resolved even if the data streaming thread falls out of sync with the rendering thread. However, on the desktop with the HDD, the slower transfer speed significantly impacts the visual quality, leading to a noticeable waiting time of several seconds for data updates.

4.6 Conclusion

Combined with the multi-resolution construction in Chapter 3, we can develop a lightweight 3D viewer (Demo video) [2], which allows users to visualize and check the 3D models without booting up the processor-hungry software. The scheme of the viewer includes multi-resolution construction and real-time rendering, which avoids LOD-popping and mesh cracks between LOD parts based on vertex interpolation between child and parent. The implementation of the viewer could render scanned models of multiples tens of million triangles at optimal visual quality and

[2] https://drive.google.com/file/d/1QoIlu-2RYoWaYcheJXtbeWpXiwQP505v/view

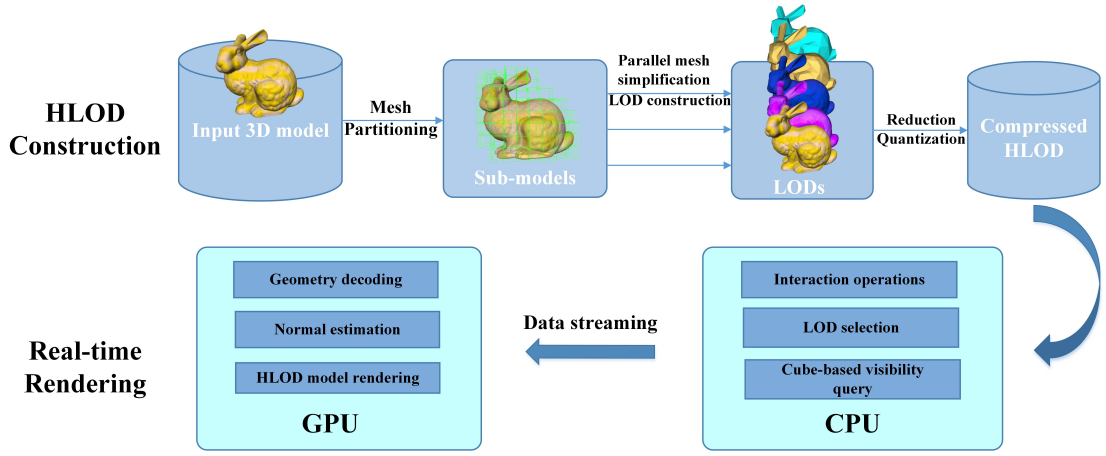


Figure 4.19: The overview of the 3D viewer, which includes HLOD construction and real-time rendering.

interactive frame rate on a computer with integrated GPU or discrete GPU. The overview of the 3D viewer is shown in Figure 4.19.

Chapter 5

Application to terrains

In this chapter, we will address the challenges of reconstructing 3D terrain models from airborne LiDAR point cloud data while preserving the terrain's intricate features. Additionally, we will discuss the real-time visualization of large-scale terrain models, which remains a challenging task. This chapter will cover the following aspects:

(1) The reconstruction of a 3D terrain model using Digital Elevation Models (DEMs) and LiDAR point cloud data.

(2) The construction of a multi-resolution model of the large-scale terrain.

(3) The implementation of a terrain rendering application that leverages the generated multi-resolution model to achieve real-time visualization.

5.1 3D terrain model reconstruction

In this section, we will introduce two methods for reconstructing large terrain models: one based on DEM and the other on LiDAR point clouds. DEM-based terrain models are gridded elevation data, where each grid cell represents a single elevation value. The grid resolution determines the level of detail in the terrain model, making it suitable for representing large areas with relatively uniform terrain features, such as rolling hills or flat plains. In contrast, point cloud data represents the terrain surface as a collection of 3D points, with each point representing a specific location and elevation. This approach offers higher spatial resolution and accuracy compared to DEMs, making it ideal for capturing intricate terrain features like canyons, cliffs, and overhangs. In summary, DEMs provide gridded elevation data that can represent large areas with relatively uniform terrain features using smaller data amounts, while point clouds provide highly detailed 3D points capable of capturing complex terrain features but with larger data volumes.

5.1.1 Terrain model reconstruction based on DEM

In practice, DEM data is often stored in the form of tiles, which are small rectangular or square subsets of the entire dataset. Storing DEM data in tiles offers the advantage of efficient storage and retrieval of specific regions or areas of interest, rather than having to load the entire dataset all at once.

Given a DEM tile as input, the mesh generation process relies on the original elevation data represented by geographic coordinates (x, y, z). The initial step involves normalizing the geographic coordinates. Subsequently, a triangle soup is created using these normalized positions. Finally, a compact indexed mesh representation is generated, encompassing unique vertices and indices information. During the mesh reconstruction for each tile, it is essential to ensure that the resulting tile mesh clusters can seamlessly connect to adjacent tiles.

The stitching process is illustrated within the region enclosed by the orange dashed line in Figure 5.1. Opting for the bottom-left tile as the master tile necessitates information from its neighboring tiles: the upper tile (red vertex in Figure 5.1), the upper-right tile (yellow vertex in Figure 5.1), and the right tile (blue vertex in Figure 5.1). This information is used to refine the boundary mesh of the master tile, ultimately creating a mesh tile that can be seamlessly joined with other adjacent tiles.

Algorithm 5 Terrain mesh tile reconstruction based on DEM

Input : The number of DEM tiles along the X and Y axes: w, h;
The regular grid tile : $Tile[i][j], 0 \leq i < w, 0 \leq j < h$;
for $i \leftarrow 0$ **to** w **do**
  for $j \leftarrow 0$ **to** h **do**
    GetAdajacentInformation($Tile[i+1][j], Tile[i][j+1], Tile[i+1][j+1]$);
    BuildTriangleSoup($Tile[i][j]$);
    $M(Tile[i][j]) \leftarrow$ GenerateIndices($Tile[i][j]$);
    WriteTileMeshFilePath($M(Tile[i][j])$);
  end
end
for $i \leftarrow 0$ **to** w **do**
  for $j \leftarrow 0$ **to** h **do**
    StitchMeshTile($M(Tile[i][j])$);
  end
end
Output : The terrain mesh data file: M, $M(Tile[i][j])$.

The overview of terrain mesh tile construction based on DEM is shown in Algorithm 5. In this algorithm, $Tile[i][j]$ corresponds to the (i-th, j-th) DEM tile, and $M(Tile[i][j])$ is the associated mesh for $Tile[i][j]$. GetAdajacentInformation() obtains the vertex coordinate information of surrounding adjacent tiles (as shown in Figure 5.1). BuildTriangleSoup() constructs the triangle

soup by using elevation data. GenerateIndices() generates the compact indexed mesh representation. WriteTileMeshFilePath() saves data to disk for the current tile. StitchMeshTile() stitches the mesh tiles together and generates new indices for the final whole mesh model. It is important to note that $M(Tile[i][j])$ can serve as the unit for constructing a multi-resolution model, such as a node in the HLOD structure.

In the case of very large-scale terrain surfaces, directly constructing a surface representation by stitching individual mesh tiles can be highly inefficient, resulting in a large amount of data. To address this, the bottom-up multi-resolution construction method can be employed to generate an irregular triangular terrain model. This process involves selecting an appropriate mesh simplification constraint. The details of the multi-resolution model construction will be discussed in section 5.2.1.

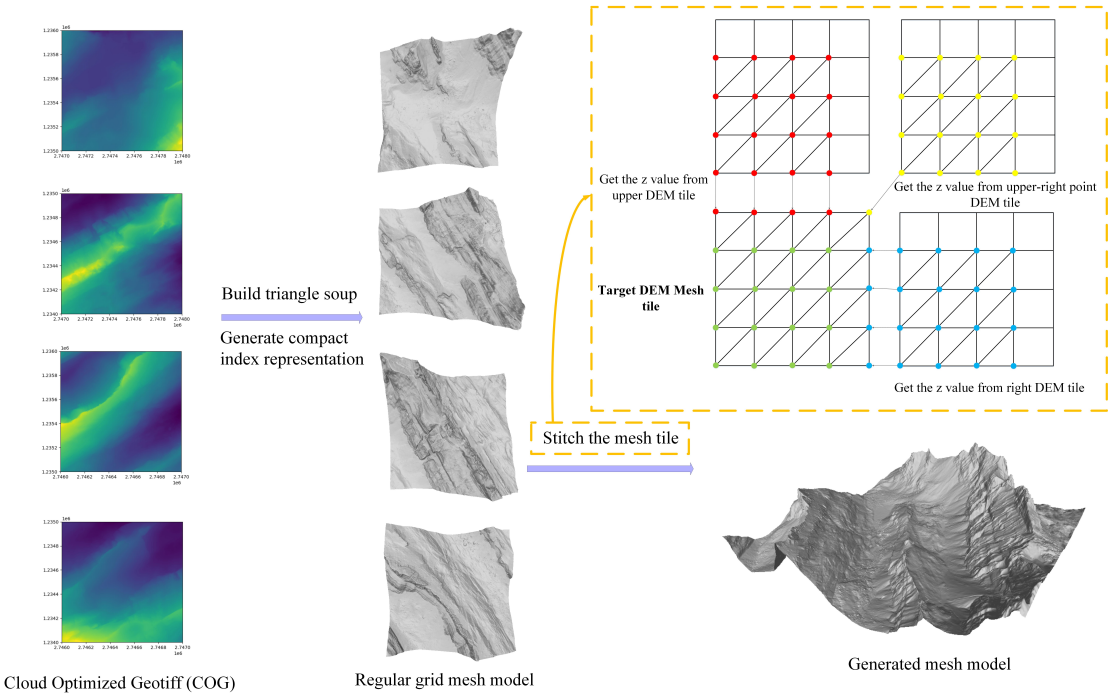


Figure 5.1: The 2D schematic of the terrain model generation based on the DEM. The orange dotted area represents the process of DEM-based mesh tile stitching.

5.1.2 Terrain model reconstruction based on LiDAR point cloud

The reconstruction of a terrain model based on LiDAR point cloud data involves several key steps. First, the point cloud data undergoes preprocessing to remove non-terrain points. It is assumed that the sensor information, such as the LiDAR sensor's position, orientation, and scanning pattern, is available in its original data format. Next, the normals of the points are estimated and combined with the sensor information. Subsequently, a screened Poisson reconstruction

[KH13] algorithm is applied to generate a water-tight 3D model of the terrain. This algorithm utilizes the estimated normals to calculate a signed distance function for each point in the point cloud. This signed distance function is then employed to reconstruct the surface of the terrain as a mesh, creating a continuous representation. Once the surface mesh model is obtained, optimization of the mesh can be performed to enhance its quality and efficiency. Finally, after generating surface meshes for individual tiles, a mesh tile stitching method is introduced to merge the irregular mesh tiles into a cohesive and seamless representation of the entire terrain model.

Normal estimation

The quality of the 3D reconstruction heavily depends on the accuracy of normal estimation. Therefore, the initial crucial step in our method is the estimation of normals from the input point cloud data. The point cloud data is typically stored in a standard file format, including information about the LiDAR scan information. Combining LiDAR source statistics is crucial for the normal estimation process. For instance, scan lines provide valuable information about the relationships between neighboring points, which is essential for achieving accurate normal estimations. The overall process for estimating normals is as follows:

(1) Load the point cloud data from a file, and preprocess the data to extract the terrain point cloud P (if the file contains different types of points).

(2) Estimate the flight lines FL_i based on the statistics of LiDAR scan information (such as GPS time and scan angle). These flight lines, as illustrated in Figure 5.2, are estimated by deriving the across-track azimuth θ_{across} and along-track azimuth θ_{along} from the points. The θ_{along} can be estimated based on the points $p_{start_{gps}}$ and $p_{end_{gps}}$, which are associated with the start and end GPS time:

$$\theta_{along} = atan(\frac{p_{end\_gps}.y - p_{start\_gps}.y}{p_{end\_gps}.x - p_{start\_gps}.x}).$$

The θ_{across} can be estimated based on the points p_{max_angle} and p_{min_angle}, which are related to the maximum and minimum scan angle of each point:

$$\theta_{across} = atan(\frac{p_{max\_angle}.y - p_{min\_angle}.y}{p_{max\_angle}.x - p_{min\_angle}.x}).$$

This angle allows us to approximate the LiDAR scan angle range in current FL_i.

(3) The position of each point is rescaled and normalized to the unit cube. The normalization process can ensure that the point cloud data has a consistent scale and distribution during Poisson reconstruction, avoiding instability or distortion caused by differences in coordinate values.

(4) Build KD-Tree for the re-scaled points to enable efficient point queries.

(5) Estimate the non-oriented normal based on Principal Component Analysis (PCA). Choose a set of points (e.g., a neighborhood of k points) around the target point p_i by KNN search. And then we compute the covariance matrix $\boldsymbol{C}$ of the points, which is as follows:

$$\boldsymbol{C} = \frac{1}{k}\sum_{i=1}^{k}(p_i - \overline{p})(p_i - \overline{p})^T,$$

$$\boldsymbol{C} \cdot \vec{v_j} = \lambda_j \cdot \vec{v_j}, j \in \{0, 1, 2\},$$

$$\lambda_0 \leq \lambda_1 \leq \lambda_2.$$

Where k is the number of neighbor points of p_i, $\vec{p}$ represents the mean of the nearest neighbors, λ_j is the $j-$th eigenvalue of the covariance matrix, and $\vec{v_j}$ is the relevant eigenvector. The eigenvector corresponding to the smallest eigenvalue is the normal vector $n_{non-oriented}$ of the plane that best fits the neighborhood of points.

The eigenvalues demonstrate the quality of normal estimation. If the smallest eigenvalue $\lambda_0 \ll \lambda_1, \lambda_2$, the ellipsoid looks like a disk, which represents the good normal estimation result. If $\lambda_0 \approx \lambda_1 \ll \lambda_2$, the ellipsoid will be cigar-shaped, which makes it difficult to fix the orientation of the normal. If three eigenvalues are similar, the ellipsoid looks like a sphere and the orientation of normal has many possibilities.

After conducting PCA plane fitting for each point cloud to obtain normals, we also calculated the quality measure, denoted as q_i, which falls within the range of 0.0 to 1.0. This quality measure is used to assess the quality of the fitted planes. The overall quality metric q_i is determined by combining by two quality measures, *qual1* and *qual2*. *qual1* computes a quality measure by considering the ratio of the smallest eigenvalue (λ_0) to the second smallest eigenvalue (λ_1). It reflects the estimated ellipsoid shape, which further reflects the normal estimation result. The calculation of *qual2* is based on the perpendicular distance from the point to the estimated plane, which can check if this point is the outlier point. The quality is calculated by multiplying the first quality and the second quality empirically. The detailed calculation process of q_i is as follows pseudo code:

```
// minEval: the smallest eigenvalue
// secondMinEval: the second small eigen value
float qual1 = (secondMinEval == 0) ? 0:1 - minEval / secondMinEval;
// average: average value of points
float perp_error = abs(dot(normal, points[0] - average));
// maxEval: the largest eigenvalue
float qual2 = max(1 - 3 * (perp_error * perp_error) / maxEval, 0);
float q_i = qual1 * qual2;
```

However, note that the orientation of the normal computed by PCA can be ambiguous.

(6) Fix the orientation of each $n_{non-oriented}$ by combining with the scan angle θ_{scan} of p_i and the azimuths of its flight line. The approximate *beam* is estimated as follows:

$$\begin{aligned} beam.x &= cos(\theta_{across}) \cdot sin(\theta_{scan}), \\ beam.y &= sin(\theta_{across}) \cdot sin(\theta_{scan}), \\ beam.z &= -cos(\theta_{scan}). \end{aligned}$$

Then calculate the dot product of $n_{non-oriented}$ and *beam* to set the correct normal orientation, if the normal orientation of the current point is similar to the *beam*, its normal orientation should be corrected. The pseudo-code is as follows:

```
if (fabs(beam · n_non-oriented) > SCAN_TOLERANCE - 2 * (1 - q_i)){
    if(beam · n_non-oriented > 0){
            n_oriented = - 1.0 *  n_non-oriented;
    }
    else{
        n_non-oriented ← unsettled;
    }
}
```

Where the SCAN_TOLERANCE is a preset value, it is set to be 0.25. However, some points, such as the one represented by p_k in Figure 5.2, its normal is perpendicular to the beam direction, which can pose a challenge for accurate normal orientatoion estimation.

(7) For the normal orientation that is difficult to estimate combined with *beam* information, we estimate its orientation by propagating the direction of its neighbor. Given a distance threshold d_t, the KNN search algorithm is employed to identify the neighboring points within this distance threshold for each non-oriented point. For each non-oriented point, its orientation is fixed by propagating the orientation of its neighbor points which have fixed normal orientation. The quality value of each neighboring point determines its weight in the majority value. If the majority value is negative, the normal direction will be corrected to the opposite direction. The pseudo-code is as follows:

```
// Record the accumulation
float majority = 0.0;
// Search the neighbor points by KNN
while(p_j in neighbour points of p_k){
    if (n_pj is settled){
        if (fabs( p_k · p_j) >  1.0 + FLIP_TOLERANCE - q_i){
            majority += q_i * (p_k · p_j);
        }
     }
}
```

```
if( majority < 0 ){
  n_{p_k} = - 1.0 * n_{p_k};
}
```

Where the FLIP_TOLERANCE is a preset value, we set it to 0.35. The iterative propagation continues until all non-oriented normals have been corrected and settled. After the correction of the normal estimation for each point, the points without fixed orientation will be skipped. At last, the oriented point cloud $P_{oriented}$ is obtained.

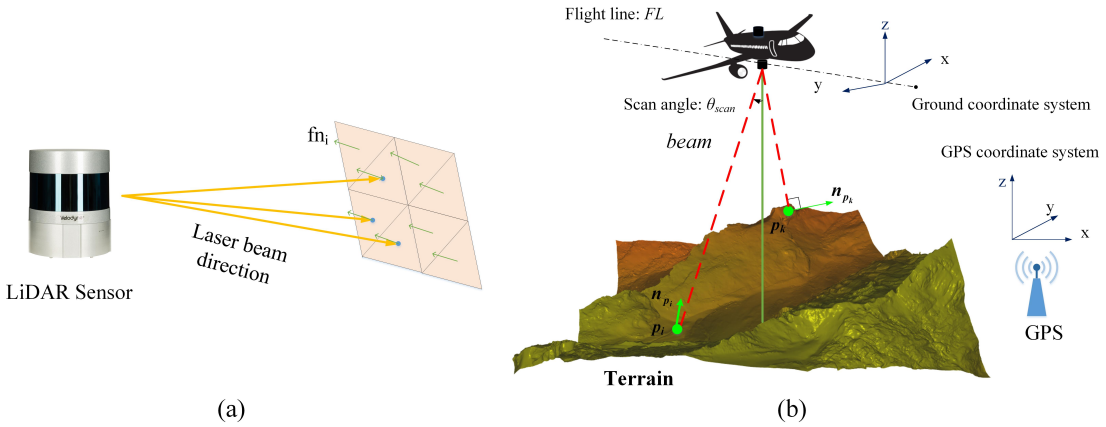


Figure 5.2: (a) Relation of the face normals and the laser beam direction. (b) Normal estimation combined with LiDAR scan information.

Surface reconstruction

The screened Poisson reconstruction [KH13] is used for generating the watertight mesh model for oriented point cloud, which is an improved version from [KBH06] that generates the overly smooth surface. The basic concept behind Poisson reconstruction is that the point cloud provides information about the object's surface position, with its normal vectors representing the inside and outside of the surface. By formulating surface reconstruction as a Poisson problem, it aims to fit the indicator function $\mathcal{X}$, which is defined as 1 inside the surface and 0 outside the surface. Given a set of oriented point samples $s \in S$, a smoothed filter function $\widetilde{F}(q)$, and $\widetilde{F}_p(q) = \widetilde{F}(q-p)$, the vector field $\vec{V}$ represents the normal surface. The relationship between the indicator function, smoothed filter function, and the vector field is as follows:

$$\triangle(\mathcal{X} * F)(q) \approx \sum_{s \in S} |\mathscr{P}_s| \widetilde{F}_{s.p}(q) s.\vec{N} \equiv \vec{V}(q), \tag{5.1}$$

where $s.p$ and $s.\vec{N}$ represent the position and normal of s, and $|\mathscr{P}_s|$ represents the size of patch area associated with s. And then the smoothed indicator function can be $\widetilde{\mathcal{X}} = \mathcal{X} * \vec{V}$, the

iso-surface based on input points can be obtained by computing the average function value. So we need to find a function $\widetilde{\mathcal{X}}$ whose gradient is close to the vector field of input points. This problem can be transformed into a Poisson problem:

$$\Delta\widetilde{\mathcal{X}} = \nabla \cdot \vec{\mathcal{V}}, \tag{5.2}$$

which can be solved by selecting a set of basis functions, and transforming this equation into a linear system of finite elements. Given a hierarchy octree of maximum depth D, the basis functions are decided by the depth d of their associated nodes, the nodes of depth d can be expressed as the B-spline function set $\{B_1^d, ...B_{N_d}^d\}$, and the linear system can be defined as follows:

$$A^d x^d = b^d, \tag{5.3}$$

where $\widetilde{\mathcal{X}}$ is written as the sum of basis functions

$$\widetilde{\mathcal{X}} = \sum\nolimits_{d=0}^{D} \sum\nolimits_i x_i^d B_i^d. \tag{5.4}$$

After solving the Poisson equation, the zero level-set of obtained $\mathcal{X}$ corresponds to the desired surface. However, in practice, the $\mathcal{X}$ can deviate from the true indicator function because of the noisy point sampling, outliers, and the approximated point density during the octree construction. [KH13] adds an interpolation constraint to the element

$$A_{ij}^{dd'} = \langle \Delta B_i^d, \Delta B_j^{d'} \rangle_{[0,1]^3} + 2^d \alpha \langle B_i^d, B_j^{d'} \rangle_{(w^d, \mathcal{P}^d)}.$$

For $(w^d, \mathcal{P}^d)$, $p_i \in \mathcal{P}^d$ is the weighted average position of the points falling into octree node i at depth d, and $w^d(p_i)$ is the sum of the associated weight.

The implementation of screened Poisson reconstruction steps are as follows:

(1) Voxelization: convert the oriented point cloud $P_{oriented}$ into a voxel grid representation, where each voxel represents a small 3D volume element and is organized by octree. The maximum depth D of octree can be chosen based on the desired level of detail.

(2) Weight computation: assign weights to each octree node based on its proximity to the input points and the estimated surface normals. These weights determine the influence of each voxel on the reconstructed surface.

(3) Poisson equation setup: set up and solve the Poisson equation using the voxel grid and the computed weights. The Poisson equation relates the Laplacian of the potential field to the divergence of the weighted gradient field.

(4) Poisson equation solving: solve the Poisson equation numerically to obtain the potential field values for each voxel. This can be done using iterative solvers or other numerical techniques.

(5) Surface extraction: extract the reconstructed surface from the potential field solution. Marching cubes [LC98] is employed for surface extraction, however, this method can potentially lead to the generation of elongated triangles within the mesh model.

Following the acquisition of the mesh model through screened Poisson reconstruction, a series of post-processing steps are executed. Initially, triangles lying outside the boundaries of the input point cloud region are eliminated. Subsequently, the mmg [INP04] tool is employed to enhance and optimize the mesh quality.

Mesh tile Stitching

Given the substantial size of point cloud data, creating a large terrain model (exceeding $10km \times 10km$) demands considerable processing power and memory usage. However, the size of individual mesh tiles (approximately $1km \times 1km$) generated from this data is relatively small, typically a few dozen megabytes. Therefore, it's more efficient to first generate these smaller mesh tiles and then stitch them together to form the complete terrain model. Nevertheless, because the mesh model derived from point cloud data is irregular, the borders of each tile do not align perfectly. To address this challenge, we have developed an algorithm for stitching mesh tiles together, relying on their border vertices (as illustrated in Figure 5.3).

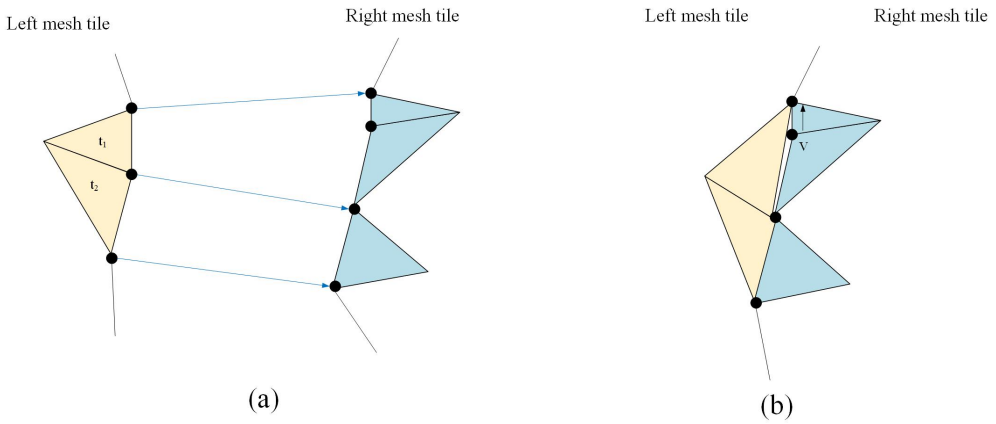


Figure 5.3: Mesh tile stitching. (a) Find the matching points on the borders of two mesh tiles. (b) Collapse the middle vertex v to the nearest vertex to prevent cracks.

Our stitching algorithm involves two steps, using two adjacent tiles: the left tile (master tile) and the right tile (slave tile). In the first pass, we match the vertices from the left tile to the nearest vertices in the right tile. During this process, we project the left tile's vertices onto the right tile's vertices. However, in some cases, vertices in the middle of two projected vertices might lead to cracks, resulting in a non-conforming mesh. In the second pass, we address this issue by collapsing vertex v (as shown in Figure 5.3 (b)) to its nearest vertex.

Overall implementation

The model generation process based on LiDAR point cloud data is outlined in Algorithm 6. Here are the key operations: AnalysisRawData() involves analyzing the LiDAR scan information to extract flight lines and azimuths. These metadata are crucial for subsequent processing. NormalEstimation() estimates the normal direction for each terrain point cloud. ScreenPoissonReconstruction() applies the screened Poisson algorithm to generate a watertight 3D mesh terrain model. SurfaceTrimmer() removes vertices that do not belong to the terrain. This process ensures that only relevant points are considered for the subsequent mesh generation. Remesh() improves the quality and topology of the generated mesh. We use mmg (a surface and volume remeshers [INP04]) to generate isotropic mesh, which has a more uniform distribution. Once all the mesh tiles are generated, they are stitched together column by column. This process combines individual tiles into a complete mesh terrain model.

Algorithm 6 Terrain mesh reconstruction based on LiDAR point cloud

Input : The number of point cloud tiles along the X and Y axes: w, h;
The point cloud tile : $PC[i][j], 0 \leq i < w, 0 \leq j < h$;
for $i \leftarrow 0$ **to** w **do**
  for $j \leftarrow 0$ **to** h **do**
    AnalysisRawData($PC[i][j]$);
    NormalEstimation($PC[i][j]$);
    $M(PC_{oriented}[i][j]) \leftarrow$ ScreenPoissonReconstruction($M(PC_{oriented}[i][j])$);
    SurfaceTrimmer($M(PC_{oriented}[i][j])$);
    Remesh($M(PC_{oriented}[i][j])$);
  end
end
for $i \leftarrow 0$ **to** $w - 1$ **do**
  for $j \leftarrow 0$ **to** $h - 1$ **do**
    $M \leftarrow Stitch$(M(PC$_{oriented}[i][j]$), $M(PC_{oriented}[i+1][j])$);
  end
end
Output : Terrain mesh data M.

5.2 Multi-resolution construction

5.2.1 Out-of-core construction

Once the meshes tiles are obtained during the terrain model reconstruction, they can serve as the basic units (T_L) for the highest resolution LOD, denoted as M_L. The choice of the level L can be a preset value, which is determined based on factors like the total number of tiles available.

With the selected level L and the mesh tiles T_L, the initialization of the Octree-based Out-of-core Mesh (OOM) tree structure is performed, and the logical coordinates distribution is established. The simplification sequence of M_L can be viewed as a multi-resolution structure, allowing for different levels of detail representation:

$$M_L \geq M_{L-1} \geq ... \geq M_0.$$

We choose to build blocks of size $n \times n$, and the block selection method is detailed in section 3.2.4. The QEM-based edge collapse is used to reduce the number of triangles, however, the triangles at the boundary of each block will not be simplified.

During the simplification, the child-parent dependency is initialized, and we note $P(v)$ as the parent of the vertex v which means this vertex collapses to the $P(v)$. We set a simplification constraint

$$d(v, P(v)) \leq \sigma_0 2^{-L},$$

where the σ_0 is the first parameter of our multi-resolution model method, which can be adjusted based on the edge portion of the tile, for each edge of M_L, the maximum length is $\sigma_0 2^{-L}$. The consequence of this condition is that for each $1 \leq l \leq L$, each $v \in M_l$,

$$d(v, P(v)) \leq \sigma_0 2^{-(l+1)}. \tag{5.5}$$

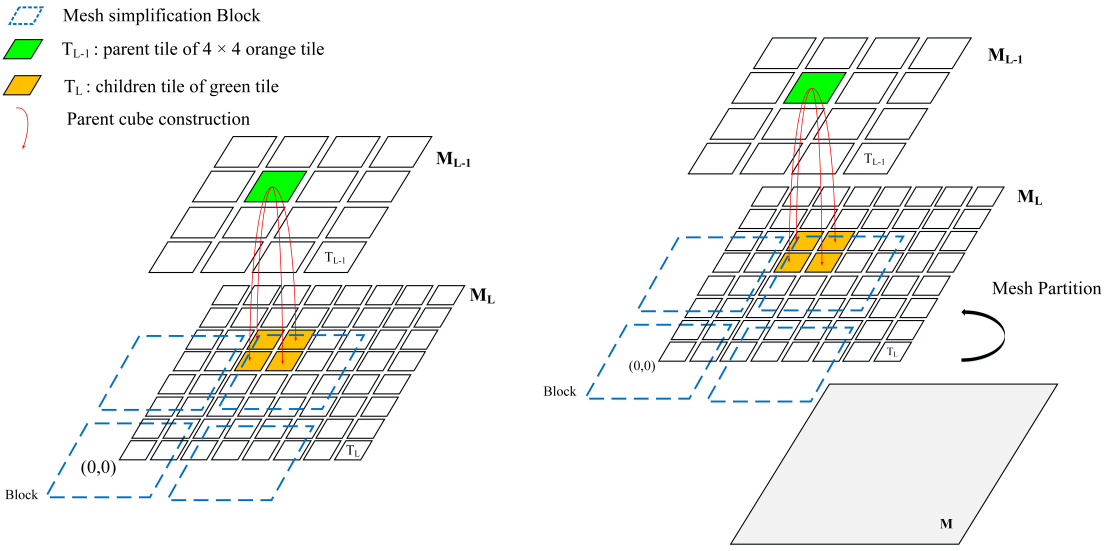
Once we have the simplified meshes, we split them into 2×2 tiles at a time to construct the parent tile T_{L-1}. We use the notation $P(T_L) = T_{L-1}$, indicating that the single tile T_{L-1} in M_{L-1} contains T_L. The Quadtree-based splitting of simplified meshes ensures that the child-parent relationships can be directly established and updated.

The entire multi-resolution structure is built recursively in a bottom-up manner until the desired LOD level is reached. This construction process for the multi-resolution terrain is executed in an out-of-core manner, where only the selected tiles need to be loaded into the main memory for relevant operations. The independent nature of different tiles enables parallel processing, further enhancing efficiency.

The multi-resolution construction process of the DEM-based terrain model can also produce models consisting of irregular triangles at different resolutions, as shown in Figure 5.4 (a). This conversion can be beneficial when dealing with DEMs that have a high density of elevation points, as it allows for efficient data management and improved rendering performance.

5.2.2 In-core construction

For terrain models that can fit within the available RAM memory, the construction of multi-resolution can be performed in in-core mode. The schematic diagram of the process is shown in Figure 5.4 (b). Given an input terrain model M, the first step is to construct the highest-resolution model M_L. At the top level L, we split the model according to regular cubes C_L of size



(a) Out-of-core multi-resolution construction

(b) In-core multi-resolution construction

Figure 5.4: The 2D schematic of out-of-core multi-resolution construction (a) and in-core multi-resolution construction (b).

2^{-L}. The mesh is partitioned based on its barycenter, and we do not cut the triangles according to the grid, even if some triangles might fall outside of the cube. To handle the geographic and landform features at different heights in the terrain model generated from LiDAR point cloud data, we divide the model in 3D space.

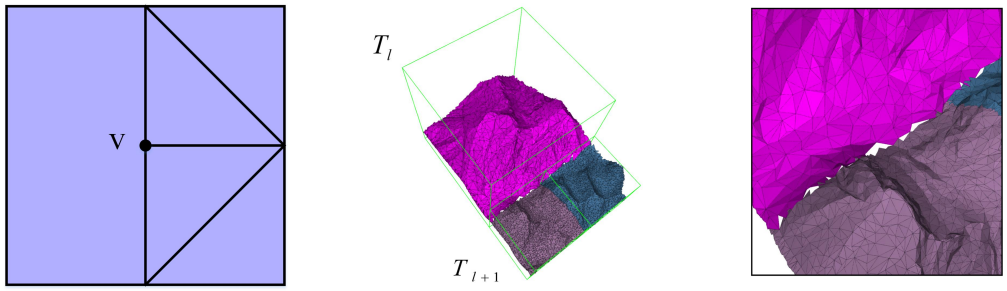
Based on the mesh partition of the highest-resolution model M_L, we apply the same simplification strategy as discussed in section 5.2.1. Following the block-based mesh simplification, we split the resulting simplified meshes into $2 \times 2 \times 2$ cubes to create a parent cube C_{l-1}. We also generate a compact mesh representation (vertex attribute buffer and index buffer) for each C_{L-1} and establish the child-parent dependency after the mesh simplification and parent cube construction.

For the remaining LODs, we proceed with mesh simplification for each LOD layer sequentially. The process involves simplifying the mesh representation at each LOD level and constructing a coarser LOD based on the simplified results. This iterative approach continues until the preset LOD level is achieved.

For large-scale terrains that exceed the available memory capacity, it is necessary to employ out-of-core construction techniques. It only requires saving each tile of M_L as a separate file after the construction of the highest LOD. Then, the out-of-core multi-resolution construction process, as described in section 5.2.1, can be performed.

5.3 Real-time rendering

In the real-time rendering of terrain based on the multi-resolution model, several visual artifacts can arise, including LOD popping occurring during the LOD switching, mesh cracks resulting from the different resolutions of LOD parts (Figure 5.5 (b)), and T-junctions caused by floating point rounding errors (Figure 5.5 (a)). These artifacts can disrupt the visual quality and immersion of the rendered terrain.



(a) A T-junction of vertex v

(b) Mesh cracks between different LODs

Figure 5.5: Visual artifacts that may occur during rendering: (a) A T-junction of vertex v. (b) Mesh cracks are caused by the LOD parts at different resolutions.

Based on the storage method of different multi-resolution structures, two types of rendering methods are designed. For models that can fit into memory, the in-core rendering mode can be used. For models that exceed the memory capacity, such as multi-resolution models based on DEM tiles, out-of-core rendering technology will be employed.

5.3.1 Adaptive and continuous LODs

The visibility query is based on the current position of the viewpoint v_p. The terrain tile T_l is said to be suitable for visualization if either $l = L$ or

$$d(v_p, T_l) \geq \kappa_0 2^{-l}, \tag{5.6}$$

where κ_0 is a real-time constant during rendering, d represents the distance from the viewpoint to the T_l.

After the visibility query, we might obtain two tiles: T_l and T_{l+1}, which are neighbors and both retained for visualization. As illustrated in Figure 5.5 (b), the purple tile T_l corresponds to the coarser LOD M_l, and the brown cube T_{l+1} corresponds to the LOD at higher resolution M_{l+1}. If we draw the geometry of these two tiles directly, mesh cracks occur at the border between the two tiles.

We derive some estimates for interpolating the position of a vertex with its parent at different resolutions. We consider the vertex $v \in V(T_l)$, the distance inequality relationship from the

viewpoint to the vertex v:

$$d(v_p, v) \geq 2(\kappa_0 - \sigma_0)2^{-(l+1)}, \tag{5.7}$$

the distance from the viewpoint to the parent of this vertex

$$d(v_p, P(v)) \leq (\kappa_0 + 1 + \sigma_0)2^{-l}, \tag{5.8}$$

An interpolation factor $\lambda \in [0, 1]$ can be derived based on the blending function $\lambda = b(d(v_p, v)/2^{-l})$, the blending range begins at $d_{start} = (1 + \kappa_0 + \sigma_0)2^{-l}$, and ends at $d_{end} = (\kappa_0 - \sigma_0)2^{-(l-1)}$, so the interpolation factor λ can be computed as follows:

$$\lambda = clamp((d_{end} - d(v_p, v)/(d_{end} - d_{start}), 0.0, 1.0). \tag{5.9}$$

As a result, the view-dependent and LOD-dependent vertex position will be

$$\omega = \lambda v + (1 - \lambda)P(v). \tag{5.10}$$

For each vertex, its resulting position depends on the real-time viewpoint. When the viewpoint is at the minimum distance d_{start}, the resulting position is the vertex itself, denoted as $\omega = v$. Conversely, when the viewpoint is at the maximum distance d_{end}, the resulting position is the position of its parent, denoted as $\omega = P(v)$.

In practical implementation, we typically choose a parameter κ_0 with an initial value of 4.0. The choice of κ_0 can be adjusted in conjunction with another parameter σ_0. The interpolation band can be computed as $(\kappa_0 - 3\sigma_0 - 1)2^{-l}$, where l is the level of LOD. This computation ensures a relationship between κ_0 and σ_0 such that $\kappa_0 > 1 + 3\sigma_0$, by properly setting the parameters κ_0 and σ_0, we can achieve effective interpolation of vertex positions based on the viewpoint distance.

During real-time rendering, the value of κ_0 can be adjusted to different values. Increasing the value of κ_0 expands the range of interpolation, resulting in a smoother transition between different levels of detail. This flexibility allows for adaptability to different rendering scenarios and desired visual effects.

For the other attributes, such as texture coordinates, color, and normal. We use the same interpolation coefficient λ to blend these attributes with their parents. Applying the same interpolation coefficient λ to all attributes can ensure consistency and coherence in the interpolation process, especially for the texture mapping, the interpolation helps preserve the visual expression.

5.3.2 In-core rendering

In-core rendering is conducted as long as the model's size is within the memory capacity. Before the rendering process begins, the geometry data of the multi-resolution model structure is directly uploaded to the GPU. To optimize data access, the memory offset of each tile's data is recorded during the memory allocation phase. Storing the memory offset allows the rendering pipeline to

efficiently retrieve the data of each required cube, providing direct access to the cube's data and avoiding unnecessary memory lookups.

During the rendering loop, the HLOD tree traversal is executed to determine the visibility of individual cubes based on the viewpoint position. This traversal process starts from the root and descends down to the leaf nodes, prioritizing objects based on their distance from the camera. For each cube in the hierarchy, frustum culling is performed to determine its visibility, which checks if the cube is within the view frustum of the camera. If a cube passes the frustum culling test and satisfies the visualization conditions, its logical coordinate is stored in a stack.

For each draw call, the cubes stored in the stack are popped in a depth order. This means that cubes closer to the viewpoint are rendered first, followed by those that are farther away. This depth-based ordering aligns with the principles of the modern rendering pipeline. By rendering cubes in a depth order, we prioritize the rendering of objects that are more likely to be visible to the viewpoint. This approach optimizes the rendering process by reducing overdraw and minimizing unnecessary computations for objects that may be occluded by others.

5.3.3 Out-of-core rendering

The construction process of a terrain model based on DEM mesh tiles involves creating an out-of-core multi-resolution model. Therefore, out-of-core rendering can be directly applied. Unlike in-core rendering, where all data is loaded into memory at once, out-of-core rendering operates with limited memory capacity, necessitating a more dynamic and memory-efficient approach.

To efficiently manage memory and minimize rendering stuttering due to data streaming from disk, we employ a multi-threaded approach with two separate threads: one for rendering and one for data streaming. Each thread maintains its own stack to manage cubes.

During the rendering process, the OOM tree traversal takes precedence, and the rendering thread's stack is primarily updated within this traversal. If a cube meets the visualization criteria (Eq. (5.6)) but its data is not currently present in RAM, its parent cube is also pushed into the rendering stack. This streaming standard ensures that our vertex interpolation method functions correctly, as it requires access to parent vertex data for interpolation. However, it may introduce a delay in achieving the desired rendering result.

The impact of this delay can be reduced by utilizing disk storage types with faster read/write rates, such as SSDs, which have significantly faster data transfer rates compared to traditional HDDs. For further implementation details on out-of-core rendering, please refer to section 4.3.

5.4 Results and comparisons

5.4.1 Data set

The data used to reconstruct the terrain model is from Federal Office of Topography swisstopo, it includes LiDAR point cloud data (swissSURFACE^{3D} [Top23b]), and DEM data set (swissATLI^{3D}

[Top23a]).

The swissATLI^{3D} dataset is a high-precision digital elevation model of Switzerland, provided in the form of a raster grid. It offers two different resolutions: $0.5m$ and $2m$, and each tile covers an area of $1km^2$. For the higher resolution of $0.5m$, the spatial density of the data may not always be sufficient to provide the finest level of detail in the mesh. In such cases, oversampling is applied to enhance the data quality. The data is stored in two data formats: GeoTIFF and ASCII grid. GeoTIFF is a popular format for storing raster data, it allows for georeferencing information to be included in the file, which allows the data to be located in geographic space. ASCII grid represents raster data as plain text, where each line of text corresponds to a row of the raster grid. The swissATLI^{3D} dataset provides high accuracy for elevations below 2000 meters, with an accuracy of approximately $\pm 30cm$. For elevations above 2000 meters, the accuracy ranges from $\pm 1m$ to $3m$.

The 3D LiDAR point cloud data used in this thesis is sourced from swissSURFACE^{3D}. swissSURFACE^{3D} is a dataset generated from airborne LiDAR data and boasts a high point density, containing several million points per square kilometer. These points are distributed irregularly in three-dimensional space, providing a detailed representation of the terrain. Each point in the dataset is characterized by precise X, Y, and Z coordinates and is classified based on predefined criteria, offering information about the nature of the objects or features that were measured. swissSURFACE^{3D} is typically available in the LAS file format, which is well-suited for efficiently handling large datasets while preserving essential LiDAR scan information.

5.4.2 Implementation details and test platforms

The terrain model generation, multi-resolution model construction, and real-time rendering codes have been implemented in C++ on the Linux platform. The rendering scheme is based on the OpenGL graphics library.

In practice, the reconstruction of large-scale terrain models, such as those spanning a size of $64km \times 64km$, requires careful platform selection due to their demanding RAM requirements. In such cases, it is necessary to utilize a cluster computer with a sufficient amount of memory to handle the terrain generation process. The configuration of the cluster is as follows: Intel(R) Xeon(R) Gold 6152 CPU with 256GB RAM.

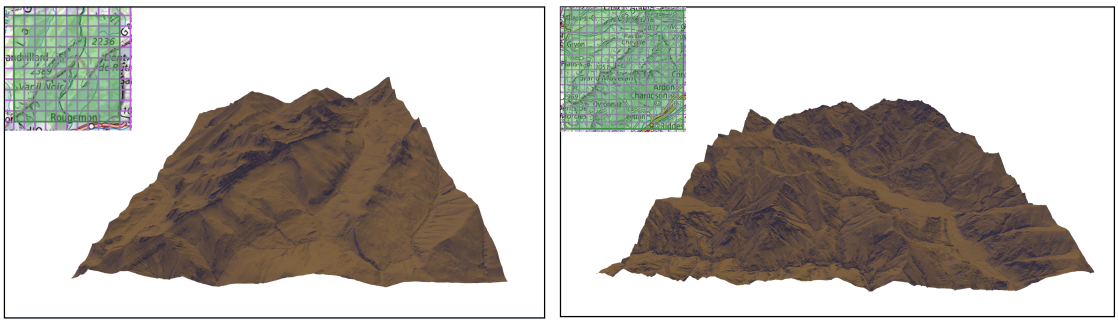
The multi-resolution model construction and real-time rendering were tested on different hardware configurations. A desktop machine, featuring an Intel UHD Graphics 630 and an Intel i7-8700 CPU with 16GB of RAM, was used for testing. Additionally, a laptop with an Nvidia A450 GPU and 16GB of RAM was also utilized. By testing the framework on these different hardware configurations, we can evaluate its performance under varying conditions and ensure that it is capable of delivering real-time rendering of the multi-resolution models effectively.

5.4.3 Real-world terrain model reconstruction

First, the terrain reconstruction result will be introduced. We compare the resulting terrain mesh models reconstructed from DEM data and LiDAR point cloud. The DEM-based terrain model serves as our benchmark, given its widespread use and standardized format for representing terrain data.

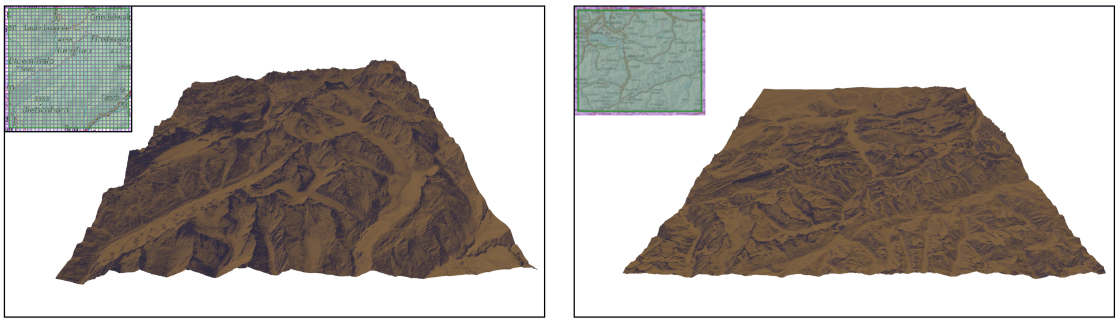
DEM-based terrain reconstruction

We selected four regions to generate DEM-based terrain models at varying scales. These regions include the Vanil Noir region (covering an area of $9km \times 9km$), the Locarno region ($16km \times 16km$), the Jungfrau region ($32km \times 32km$), and the Aldorf region ($64km \times 64km$). Upon downloading all the required tiles, each tile's data has a resolution of 2 meters, with each tile measuring $1km \times 1km$. To enable the seamless stitching of all the tiles, we first generated boundary meshes by combining adjacent tiles. This process resulted in mesh tiles with 251,001 vertices and 500,000 triangles each.



(a) Vanil Noir region terrain
40.5 M triangles

(b) Locarno region terrain
32.0 M triangles



(c) Jungfrau region terrain
110.7 M triangles

(d) Aldorf region terrain
126.2 M triangles

Figure 5.6: Terrain models of different scales (with the number of triangles), the insertion figure in the upper left corner indicates the selection of geographical regions.

Due to the relatively large number of triangles, especially when the region size exceeds $16km \times 16km$, the resulting terrain model can contain up to 128 million triangles. To address this challenge, we also produced simplified versions of the large-scale terrain models by employing the out-of-core multi-resolution model construction. These simplified models are represented by irregular triangles. The four generated models are shown in Figure 5.6.

Terrain reconstruction from LiDAR point cloud

For the reconstruction of the 3D model using LiDAR point cloud data, we selected the Vanil Noir region, which has a spatial extent of $9km \times 9km$. The data density in this region varies, with a minimum of 5 points per square meter (pts/m^2). Although the measurements exhibit significant spatial heterogeneity, the average spatial density ranges between $15pts/m^2$ and $20pts/m^2$.

To process the LiDAR data, we initially performed terrain point filtering and normal estimation as preprocessing steps. Following that, we employed a screened Poisson reconstruction method using a spatial octree with a depth of 11, the code implementation of Kazhdan [Mic23] was used in our pipeline. This reconstruction approach utilizes the spatial organization of the points to generate a smooth and continuous surface representation of the terrain. After the reconstruction process, we further refined the model by removing vertices that do not belong to the terrain. This step ensures that the final 3D model accurately represents the topography of the region, focusing on the relevant terrain features. Finally, we used the mmg library to improve the mesh quality and smooth the surface meshes. Specifically, we utilized the libmmgs library of mmg, which can adapt and optimize a surface triangulation.

Comparison

We compare the reconstructed terrain models from different sources and assess them based on two indicators: the processing time required to reconstruct the mesh tile from different data sources, and the number of triangles in each reconstructed terrain. In addition, the preservation of visual terrain and topography is also a comparative aspect.

In Table 5.1, we have provided the reconstruction times for each tile (measuring $1km \times 1km$) based on LiDAR point cloud and DEM data sources. The experimental tile data was obtained from the Säntis region, using coordinates in the LV95 system [MS13], specifically at $(2746, 1234)$. For the LiDAR-based reconstruction, the original LiDAR data tile contained approximately 20,393,100 points, resulting in a terrain mesh with 1.2 million triangles. In contrast, the original DEM data was at a resolution of $2m$, resulting in a terrain mesh with 500,000 triangles.

The process of reconstructing a 3D model based on DEM data is proved to be significantly faster than generating meshes from LiDAR point cloud data. The DEM-based terrain model reconstruction involves generating a triangle soup from the DEM tiles and subsequently creating an index-based compact model representation. For the terrain model reconstruction from LiDAR point cloud, the process included normal estimation ($45.0s$), screened Poisson reconstruction ($17.5s$), remeshing ($46.0s$), and file writing ($52.9s$).

Table 5.1: Comparsion of construction time (Tile size: $1km \times 1km$)

Data type	DEM	LiDAR point cloud
Time (s)	0.54	161.4

We conducted a comparative analysis of the mesh models generated from DEM data and LiDAR point cloud for the Vanil Noir region ($9km \times 9km$). Table 5.2 presents each model's vertex count (n_v) and triangle count (n_t). The LiDAR point cloud based model demonstrates a significant reduction in triangle count compared to the DEM-based model, achieving nearly three times fewer triangles.

Table 5.2: Quantitative results of reconstructed terrain model (Vanil noir region)

Model	n_v	n_t
DEM-based	20259001	40500000
Point cloud based	6422780	12839353

We conducted terrain reconstruction in the Säntis region of Switzerland using two different data: LiDAR point cloud and DEM. Table 5.3 presents the quantitative results of the generated models in Säntis region, where n_v represents the number of vertices and n_t is the number of triangles. Figure 5.7 provides a visual comparison of different landforms of different regions. The LiDAR point cloud based model outperforms the DEM-based model in preserving the detailed features of cliffs and overhangs. It captures the rugged and vertical nature of the cliffs, while the DEM-based model represents them as narrow and elongated triangles, lacking fine details. Additionally, when it comes to other complex geomorphic features such as overhangs (rock faces or artificial climbing walls with slopes exceeding 90°), the DEM-based approach falls short.

Table 5.3: Quantitative results of reconstructed terrain model (Säntis region)

Model	n_v	n_t
DEM-based	251001	500000
point cloud based	589437	1175924

We present the comparison between the generated Jungfrau terrain model and a realistic photograph, as well as the rendering results using our method and the Google Earth rendering engine in Figure 5.8. For the Jungfrau terrain model, we generated it using both DEM and LiDAR point cloud data. Our rendering approach emphasizes real-time mesh colorization based on per-face normal values and altitude, reducing the reliance on textures. This method allows us to distinguish between different landform features, such as white for snow-covered areas and brown for rocky terrain. The orientation of a triangle face's normal, which represents the direction perpendicular to the surface, plays a crucial role in determining whether snow can adhere to it. If the normal of a triangle is oriented in a way that permits snow accumulation and adherence, then this triangle is more likely to have a snow-covered surface. Conversely, if the normal is oriented

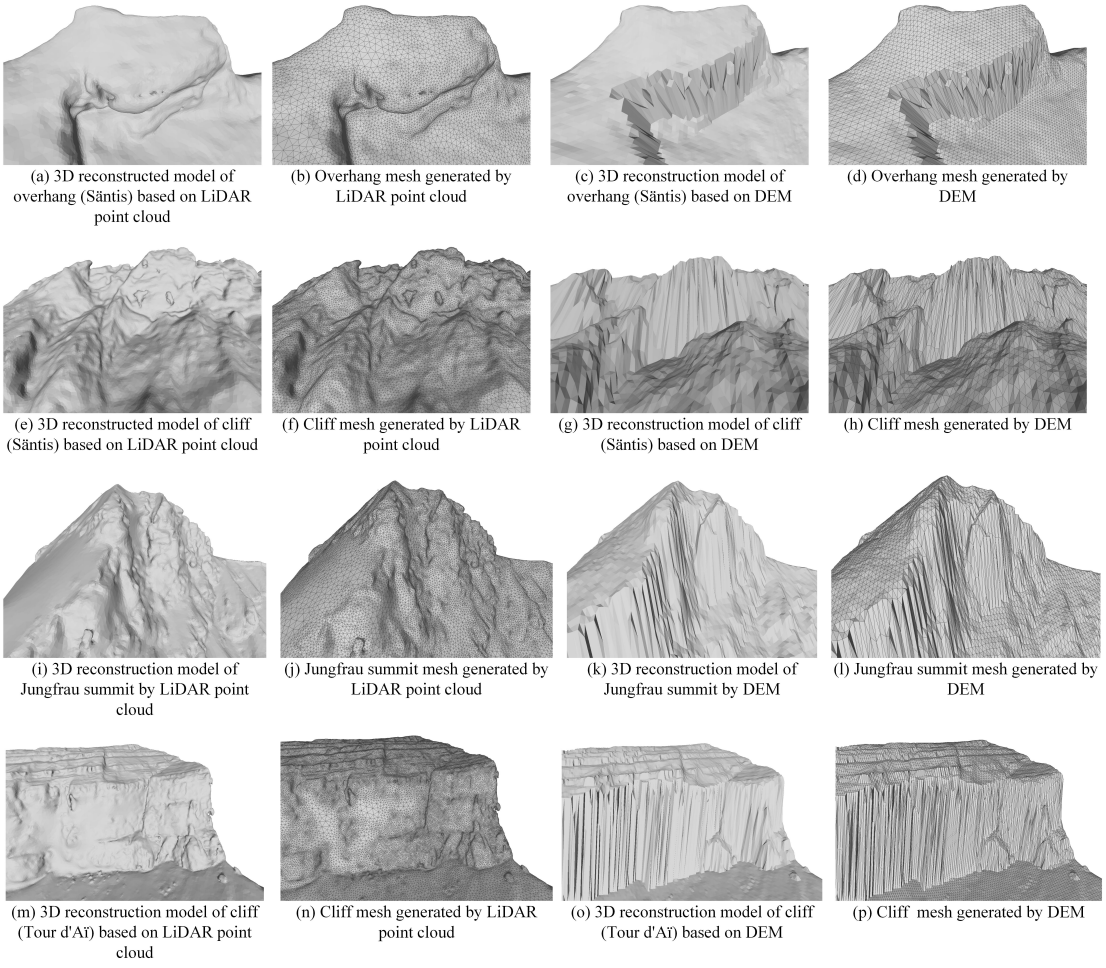


Figure 5.7: Comparison of mesh model generation results for overhangs and cliffs based on DEM and LiDAR point clouds.

in a manner that makes it challenging for snow to stick, this triangle may remain relatively snow-free.

As shown in Figure 5.8, the terrain model reconstructed from the LiDAR point cloud closely resembles the realistic photograph, capturing the intricate details of the landforms. In contrast, the model based solely on DEM data lacks some of the fine landform details, resulting in a less realistic representation. Furthermore, we compared our rendering results with the Google Earth model (Figure 5.8 (d)). Google Earth utilizes aerial or satellite imagery for texture mapping, typically captured from a top-down perspective. These images are then applied as textures onto the terrain model. However, when viewing the terrain from an oblique or bird's-eye angle, the flat imagery projected onto the three-dimensional terrain can cause stretching or distortion, resulting in an unrealistic appearance (Figure 5.8 (d_2)). In contrast, our method focuses on preserving

the landform details without relying on texture mapping. By accurately representing the terrain geometry using the reconstructed mesh, we achieve a more accurate and realistic rendering, even when viewing the terrain from various angles. This approach avoids the stretching or distortion issues associated with texture mapping and ensures a faithful representation of the terrain's features.

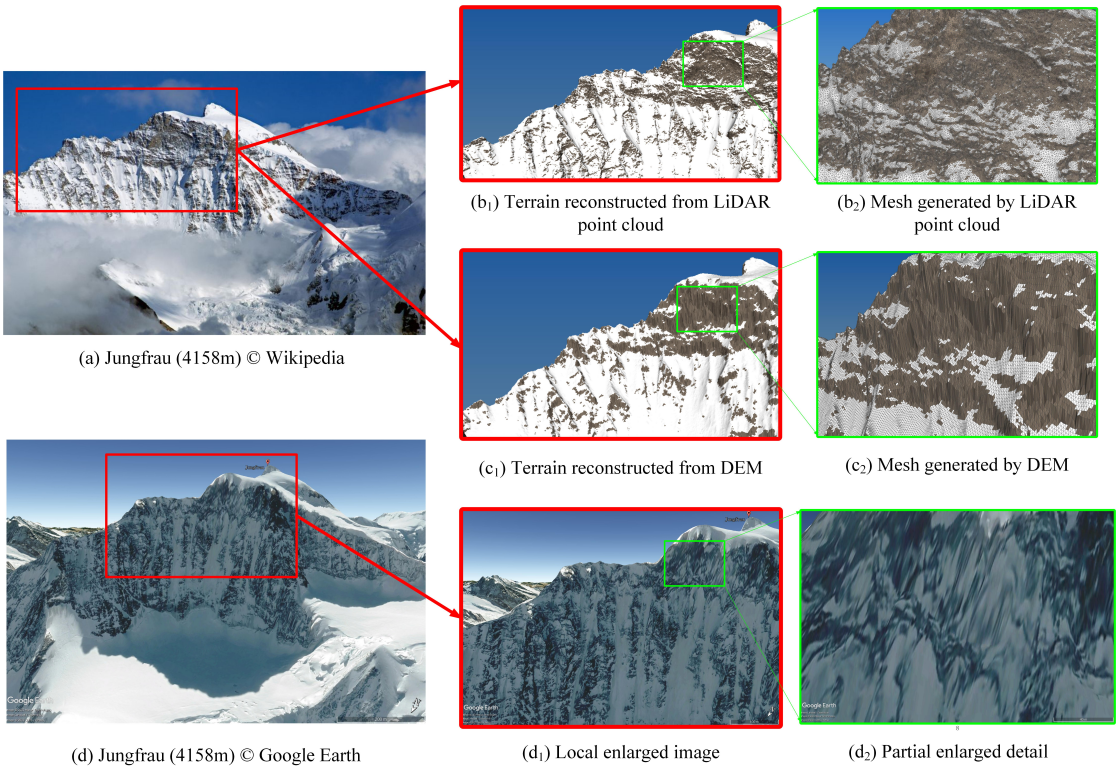


Figure 5.8: (a) Photography of Jungfrau. (b_1) Terrain reconstructed from LiDAR point cloud with the preservation of geographic features (b_2). (c_1) Terrain reconstructed from DEM but lack of detail capture (c_2). (d) The capture of Google Earth. (d_1) Local enlarged image. (d_2) Unrealistic rendering with stretched texture.

We also utilized Unreal Engine 5 to render the generated terrain models, demonstrating that a well-preserved model can produce more realistic rendering results. Unreal Engine 5's lighting system offers various types of lights, allowing for versatile lighting styles. In the rendering results displayed in Figure 5.9, we primarily utilized directional light and skylight in Unreal Engine 5. The depicted mountains include Muveran, Pic sans nom, Cervin, Jungfrau, Dent Blanche, Barre des Écrins, Dent du chat, and Aiguilles Dorées.

In summary, the DEM-based terrain mesh model struggles to adequately preserve the features of complex landforms, such as cliffs and overhangs. Conversely, the LiDAR point cloud based approach offers a more detailed and accurate representation of these features, making it the preferred choice for terrain modeling applications. However, it's important to note that

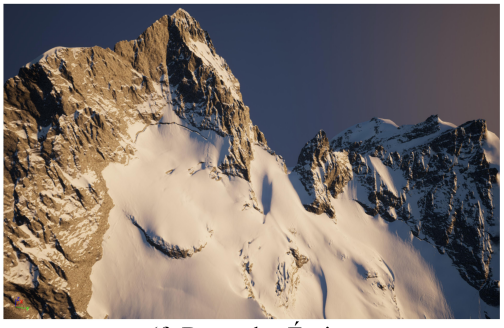
(a) Muveran

(b) Pic sans nom

(c) Cervin

(d) Jungfrau

(e) Dent Blanche



(f) Barre des Écrins

(g) Dent du chat

(h) Aiguilles Dorées

Figure 5.9: The rendering results of different reconstructed mountain models produced by Unreal Engine 5.

working with point cloud data involves handling larger volumes of data, which translates to increased processing time, memory usage, and computational demands during the terrain model reconstruction process. While we can mitigate this by first reconstructing mesh data for each tile and then stitching the tiles together by merging nearby points along the tile boundaries, this approach is more complex compared to the DEM-based method, which benefits from the regularity of DEM-based mesh tiles that can be stitched together directly.

There are open-source software tools available for computer graphics and 3D data processing that can be used to estimate the normals of LiDAR point clouds. However, these tools may not fully utilize the LiDAR scanning information within the point cloud files. As a result, they may encounter challenges in accurately estimating the normal directions, leading to lower-quality reconstructed surface models. In contrast, our method generates more accurate normal vectors by combining the LiDAR scan information, resulting in improved 3D reconstruction models that better represent the true geometric features of the scanned environment.

5.4.4 Multi-resolution construction

In-core construction

The quantitative results of multi-resolution models are presented in Table 5.4, which includes the vertex count (n_v) and the triangle count (n_t) of the full-resolution model (M_L), coarsest LOD (M_0), and multi-resolution model. During the multi-resolution construction, we formed blocks using $4 \times 4 \times 4$ cubes and applied simplification with a target index number of one-fourth of the original count, 8 threads were used. For the Vanil Noir region, it corresponds to 2 models: The first model was generated using point cloud data, while the second model was generated using DEM data. The LiDAR point cloud based model had a lower number of triangles and vertices, resulting in faster construction times compared to the DEM-based model.

The remaining data (Locarno, Jungfrau, and Aldorf) was generated by out-of-core multi-resolution construction, which is the mesh model of LOD0. For all of the data, the construction time increased with the size of the models. Notably, the size of the multi-resolution model was approximately 4/3 times that of the original models, due to the choice of our block size and target simplification number, indicating the memory and time efficiency of our approach.

Table 5.4: Quantitative results of multi-resolution model construction

Model	M_L		M_0		Multi-resolution Model		level L	time (s)	RAM usage (MB)
	n_v	n_t	n_v	n_t	n_v	n_t			
Vanil Noir - LiDAR	6422780	12839353	1693	3120	9277947	17101916	6	3.554	574
Vanil Noir - DEM	20259001	40500000	1320	2497	29979065	54013561	7	8.043	1524
Locarno	16015674	31999292	1419	2662	24333117	43213172	7	7.149	1264
Jungfrau	27483159	110757478	1775	3348	39915384	73201146	7	12.853	2086
Aldorf	63201074	126274068	1434	2679	94128210	169079627	8	28.679	4616

Out-of-core construction

The out-of-core multi-resolution construction was tested on the tile set of Locarno, Jungfrau, and Aldorf regions. Table 5.5 shows the quantitative results of the out-of-core construction, where M_l is composed of tile units, each tile is stored independently on the disk, and M_0 represents the coarsest resolution. During the block-based mesh simplification, the choice of our block size and target simplification number is similar to the in-core construction. It can be seen from the table that the final number of indices is 4/3 of the input model. The out-of-core OOM representation is stored in the second memory, and the size of the multi-resolution model is less than two times that of the original 3D model file with adding the child-parent dependency of each vertex. Meanwhile, the parallel implementation improves the construction speed greatly. However, due to the large number of tiles and the large amount of total file size, it requires large computing power and memory space. For example, the out-of-core multi-resolution construction process of the $64km \times 64km$ terrain was carried out on the cluster computer.

Table 5.5: Quantitative results of out-of-core multi-resolution model construction

Model	M_L		M_0		Multi-resolution Model		size (GB)		time (s)	level L
	n_v	n_t	n_v	n_t	n_v	n_t	input	output		
Locarno	16064064	128000000	3157711	6283412	90723770	180566270	3.1	4.7	171.821	4
Jungfrau	257025024	512000000	4837599	9611158	431526060	859606674	12.3	18.3	596.765	5
Aldorf	1028100096	2048000000	15527366	30926558	1413030880	2812727138	49.3	73.3	2455.303	6

Comparison

We conducted a performance comparison between our multi-resolution construction method and Nexus on a desktop computer, the multi-resolution construction module nxsbuild of Nexus is used. The processing time of two methods are listed in Table 5.6. In both methods, the most time-consuming step in multi-resolution construction is mesh simplification. Our proposed method employs a QEM-based edge collapse approach with vertex sorting and query optimization algorithms. Nexus utilizes the simplification algorithm of the VCG library, which is also based on QEM-based mesh simplification.

Table 5.6: Multi-resolution model construction time comparison between Nexus and our method.

Model	Preprocessing time (s)	
	Our Method	Nexus
Vanil Noir - LiDAR	8.22	194.95
Vanil Noir - DEM	30.91	408.146
Locarno	18.468	342.491
Jungfrau	168.406	602.894
Aldorf	240.949	1298.323

5.4.5 Real-time rendering

In-core rendering

We test our in-core rendering algorithm on the models with different scales. First, the rendering performance of multi-resolution models and the full-resolution models are compared, the frame time is listed in Table 5.7.

Table 5.7: Frame time comparison of rendered multi-resolution models and original full-resolution models

Model	Frame time (ms)	
	Full-resolution	Multi-resolution
Vanil Noir - LiDAR	27.21-32.45	2.73-8.90
Vanil Noir - DEM	75.18-82.10	2.93-13.01
Locarno Region	87.19-93.05	3.23-11.03
Jungfrau Region	94.52-110.85	2.21-13.02
Aldorf Region	68.02-72.36	10.54-14.01

Table 5.7 shows that the real-time frame time of the multi-resolution model outperforms the full-resolution model. We rendered the multi-resolution model by controlling the movement of the viewpoint to the model from a distance, so the frame time has a certain range of volatility. Especially for larger terrain models, rendering the original model results in a loss of real-time and interactivity.

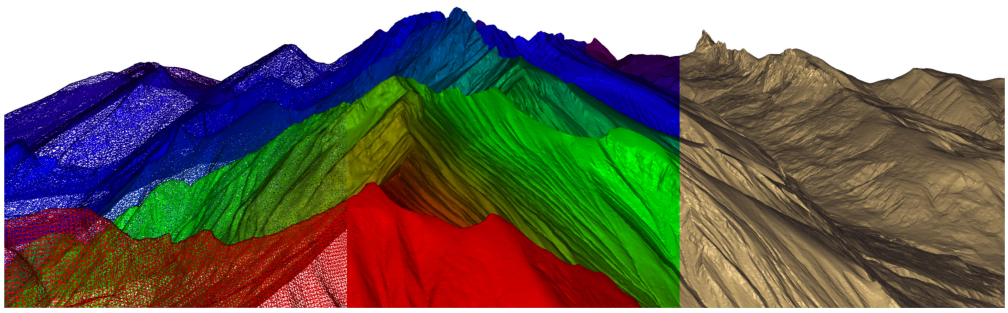
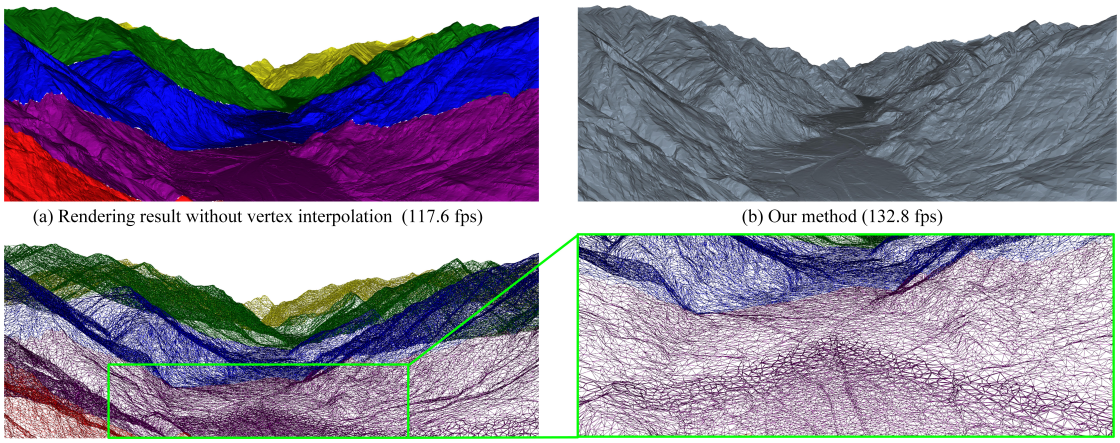


Figure 5.10: View-dependent real-time rendering results of the terrain model ($9km \times 9km$) of Vanil Noir region.

The rendering result of Vanil Noir is shown in Figure 5.10. The right part of Figure 5.10 is the rendering result under the Phong shading, the middle part is the rendering results of the multi-resolution model, and the different color represents the different LOD. The left part is the wireframe rendering result. The full-resolution model generated from the LiDAR point cloud contains 52 million triangles and is rendered at 160 fps on the desktop platform without a discrete GPU.



(a) Rendering result without vertex interpolation (117.6 fps)

(b) Our method (132.8 fps)

(c) Wireframe rendering result of our method

(d) Vertex interpolation

Figure 5.11: The rendering result of the Locarno region ($16km \times 16km$) terrain, where (a), (c), and (d) are rendering results with LOD coloration.

The rendering result of the Locarno region terrain model is shown in Figure 5.11. Our method can avoid cracks relative to rendering without vertex interpolation, it can even achieve higher FPS with vertex interpolation.

In our evaluation of the rendering performance of the 5 terrain models on the desktop platform, we utilized two indicators: frame time and the number of rendered triangles per frame. Figure 5.12 (a) illustrates a comparison of the number of rendered triangles per frame for different models under different rendering frameworks. The figure shows that as the viewpoint moves from far to the model, the number of triangles rendered per frame increases due to the higher granularity of the model. However, when the viewpoint moves close enough to the model, only part of the model needs to be rendered, so the number of rendered triangles drops.

Figure 5.12 (b) presents a comparison of the frame time of different models. Our method demonstrates the ability to generate interactive frame times for rendering different-scale models, even when the viewpoint is moving dynamically. It is important to note that the maximum and minimum frame times may vary across different models due to variations in complexity and scale. For each model, at the initial stage, when the viewpoint is far from the terrain model, the coarsest LOD level is rendered. This LOD level consists of fewer triangles, resulting in a higher frame rate. As the viewpoint approaches the object, the rendering system gradually increases the resolution of the LOD suitable for visualization.

In addition, we conducted an overdraw analysis on the model of Vanil Noir region. Given that our method utilizes vertex interpolation, which has the potential to exacerbate the overdraw issue, we specifically visualized the overdraw count in the region. As illustrated in Figure 5.13, in (b_1) and (c_1), different color represents different mesh partitions belonging to different cubes. In (b_2) and (c_2), the green part indicates that the pixel is not drawn, the blue part signifies that

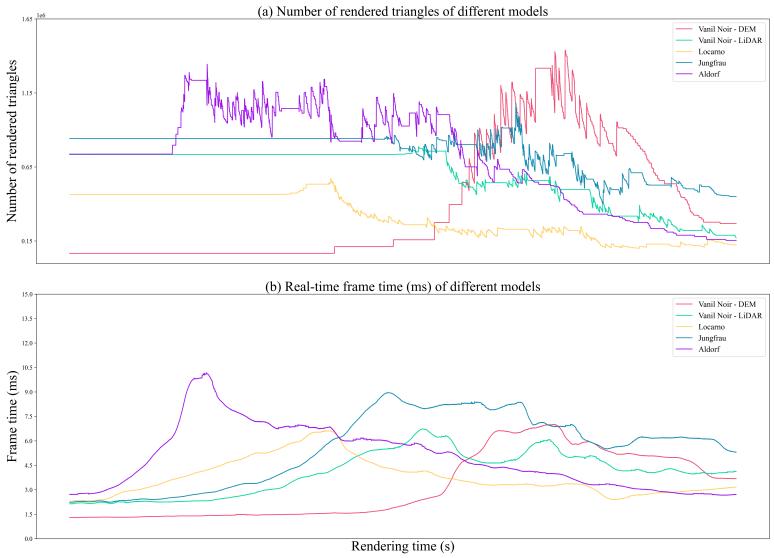


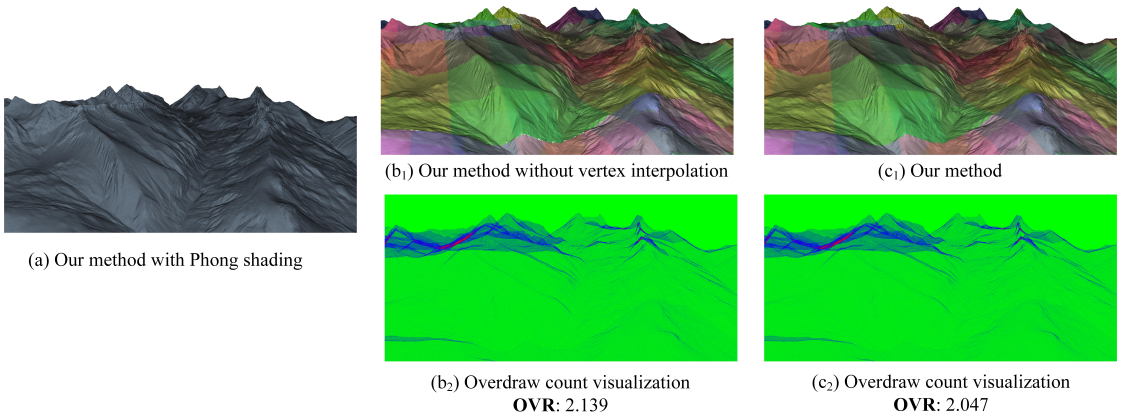
Figure 5.12: The number of rendered triangles comparison (a) and real-time rendering frame time comparison (b) of different models with dynamic viewpoint.

the number of overdraws is within the normal range, and the red part indicates that the number of overdraws is high. Additionally, we calculated the overdraw ratio (OVR), which quantifies the efficiency of the rendering process. The overdraw ratio is defined as the ratio between the total number of pixels that pass through the depth buffer and the number of visible pixels. The OVR of our method is 2.047, and the OVR of our method without vertex interpolation is 2.139. The results of our analysis demonstrate that our rendering method does not exacerbate the overdraw problem. In fact, it helps reduce overdraw compared to our method without vertex interpolation. This indicates that our approach efficiently utilizes the rendering resources and minimizes redundant computations, leading to improved performance.

Out-of-core rendering

For larger-scale terrain models that exceed a certain size, the out-of-core rendering scheme introduced in Chapter 4 can be applied. In this approach, the multi-resolution model data of a large-scale terrain model is stored on the disk, and the geometry data of the required cube is streamed when needed. Simultaneously, the parent cube needs to be loaded to ensure a smooth and artifact-free visualization result, avoiding popping and cracks. In practice, two threads are employed to handle data fetching and real-time rendering. Lower-resolution LODs are loaded into memory, giving priority to objects or areas that are distant or less visually significant. During real-time rendering, the Least Recently Used (LRU) strategy is employed to manage the streaming of the remaining high-resolution LOD data. The cubes within our multi-resolution model structure can be directly linked as LRU cache slots, facilitating seamless integration with

the data streaming scheme. For more details, please refer to section 4.3.



(a) Our method with Phong shading

(b_1) Our method without vertex interpolation

(c_1) Our method

(b_2) Overdraw count visualization
OVR: 2.139

(c_2) Overdraw count visualization
OVR: 2.047

Figure 5.13: The overdraw count visualization and OVR calculation results.

5.5 Conclusion

Our framework consists of three main components: terrain model reconstruction, multi-resolution model construction, and adaptive real-time rendering. Each component contributes to achieving high-quality and efficient rendering of large terrain models. In the terrain model reconstruction phase, we utilize 2 geospatial data sources: LiDAR point cloud and DEM, to create 3D models. By leveraging LiDAR point cloud data, we can capture geomorphic features like overhangs and cliffs, resulting in highly realistic terrain geometry. Additionally, we employ simple color shading techniques to generate visually appealing rendering results without relying on texture mapping in real-time. After obtaining the 3D models, the multi-resolution construction process focuses on creating a multi-resolution representation of the terrain model. Our proposed multi-resolution structure uses simple child-parent relationships, enabling real-time construction and minimizing memory consumption. In the adaptive real-time rendering phase, our algorithm is designed to be GPU-friendly, harnessing the power of modern graphics hardware. This enables us to render large terrain models in high fidelity while maintaining interactive frame rates.

Chapter 6

Conclusion and future work

6.1 Conclusion

In this thesis, a multi-resolution structure that incorporates a per-vertex child-parent dependency across different LODs is proposed. This structure enables efficient parallel processing, harnessing the advantages of modern CPU architectures. Notably, the construction implementation (starting from standard mesh file formats) achieves competitive levels in terms of 1 million input triangles per second per core on a standard configuration computer. Furthermore, the resulting multi-resolution model data size is less than two times that of the original model.

The real-time rendering scheme is built upon the multi-resolution model. Our approach successfully avoids LOD-popping and mesh cracks by utilizing vertex interpolation between child and parent. This ensures smooth transitions between different LODs, enhancing the visual quality of rendered models. The efficient implementation of our technique is supported by the capabilities of modern GPU architectures. To handle very large-scale models, we employ an out-of-core multi-resolution construction and rendering approach. By streaming data from external storage efficiently during runtime, the multi-resolution structure facilitates the seamless rendering of massive models that surpass the available memory capacity.

For the application to real-world terrain, we reconstruct the terrain model using LiDAR point cloud data and DEM. By leveraging the information from the LiDAR point cloud, our method creates mesh models that capture the landform features, including cliffs, overhangs, and other intricate terrain details. A comparative analysis with DEM-based models validates the superiority of our proposed method in generating more realistic terrain models. We then adapt the multi-resolution construction and rendering framework mentioned in Chapter 3 and Chapter 4 to large-scale terrain models, resulting in high-fidelity real-time rendering results.

Our rendering pipeline is designed to handle both regular-grid models and irregular mesh models, making it applicable across various domains including archaeology, art, and terrains. The implementation of this pipeline is a lightweight 3D viewer that facilitates multi-resolution con-

struction and interactive, view-dependent visualization of massive surface models across general graphics platforms, without large preprocessing times or booting up resource-hungry software.

6.2 Future work

In this thesis, our primary focus has been on 3D terrain reconstruction from airborne LiDAR point clouds, multi-resolution construction, and real-time rendering of massive 3D models. As we conclude this thesis, we identify several potential avenues for future work.

- The core concept of our multi-resolution construction and real-time rendering method has the potential to be seamlessly integrated into the pipeline of a rendering engine. The successful implementation of our 3D viewer serves as compelling proof of the viability and effectiveness of our approach.
- The out-of-core rendering pipeline discussed in Chapter 4 has the potential for further improvement. Data compression techniques can be employed to reduce texture size and streaming bandwidth. Additionally, implementing prefetching strategies to anticipate which data will be needed next and loading it in advance can help reduce latency. Lastly, the design of optimized file formats for storing assets, specifically tailored for efficient data retrieval and rendering, can be explored to enhance overall performance.
- In real-world large-scale terrain rendering, our method utilizes high-resolution geographic data. However, it's important to note that not every location in the world has access to such high-resolution geographic data. In the future, our approach could be further extended to allow users to integrate lower-resolution geographic data into an existing terrain hierarchy. Since our multi-resolution model mesh partition is based on regular space, it allows for more flexible integration of new data, such as DEM tile data. For the texture mapping of real-time rendering, we can incorporate satellite images as texture maps for rendering in flat terrain areas, such as plains or fields. Meanwhile, we can apply the algorithm discussed in Chapter 5 to render mountainous regions.

List of Figures

List of Tables

Bibliography

[AHH19] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/crc Press, 2019.

[Alf+18] Matthias Alfeld et al. "Joint data treatment for Vis–NIR reflectance imaging spectroscopy and XRF imaging acquired in the Theban Necropolis in Egypt by data fusion and t-SNE". In: *Comptes Rendus Physique* 19.7 (2018), pp. 625–635.

[Álv+07] Rafael Álvarez et al. "A mesh optimization algorithm based on neural networks". In: *Information Sciences* 177.23 (2007), pp. 5347–5364.

[App] Austin Appleby. *SMHasher: Test Hash Functions*. `https://github.com/aappleby/smhasher`.

[Bau72] Bruce G Baumgart. *Winged edge polyhedron representation*. Tech. rep. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1972.

[Bax+02] William V Baxter III et al. "GigaWalk: Interactive Walkthrough of Complex Environments." In: *Rendering Techniques* 203 (2002).

[Ber+99] Fausto Bernardini et al. "The ball-pivoting algorithm for surface reconstruction". In: *IEEE transactions on visualization and computer graphics* 5.4 (1999), pp. 349–359.

[BF01] Bedrich Benes and Rafael Forsbach. "Layered data representation for visual simulation of terrain erosion". In: *Proceedings Spring Conference on Computer Graphics*. IEEE. 2001, pp. 80–86.

[BF05] H Borouchaki and PJ Frey. "Simplification of surface mesh using Hausdorff envelope". In: *Computer methods in applied mechanics and engineering* 194.48-49 (2005), pp. 4864–4884.

[Bor+05] Louis Borgeat et al. "GoLD: interactive display of huge colored and textured models". In: *ACM Transactions on Graphics (TOG)* 24.3 (2005), pp. 869–877.

[Bot+10] Mario Botsch et al. *Polygon mesh processing*. CRC press, 2010.

[BW10] ZM Bi and Lihui Wang. "Advances in 3D data acquisition and processing for industrial applications". In: *Robotics and Computer-Integrated Manufacturing* 26.5 (2010), pp. 403–413.

[Cal02] Dean Calver. "Vertex decompression in a shader". In: *ShaderX: Vertex and Pixel Shader Tips and Tricks* (2002), pp. 172–187.

[Cha+01] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[Cig+03a] Paolo Cignoni et al. "BDAM—Batched Dynamic Adaptive Meshes for high performance terrain visualization". In: *Computer Graphics Forum*. Vol. 22. 3. Wiley Online Library. 2003, pp. 505–514.

[Cig+03b] Paolo Cignoni et al. “External memory management and simplification of huge meshes”. In: *IEEE Transactions on Visualization and Computer Graphics* 9.4 (2003), pp. 525–537.

[Cig+03c] Paolo Cignoni et al. “Planet-sized batched dynamic adaptive meshes (P-BDAM)”. In: *IEEE Visualization, 2003. VIS 2003.* IEEE. 2003, pp. 147–154.

[Cig+04] P. Cignoni et al. “Adaptive TetraPuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models”. In: *ACM Transactions on Graphics* 23.3 (2004), p. 796–803.

[Cig+05] Paolo Cignoni et al. “Batched multi triangulation”. In: *VIS 05. IEEE Visualization, 2005.* IEEE. 2005, pp. 207–214.

[CKS98] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. “Directed edges—a scalable representation for triangle meshes”. In: *Journal of Graphics tools* 3.4 (1998), pp. 1–11.

[CL96] Daniel Cohen-Or and Yishay Levanoni. *Temporal continuity of levels of detail in delaunay triangulated terrain.* IEEE, 1996.

[Cla76] James H Clark. “Hierarchical geometric models for visible surface algorithms”. In: *Communications of the ACM* 19.10 (1976), pp. 547–554.

[Clo23] CloudCompare. *3D point cloud and mesh processing software Open Source Project.* `https://www.cloudcompare.org/main.html`. 2023.

[Coh+96] Jonathan Cohen et al. “Simplification envelopes”. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques.* 1996, pp. 119–128.

[Cor] Microsoft Corporation. *Texture Filtering with Mipmaps.* `https://learn.microsoft.com/en-us/previous-versions/aa921432(v=msdn.10)`.

[CR11] Patrick Cozzi and Kevin Ring. *3D engine design for virtual globes.* AK Peters/CRC Press, 2011.

[CRS98] Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. “Metro: measuring error on simplified surfaces”. In: *Computer graphics forum.* Vol. 17. 2. Wiley Online Library. 1998, pp. 167–174.

[DG12] Evgenij Derzapf and Michael Guthe. “Dependency-free parallel progressive meshes”. In: *Computer Graphics Forum.* Vol. 31. 8. Wiley Online Library. 2012, pp. 2288–2302.

[DMG10] Evgenij Derzapf, Nicolas Menzel, and Michael Guthe. “Parallel view-dependent out-of-core progressive meshes”. In: *Realismus der Echtzeitgrafik* 61 (2010).

[DMP98] Leila De Floriani, Paola Magillo, and Enrico Puppo. “Efficient implementation of multi-triangulations”. In: *Proceedings Visualization’98 (Cat. No. 98CB36276).* IEEE. 1998, pp. 43–50.

[DNJ20] Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. “Overfit neural networks as a compact shape representation”. In: (2020).

[DP10] Jonathon Doran and Ian Parberry. “Controlled procedural terrain generation using software agents”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.2 (2010), pp. 111–119.

[Dup20] Jonathan Dupuy. “Concurrent Binary Trees (with application to longest edge bisection)”. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (2020), pp. 1–20.

[DVP22] Dilip Kumar Dalei, N Venkataramanan, and Narayan Panigrahi. "A Review of LOD based Techniques for Real-time Terrain Rendering". In: *2022 IEEE 6th Conference on Information and Communication Technology (CICT)*. IEEE. 2022, pp. 1–6.

[ECT17] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. "A survey of different approaches for overcoming the processor-memory bottleneck". In: *International Journal of Computer Science and Information Technology* 9.2 (2017), pp. 151–163.

[EM00] Carl Erikson and Dinesh Manocha. "Hierarchical levels of detail for fast display of large static and dynamic environments". In: *UNC Chapel Hill Computer Science Technical Report TR00* 12 (2000).

[EMB01] Carl Erikson, Dinesh Manocha, and William V Baxter III. "HLODs for faster display of large static and dynamic environments". In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001, pp. 111–120.

[Eng23] OGRE - Open Source 3D Graphics Engine. *OGRE*. 2023. URL: `https://www.ogre3d.org/`.

[FKP05] Leila De Floriani, Leif Kobbelt, and Enrico Puppo. "A survey on data structures for level-of-detail models". In: *Advances in multiresolution for geometric modelling*. Springer, 2005, pp. 49–74.

[FPM97] Leila De Floriani, Enrico Puppo, and Paola Magillo. "A formal approach to multiresolution hypersurface modeling". In: *Geometric Modeling: Theory and Practice: The State of the Art*. Springer, 1997, pp. 302–323.

[FS93] Thomas A Funkhouser and Carlo H Séquin. "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993, pp. 247–254.

[Fun03] Thomas Funkhouser. "Overview of 3d object representations". In: *Princeton University, C0S D* 597 (2003).

[Gal+19] Eric Galin et al. "A review of digital terrain modeling". In: *Computer Graphics Forum*. Vol. 38. 2. Wiley Online Library. 2019, pp. 553–577.

[Gar99] Michael Garland. *Quadric-based polygonal surface simplification*. Carnegie Mellon University, 1999.

[GH97] Michael Garland and Paul S Heckbert. "Surface simplification using quadric error metrics". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, pp. 209–216.

[GH98] M. Garland and P.S. Heckbert. "Simplifying surfaces with color and texture using quadric error metrics". In: *Proceedings Visualization '98 (Cat. No.98CB36276)*. 1998, pp. 263–269. DOI: `10.1109/VISUAL.1998.745312`.

[GK04] Michael Guthe and Reinhard Klein. "Streaming HLODs: an out-of-core viewer for network visualization of huge polygon models". In: *Computers & Graphics* 28.1 (2004), pp. 43–50.

[GKY08] Enrico Gobbetti, Dave Kasik, and Sung-eui Yoon. "Technical strategies for massive model visualization". In: *Proceedings of the 2008 ACM symposium on Solid and physical modeling*. 2008, pp. 405–415.

[HD04] Tan Kim Heok and Daut Daman. "A review on level of detail". In: *Proceedings. International Conference on Computer Graphics, Imaging and Visualization, 2004. CGIV 2004.* IEEE. 2004, pp. 70–75.

[Hop+93] Hugues Hoppe et al. “Mesh optimization”. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993, pp. 19–26.

[Hop96] Hugues Hoppe. “Progressive meshes”. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, pp. 99–108.

[Hop97] Hugues Hoppe. “View-dependent refinement of progressive meshes”. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, pp. 189–198.

[Hop98a] Hugues Hoppe. “Efficient implementation of progressive meshes”. In: *Computers & Graphics* 22.1 (1998), pp. 27–36.

[Hop98b] Hugues Hoppe. “Smooth view-dependent level-of-detail control and its application to terrain rendering”. In: *Proceedings Visualization'98 (Cat. No. 98CB36276)*. IEEE. 1998, pp. 35–42.

[HSH09] Liang Hu, Pedro V Sander, and Hugues Hoppe. “Parallel view-dependent level-of-detail control”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.5 (2009), pp. 718–728.

[INP04] INP/Inria/UBordeaux/UPMC. *mmg - Surface and volume remeshers*. 2004. URL: `https://github.com/MmgTools/mmg`.

[Ioa+17] Anastasia Ioannidou et al. “Deep Learning Advances in Computer Vision with 3D Data: A Survey”. In: *ACM Comput. Surv.* 50.2 (Apr. 2017). ISSN: 0360-0300. DOI: `10.1145/3042064`. URL: `https://doi.org/10.1145/3042064`.

[Kap20] Arseny Kapoulkine. *Learning from data*. 2020. URL: `https://zeux.io/2020/01/22/learning-from-data/`.

[Kap22] Arseny Kapoulkine. *meshoptimizer*. `https://github.com/zeux/meshoptimizer`. Version 0.19. GitHub, 2022.

[Kat+20] Hiroharu Kato et al. “Differentiable rendering: A survey”. In: *arXiv preprint* (2020).

[KBH06] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. “Poisson surface reconstruction”. In: *Proceedings of the fourth Eurographics symposium on Geometry processing*. Vol. 7. 2006, p. 0.

[KCS98] Leif Kobbelt, Swen Campagna, and Hans-Peter Seidel. “A general framework for mesh decimation”. In: *Graphics interface*. Vol. 98. 1998, pp. 43–50.

[KH13] Michael Kazhdan and Hugues Hoppe. “Screened poisson surface reconstruction”. In: *ACM Transactions on Graphics (ToG)* 32.3 (2013), pp. 1–13.

[KSW21] BRIAN KARIS, RUNE STUBBE, and GRAHAM WIHLIDAL. “A Deep Dive into Nanite Virtualized Geometry”. In: *ACM SIGGRAPH*. 2021.

[Lad19] Alen Ladavac. *Four Million Acres, Seriously: GPU-Based Procedural Terrains in 'Serious Sam 4: Planet Badass'. In: (Game Developers Conference)*. 2019. URL: `https://www.gdcvault.com/play/1026349/Advanced-Graphics-Techniques-Tutorial-Four`.

[LC98] William E Lorensen and Harvey E Cline. “Marching cubes: A high resolution 3D surface construction algorithm”. In: *Seminal graphics: pioneering efforts that shaped the field*. 1998, pp. 347–353.

[LCL10] Jongseok Lee, Sungyul Choe, and Seungyong Lee. “Mesh geometry compression for mobile graphics”. In: *2010 7th IEEE Consumer Communications and Networking Conference*. IEEE. 2010, pp. 1–5.

[Lev+00] Marc Levoy et al. “The digital Michelangelo project: 3D scanning of large statues”. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 131–144.

[LH04] Frank Losasso and Hugues Hoppe. “Geometry clipmaps: terrain rendering using nested regular grids”. In: *ACM Siggraph 2004 Papers*. 2004, pp. 769–776.

[LHZ20] Yaqian Liang, Fazhi He, and Xiantao Zeng. “3D mesh simplification with feature preservation based on whale optimization algorithm and differential evolution”. In: *Integrated Computer-Aided Engineering* 27.4 (2020), pp. 417–435.

[Li+21] Shi Li et al. “Multi-resolution terrain rendering using summed-area tables”. In: *Computers & Graphics* 95 (2021), pp. 130–140.

[Lim+13] Max Limper et al. “The pop buffer: Rapid progressive clustering by geometry quantization”. In: *Computer Graphics Forum*. Vol. 32. 7. Wiley Online Library. 2013, pp. 197–206.

[Lin00] Peter Lindstrom. “Out-of-core simplification of large polygonal models”. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 259–262.

[Lin03] Peter Lindstrom. “Out-of-core construction and visualization of multiresolution surfaces”. In: *Proceedings of the 2003 symposium on Interactive 3D graphics*. 2003, pp. 93–102.

[Liu+19] Shichen Liu et al. “Soft rasterizer: A differentiable renderer for image-based 3d reasoning”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 7708–7717.

[LKE09] Yotam Livny, Zvi Kogan, and Jihad El-Sana. “Seamless patches for GPU-based terrain rendering”. In: *The Visual Computer* 25 (2009), pp. 197–208.

[LP02] Peter Lindstrom and Valerio Pascucci. “Terrain simplification simplified: A general framework for view-dependent out-of-core visualization”. In: *IEEE Transactions on Visualization and Computer graphics* 8.3 (2002), pp. 239–254.

[LS01] Peter Lindstrom and Claudio T Silva. “A memory insensitive technique for large model simplification”. In: *Proceedings Visualization, 2001. VIS'01.* IEEE. 2001, pp. 121–550.

[LT97] Kok-Lim Low and Tiow-Seng Tan. “Model simplification using vertex-clustering”. In: *Proceedings of the 1997 symposium on Interactive 3D graphics*. 1997, 75–ff.

[Mag+15] Adrien Maglo et al. “3d mesh compression: Survey, comparisons, and emerging trends”. In: *ACM Computing Surveys (CSUR)* 47.3 (2015), pp. 1–41.

[Mes+19] Lars Mescheder et al. “Occupancy networks: Learning 3d reconstruction in function space”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 4460–4470.

[Mic23] MichaelKazhdan. *Adaptive Multigrid Solvers*. `https://github.com/mkazhdan/PoissonRecon`. 2023.

[Moo18] Jeremy Moore. *Terrain Rendering in 'Far Cry 5'. In: (Game Developers Conference)*. 2018. URL: `https://www.gdcvault.com/play/1025480/%20Terrain-Rendering-in-Far-Cry`.

[MS13] Urs Marti and Dieter Schneider. “The new Swiss National Height System LHN95”. In: *Advances in Positioning and Reference Frames: IAG Scientific Assembly Rio de Janeiro, Brazil, September 3–9, 1997*. Springer, 2013, pp. 143–148.

[Nat+13] Mattia Natali et al. “Modeling Terrains and Subsurface Geology.” In: *Eurographics (State of the Art Reports)*. 2013, pp. 155–173.

[OKK15] Hiromu Ozaki, Fumihito Kyota, and Takashi Kanai. “Out-of-Core Framework for QEM-based Mesh Simplification.” In: *EGPGV@ EuroVis*. 2015, pp. 87–96.

[Osw15] Martin Ralf Oswald. “Convex Variational Methods for Single-View and Space-Time Multi-View Reconstruction”. PhD thesis. Technische Universität München, 2015.

[Paj98] Renato Pajarola. “Large scale terrain visualization using the restricted quadtree triangulation”. In: *Proceedings Visualization'98 (Cat. No. 98CB36276)*. IEEE. 1998, pp. 19–26.

[Par+19] Jeong Joon Park et al. “Deepsdf: Learning continuous signed distance functions for shape representation”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 165–174.

[Pey+09] Adrien Peytavie et al. “Arches: a framework for modeling complex terrains”. In: *Computer graphics forum*. Vol. 28. 2. Wiley Online Library. 2009, pp. 457–467.

[PG07] Renato Pajarola and Enrico Gobbetti. “Survey of semi-regular multiresolution models for interactive terrain rendering”. In: *The Visual Computer* 23.8 (2007), pp. 583–605.

[Pon09a] Federico Ponchio. “Multiresolution structures for interactive visualization of very large 3D datasets”. en. PhD thesis. [Clausthal-Zellerfeld], 2009. ISBN: 978-394-039-478-1. URL: http://uri.gbv.de/document/gvk:ppn:610223534.

[Pon09b] Federico Ponchio. “Multiresolution structures for interactive visualization of very large 3D datasets”. PhD thesis. Zugl.: Clausthal, Techn. Univ., Diss., 2008, 2009.

[PPZ22] Rolandos Alexandros Potamias, Stylianos Ploumpis, and Stefanos Zafeiriou. “Neural mesh simplification”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 18583–18592.

[Pup+96] Enrico Puppo et al. “Variable Resolution Terrain Surfaces.” In: *CCCG*. 1996, pp. 202–210.

[Pup98] Enrico Puppo. “Variable resolution triangulations”. In: *Computational Geometry* 11.3-4 (1998), pp. 219–238.

[RB93] Jarek Rossignac and Paul Borrel. “Multi-resolution 3D approximations for rendering complex scenes”. In: *Modeling in computer graphics*. Springer, 1993, pp. 455–465.

[RL01] Szymon Rusinkiewicz and Marc Levoy. “Streaming QSplat: A viewer for networked visualization of large, dense models”. In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001, pp. 63–68.

[Rod14] Musée Rodin. *The Thinker at the Musée Rodin, France*. 2014. URL: https://www.myminifactory.com/object/3d-print-the-thinker-at-the-muse-rodin-france-2127.

[RR96] Rémi Ronfard and Jarek Rossignac. “Full-range approximation of triangulated polyhedra.” In: *Computer Graphics Forum*. Vol. 15. 3. Wiley Online Library. 1996, pp. 67–76.

[Sar+22] Jonathan Sarton et al. “Interactive Visualization and On-Demand Processing of Large Volume Data: A Fully GPU-Based Out-Of-Core Approach”. In: (2022).

[SBD13] Manuel Scholz, Jan Bender, and Carsten Dachsbacher. “Level of Detail for Real-Time Volumetric Terrain Rendering.” In: *VMV*. 2013, pp. 211–218.

[SG01] Eric Shaffer and Michael Garland. "Efficient adaptive simplification of massive meshes". In: *Proceedings Visualization, 2001. VIS'01.* IEEE. 2001, pp. 127–551.

[SM05] Pedro V Sander and Jason L Mitchell. "Progressive buffers: view-dependent geometry and texture LOD rendering". In: *Proceedings of the third Eurographics symposium on Geometry processing.* 2005, 129–es.

[SS05] Szymon Stachniak and Wolfgang Stuerzlinger. "An algorithm for automated fractal terrain deformation". In: *Computer Graphics and Artificial Intelligence* 1 (2005), pp. 64–76.

[Šta+08] Ondřej Šťava et al. "Interactive terrain modeling using hydraulic erosion". In: *Proceedings of the 2008 acm siggraph/eurographics symposium on computer animation.* 2008, pp. 201–210.

[Str09] Filip Strugar. "Continuous distance-dependent level of detail for rendering heightmaps". In: *Journal of graphics, GPU, and game tools* 14.4 (2009), pp. 57–74.

[SZL92] William J Schroeder, Jonathan A Zarge, and William E Lorensen. "Decimation of triangle meshes". In: *Proceedings of the 19th annual conference on Computer graphics and interactive techniques.* 1992, pp. 65–70.

[Tak+21] Towaki Takikawa et al. "Neural geometric level of detail: Real-time rendering with implicit 3D shapes". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2021, pp. 11358–11367.

[Tes+03] Matthias Teschner et al. "Optimized spatial hashing for collision detection of deformable objects." In: *Vmv.* Vol. 3. 2003, pp. 47–54.

[Tew+20] Ayush Tewari et al. "State of the art on neural rendering". In: *Computer Graphics Forum.* Vol. 39. 2. Wiley Online Library. 2020, pp. 701–727.

[Top23a] Federal Office of Topography swisstopo. *swissALTI3D.* 2023. URL: `https://www.swisstopo.admin.ch/en/geodata/height/alti3d.html`.

[Top23b] Federal Office of Topography swisstopo. *swissSURFACE3.* 2023. URL: `https://www.swisstopo.admin.ch/en/geodata/height/surface3d.html`.

[Ulr02] Thatcher Ulrich. "Rendering massive terrains using chunked level of detail control". In: *Proc. ACM SIGGRAPH 2002.* 2002.

[VC04] Sébastien Valette and Jean-Marc Chassery. "Approximated centroidal voronoi diagrams for uniform polygonal mesh coarsening". In: *Computer Graphics Forum.* Vol. 23. 3. Wiley Online Library. 2004, pp. 381–389.

[VCP19] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. "Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?" In: *Proceedings of the 2019 International Conference on Management of Data.* New York, NY, USA: Association for Computing Machinery, 2019, pp. 36–53. DOI: `10.1145/3299869.3300075`.

[Vis17] Visual Computing Lab. *Visualization and Computer Graphics Library.* `https://github.com/cnr-isti-vclab/vcglib`. 2017.

[Vis20] CNR Visual Computing Laboratory ISTI. *Nexus.* 2020. URL: `https://github.com/cnr-isti-vclab/nexus`.

[VKC05] Sébastien Valette, Ioannis Kompatsiaris, and Jean-Marc Chassery. "Adaptive polygonal mesh simplification with discrete centroidal voronoi diagrams". In: *ICMI.* 2005.

[VM02] Gokul Varadhan and Dinesh Manocha. "Out-of-core rendering of massive geometric environments". In: *IEEE Visualization, 2002. VIS 2002.* IEEE. 2002, pp. 69–76.

[WDS05] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. "An interactive out-of-core rendering framework for visualizing massively complex models". In: *ACM SIGGRAPH 2005 Courses.* 2005, 17–es.

[WK03] Jianhua Wu and Leif Kobbelt. "A stream algorithm for the decimation of massive meshes." In: *Graphics interface.* Vol. 3. 2003, pp. 185–192.

[YKC23] Ke Yu, Minguk Kim, and Jun Rim Choi. "Memory-Tree Based Design of Optical Character Recognition in FPGA". In: *Electronics* 12.3 (2023), p. 754.

[Yoo+05] S-E Yoon et al. "Quick-VDR: Out-of-core view-dependent rendering of gigantic models". In: *IEEE Transactions on Visualization and Computer Graphics* 11.4 (2005), pp. 369–382.

[ZOR00] D ZORIN. "Subdivision for modeling and animation". In: *SIGGRAPH 2000 Course Notes ACM SIGGRAPH* (2000).

Multi-scale simplification and visualization of large 3D models

Abstract

This thesis focuses on the creation and interactive visualization of surface meshes derived from massive data, such as those obtained from the use of laser remote sensing scanners or multiple-view image captures. Recent advancements in the development, miniaturization, and widespread accessibility of these capture devices aim to provide a detailed representation of reality.

The raw data resulting from these captures, whether in geographic information, archaeology, or medicine, often pose challenges due to their voluminous and disorganized nature. To address this challenge, the first step in a processing pipeline typically involves the creation of a high-resolution mesh. The chosen solution must effectively mitigate measurement noise and accurately reproduce the topology of the target object.

As capture devices have advanced in resolution, visualization hardware has improved in memory capacity and processing speed. However, it has become impractical to interactively visualize high-resolution meshes, containing tens of millions or even billions of elements. Handling such data flow by hardware and transferring it between pipeline components present significant challenges.

The subsequent step involves constructing a hierarchy of simplified meshes derived from the high-resolution mesh, with each simplification introducing controlled detail loss. While it might appear that processing capacity comes at the expense of detail, for visualization purposes, only the apparent level of detail matters. Portions distant from the point of observation require a lower absolute level of detail than those nearby. This enables the subdivision of different mesh levels into subparts, dynamically selecting and recombining them to adapt to changes in viewpoint, ensuring an optimal apparent level of detail.

In the first part of the thesis, we introduce an approach aimed at addressing the two main challenges inherent in the simultaneous management of multiple pieces of meshes with different resolutions: mesh junction issues (known as "mesh cracks") and temporal discontinuity problems resulting from resolution changes (referred to as "LOD popping"). This approach, situated at the intersection of techniques based on mesh morphing and those rooted in multi-triangulations, continues a long lineage of research in these areas. We accompany it with both a reference implementation and a lightweight visualization prototype serving as a proof of concept. The preprocessing phase, responsible for constructing the hierarchy from the high-resolution mesh, stands out with a processing capacity on the order of one million triangles per second per processing core. This represents a substantial order of magnitude improvement compared to the existing solutions we have had the opportunity to evaluate.

In the second part of the thesis, we dig into the construction of high-resolution meshes from a point cloud, specifically in the case of airborne LiDAR. We demonstrate that the reliability of conventional methods for normal estimation can be significantly enhanced by taking into account data specific to such surveys, including point timestamps and laser beam angles. This improvement has a crucial impact on the quality of subsequent surface mesh reconstruction using the Poisson method. The Poisson method derives the mesh as the level surface of a scalar function, itself obtained by solving a Poisson problem in which the oriented point cloud contributes to defining the source term. We apply these methods to recent open LiDAR data from Swiss (SwissSurface3D) and French (Lidar HD) LiDAR programs, demonstrating that they enable a more faithful reproduction compared to classical methods based on regular altitude grids (digital elevation models), particularly in mountainous areas.

Keywords: mesh simplification, multi-resolution meshes, 3d reconstruction and visualization, real-time rendering

---

Laboratoire Jacques-Louis Lions
Sorbonne Université – Campus Pierre et Marie Curie – 4 place Jussieu – 75005 Paris – France

Résumé

Cette thèse s'intéresse à la création et à la visualisation interactive de maillages surfaciques issus de données massives, telles que celles obtenues suite à l'emploi de scanners à télédétection laser ou de prises de vues multiples. Ces dernieres années ont en effet vu l'accélération du développement, de la miniaturisation et de la démocratisation de ces dispositifs de capture visant une représentation fine du réel.

Les données brutes issues des captures, par exemple en géographie, archéologie ou en médecine sont difficiles à exploiter car volumineuses et désordonnées. Pour faire face à ce défi, la première étape d'une chaîne de traitement consiste classiquement en la création d'un maillage très haute résolution. La solution retenue doit palier au mieux les bruits de mesure et reproduire la topologie de l'objet étudié.

L'augmentation de la résolution des dispositifs de capture, des capacités de mémoire et de traitement du matériel de visualisation rend souvent impraticable la visualisation interactive des maillages haute résolution. Ces maillages de dizaines de millions voire plusieurs milliards d'éléments sont un défi pour le matériel et pour le transfert de données entre les composants de la chaîne.

L'étape suivante implique la création d'une hiérarchie de maillages simplifiés dérivant de l'original, les pertes de détail à chaque simplification étant contrôlées. Bien que cette approche semble contourner le probleme au lieu de le résoudre, seul le niveau de détail importe en visualisation. Ainsi, il devient possible de découper les niveaux de maillage en sous-parties, les sélectionner et les recombiner à la volée pour suivre les mouvements du point d'observation en optimisant partout le niveau de détail apparent.

Dans la première partie de la thèse, nous proposons une approche qui s'attache à résoudre les deux difficultés principales inhérentes à la gestion simultanée de plusieurs morceaux de maillages de résolutions différentes : les défauts de jointures ("mesh cracks"), et les problèmes de discontinuité temporelle suite aux changements de résolutions ("LOD popping"). Cette approche, dans la longue lignée de travaux ayant abordé ces questions, s'inscrit au carrefour entre les techniques basées sur le morphing entre maillages et celles basées sur les multi-triangulations. Nous l'accompagnons d'une implémentation de référence, ainsi que d'un prototype de visualiseur léger servant de preuve de concept. La partie pré-traitement, qui construit la hiérarchie à partir du maillage haute résolution, se distingue par une capacité de l'ordre du million de triangles par seconde et par coeur de calcul, ce qui représente un ordre de grandeur de gain par rapports aux solutions existantes que nous avons pu tester.

Dans la seconde partie de la thèse, nous abordons la question de la construction du maillage haute résolution à partir d'un nuage de points dans le cas spécifique du Lidar aéroporté. Nous montrons que la fiabilité des méthodes classiques d'estimation de normales peut être nettement améliorée en tenant compte de données propres à de tels relevés : horodatage des points et angle du faisceau laser notamment. Cette amélioration a une incidence cruciale sur la qualité de la reconstruction ultérieure d'un maillage surfacique par la méthode dite de Poisson. Cette dernière obtient le maillage comme surface de niveau d'une fonction scalaire, elle-même obtenue par la résolution d'un problème de Poisson pour lequel le nuage de points orientés intervient dans la définition du terme source. Nous appliquons ces méthodes aux données libres récentes des programmes Lidar suisse (SwissSurface3D) et français (Lidar HD), et montrons qu'elles permettent une reproduction plus fidèle que les méthodes classiques basées sur les grilles à pas régulier (modèle numériques de terrain), d'autant plus dans les zones montagneuses.

Mots clés : simplification de maillage, maillages multi-résolution, reconstruction et visualisation 3d, rendu temps réel