LARGE SCALE VISUALIZATION OF 3D URBAN RECONSTRUCTION

by

YANG LIU

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Master of Philosophy
in Computer Science and Engineering

August 2014, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

YANG LIU

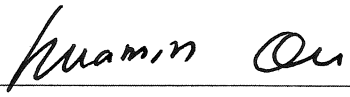
LARGE SCALE VISUALIZATION OF 3D URBAN RECONSTRUCTION

by

YANG LIU

This is to certify that I have examined the above M.Phil. thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

PROF. LONG QUAN, THESIS SUPERVISOR



PROF. HUAMIN QU, THESIS SUPERVISOR

PROF. SIU-WING CHENG, ACTING HEAD OF DEPARTMENT

Department of Computer Science and Engineering
20 August 2014

ACKNOWLEDGMENTS

I would first like to express my deep and sincere gratitude for my advisors, Professor Long Quan and Professor Humin Qu. Prof. Quan has always been a strong and supportive advisor to me throughout the period that I worked with him. His hardworking attitude and philosophy about life have had a great impact on my way of thinking. I would also like to thank Prof. Qu, first for his encouragement that I pursue Computer Science study from an unrelated background and second for his numerous advices and sharings which greatly influenced my understanding of the life and the world. I would also like to thank Professors Chew-lan Tai and Pedro Sander, who served on my thesis committee, for their patience in reading this thesis and their helpful comments.

Special thanks are due to my fellow students at HKUST Vision and Graphics lab. In particular, I want to thank Tian Fang and Zhexi Wang for their valuable suggestions and insights on my work. In addition, I want to thank Jacky Tang and Zuozhuo Dai for the thoughtful discussion on the project contained in the thesis. I am also thankful to my fellow students in the research group, with whom I spent a great time: Jinglu Wang, Siyu Zhu, Shengnan Cai, Runze Zhang, Conglei Shi, Panpan Xu, Lu Lu, Guodao Sun and Wenbin Wu.

Finally, I would like to express my heart-felt thanks to my family and friends for their love, encouragement and support. This thesis would not have been possible without them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LARGE SCALE VISUALIZATION OF 3D URBAN RECONSTRUCTION

by

YANG LIU

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

Recent advances in geospatial data acquisition technologies are enabling automatic urban reconstruction in ever increasing scale. Though a number of mature systems are developed for interative exploration of the final reconstructed urban environments, few try to visualize the intermediate products of the entire reconstruction pipeline. We present a system to interactive navigate and explore large scale reconstruction data sets consisting of input images, 3D point clouds and polygonal meshes in an integrated Internet-based platform. The massive scale of data is handled by hierachical level of details methods, which generates simplified data through mesh simplification and point resampling as well as streams and visualizes data progressively in a fine-grained manner. We employ cache, culling and compression algorithms specifically tailored to our hierachical data structure to further speed up rendering. Our system is built directly into standard browser, which can be accessed independent of platform including mobile devices with Internet access. Our system has hosted a dozen of large scale urban reconstruction datasets and we will demonstrate several representative datasets in this thesis.

CHAPTER 1

INTRODUCTION

In the recent years, advances in geospatial data acquisition techniques have led to an explosion in available 3D geoinformation data. These massive amount of data form the basis of visualizing geoinformation environments in 3D. On the one hand, imagery and elevation data acquired from satellite or high-performance airborne camera systems allow representation of 3D geodata globally, which are known as the virtual globes. On the other hand, data captured using imaging or LiDAR sensors of airborne mapping systems provide detailed information for local environments, allowing the visualization of photo-realistic 3D city. Despite the acquisition method, these geospatial data have an underlying characteristic in common, which is that the amount of data is massive, making interactive visualization of such data in large scale a challenging task.

Emerging Internet technologies, in particular WebGL, are enabling rich and interactive visualization of large scale urban virtual environments over the Internet. WebGL is a cross-platform web standard for a graphics API based on OpenGL ES 2.0 and is native to the majority of modern browsers. Before it, most Internet-based delivery of interactive 3D contents required installation of third-party technologies, such as local platform-specific software programs or browser plugins. With the introduction of WebGL standard, 3D urban scenes can be seamlessly integrated to standard web browsers, accessible from different platforms including mobile devices with compatible browser installed.

Given that the state of art web technologies provide a universal mechanism for interactive exploration of 3D scenes over the web, the last few years have witnessed a continuous growth in web-based 3D geovisualization applications. In particular, several projects were initiated to deliver interactive 3D urban scenes over the web, including the WebGL version of Google Earth (Google Maps, 2014), Nokia Maps 3D (Nokia Maps, 2014) and Bing Maps 3D (Bing Maps, 2014).

Despite the extensive efforts on consumer level 3D urban visualization, few attempts are made to integrate the presentation of urban reconstruction process along with its final products. Allowing efficient visualization of different intermediate stages in the urban reconstruction pipeline is valuable in multiple aspects. It brings the benefits of controlling in a fully automatic reconstruction process. Additionally, intuitive and convenient access to the intermediate products of different reconstruction stages help to spread the awareness and popularity of such methods in the general audience.

1.1 Our System

We built a visualization engine to integrate efficient presentation of different stages in our reconstruction pipeline, from the collection of massive high-resolution input images, to the generated semi-dense point clouds, and finally to the textured meshes of the reconstructed urban scenes. Our system is developed to address the following goals:

- Integration of multiple data types: the visualization of images, point clouds and models are integrated in one platform for efficient access.
- Handling massive scale of data: the system is designed to handle massive scale of data, with different data management and level of details techniques for each data type.
- 3D visualization on the web without plugin: our system deliver the visualization by direct integration into the browser, which allows the visualization to be viewed independent of platforms.

1.2 Organization

The remaining part of this thesis is organized as follows: in Chapter 2, we will provide a brief review of the state of art in related areas, including existing systems of 3D urban visualizations over the Internet, urban reconstruction algorithms and systems and mesh simplification techniques. We will then give an overview of our system pipeline in Chapter 3, illustrating the data we deal with and corresponding management and rendering strategy for each data type. In Chapter 4, we will present our level of details algorithm from generation of different resolutions of data to run-time management frameworks. We then proceed to present the system architecture and implementation details in Chapter 5. Chapter 6 concludes the thesis.

CHAPTER 2

RELATED WORK

2.1 3D City over the Web

Delivering interactive visualization of 3D contents over the web has been a constant ambition since twenty years ago. Back in mid 1990s, the Virtual Reality Modeling Language (VRML) file format are defined for representing 3D interactive vector graphics which permitted standardized delivery of 3D contents over the web. One of the main goals behind the design of VRML is platform independence, though displaying such files still requires local applications or browser plugins. Later, specific formats supporting important geodata features such as geographic coordinate system are developed, for instance, GeoVRML and X3D. Various other approaches for streaming large scale 3D contents are created, but they all requires the installation of third-party applications or plugins.

DILAS (Digital Landscape Server) (Nebiker, 2003) is among the earliest projects to visualize large photo-realistic 3D landscape and city models using browser plugins. It is a 3D geoinformation platform scalable to regional and even national landscape and city models. In addition to the use of a specifically designed XML-based 3D object model, DILAS employs level of details and database technologies for scalability and performance.

Numerous other approaches to web-based 3D geodata visualization are developed, and they can be categorized based on the partitioning of work between client and server. The process of displaying 3D virtual world can be divided into three consecutive steps: i) Filter/Select, ii) Render and iii) Display. In step i), data to be displayed is selected from databases or file systems and in step ii), these models are rendered with the geometry and material information. Finally, in step iii), the rendered view is displayed on the client system. If all of the three steps are performed on the client, the system is local, with the example of desktop software programs like *ArcScene* or *LandXplorer*. *Thick client* architecture refers to systems with only the first step performed on the server, and *thin client* refers to systems where both the first and second steps are handled by the server. Before native browser support is available, systems with thick client architecture are usually stand alone software programs like *Google Earth*, or integrated into the web as Java Applets that requires Java Virtual Machine run-time environments like *NASA WorldWind*. On the other hand, projects like WPVS (Hagedorn et al., 2009) try to achieve seamless integration into browser by employing thin client strategy, sacrificing interactivity for accessibility.

Researchers also present rendering or network technologies specifically tailored to visualizing large scale 3D landscapes and cities. Willmott et al. (Willmott et al., 2001) introduce a rendering package combining various techniques and optimizations to visualize large and detailed urban environments interactively. The package is based on a Scene Graph structure for scene management and it combines a number of culling and level of details techniques to achieve interactive frame rates. Royan et al. (Royan et al., 2007) presented a progressive view-dependent streaming solution for interactive visualization of 3D terrains and city models over the Internet, while enabling progressive, scalable and adaptable streaming of large scale models. The solution combines streaming with hierarchical LOD structure for terrains and buildings to allow fast model updates and low server workload, at the same time extendable to peer-to-peer network overlays.

The projects described above are all created before the development and release of standard native browser support for 3D scenes. At that time, the predominant way for displaying massive 3D virtual city on the way is through the use of plugins or Java Applets. Such approaches have clear disadvantage, as the user has to install the plugins first, and may fail to do so due to distrust, incompatibility or security reasons. For fast and easy access, it is crucial to offer seamless integration into the browser without having to obtain additional software besides what is normally present on a computer connected to the network.

The recent release of WebGL has led to the development of various projects with the goal of exploiting 3D contents directly within browsers. WebGL is a low level API providing access to graphics hardware on the client machine, which include a number of hardware accelerated functionalities such as shader programs and vertex buffers. It is exposed through the Document Object Model element of HTML5 Canvas, which is supported natively by modern browser. Various high-level libraries and engines are built on top of WebGL standard to ease the development of 3D programs. Popular WebGL based high-level rendering engines include *Three.js*, *GLGE*, *SceneJS*, *PhiloGL* and *Processing.js*.

Since the release of WebGL, a number of projects for running 3D virtual city directly in the web are built on top of WebGL. The most notable examples are the web version of Google Earth and Nokia Map 3D. In addition to providing global landscape information as virtual globe, these projects also contain detailed 3D views for selected urban areas with high level of realism and accuracy. Bing Maps 3D also provides detailed visualization of 3D urban environments but it requires installation of local client software.

2.2 Urban Reconstruction

Urban reconstruction is a topic of significant interest in multiple research communities, ranging from computer graphics, computer vision to photogrammetry and remote sensing (Hu et al.,

2003) (Musialski et al., 2013). The suitable input data for urban reconstruction algorithms comes in various sources, while the majority can be categorized to imagery and LiDAR scans (Light Detection And Ranging), which can be acquired either from the air or from the ground. Both types have their strengths, as images are easy to obtain at the same time has high resolution and density, while laser scans provide semi-regular and semi-dense 3D structure. For levels of details, airborne data is usually used for coarse building reconstruction and terrestrial data is more suitable for building and facade details.

Image-based modeling handles reconstruction from imagery, relying on stereo vision which recovers the 3D coordinates from multiple distinct images. In a first stage, image processing methods, such as SIFT (Lowe, 2004) and SURF (Bay et al., 2008), are used to detect rotation and scale invariant feature points in multiple images, which can be matched later. Robust estimation algorithms such as RANSAC (Fischler & Bolles, 1981) can be used to improve accuracy of matches across images. Once correspondence between images are established, structure triangluation, which is the intersections of rays from camera centers to the point in 3D space, can produce the extrinsic parameters of the cameras and the point coordinates in 3D space. The iterative process of the above procedure is referred to as structure from motion(SfM), where matching and projective reconstruction is carried out incrementally. Bundle adjustment methods (Triggs et al., 2000) (Agarwal et al., 2010) (Wu et al., 2011) are employed at the end to compute accurate camera parameters and point positions for the whole set of images. The described process of SfM registers images, generate camera orietations and positions as well as produces a sparse 3D point cloud. To construct structure with more details, the next step is dense stereo methods, which generate 3D positions for each pixel in the images. Dense multi-view stereo algorithms are able to deliver reconstruction results with sufficient quality for urban scenes.

Various urban reconstruction systems based on imagery are proposed to handle reconstruction of large urban areas. Frahm et al. (Frahm et al., 2010) present a system that takes millions of unstructured images from the Internet and delivers reconstructed dense structures from stereo fusion. In order to handle the huge amount of data, the system employs clustering as well as parallel computing on multi-core CPU and graphics hardware. Vu et al. (Vu et al., 2012) propose a dense multi-view stereo method that is optimized to reconstruct large scale outdoor scenes from images, which is able to produce highly accurate and detailed models through visibility based surface extraction and variational refinement. Poullis and You (Poullis & You, 2009) develop a system for massive reconstruction from airborne images and LiDAR data, which reach the scale of several thousand buildings. In addtion to the above systems that produce polygonl meshes of urban areas, Snavely et al. (Snavely et al., 2006) (Snavely et al., 2008) (Snavely, 2009) introduce a system that allows navigation in urban environment through sparse point clouds. They utilize sparse SfM to develop a system for exploration of urban environments, called *Photo*

Tourism, where the users can navigate in sparse point clouds from SfM and browse registered photographs.

2.3 Mesh Simplification

Level of details simplification of objects has been an extensively research area in computer graphics (Luebke et al., 2002). The existing techniques can be roughly grouped into topology-preserving and topology-changing simplification. In the category of methods that preserve topology, simplification algorithms typically perform local operations in each iteration step in a greedy manner. Different local operators are proposed, including edge collapse (Hoppe et al., 1993) and vertex pair collapse (Schroeder et al., 1992). In addition, various error measures are introduced to measure the output quality and guide the simplification process, such as the quadric error metric (Garland & Heckbert, 1997). As the computation for quadric metric is simpler and more efficient, the method is quite widespread despite the fact that it does not garantee limits of screen space error. For topology-changing simplifcation, Rossignac and Borrel (Rossignac & Borrel, 1993) propose the vertex clustering method that applies a 3D grid to the object and select a representative vertex for each cell, contracting others. The method has been extended by a number of works, such as (Low & Tan, 1997).

To handle exceedingly large meshes that cannot fit into the main memoery, various out-of-core simplification methods are proposed. Lindstrom (Lindstrom, 2000) introduces an algorithm based on vertex clustering that computes and accumulates quadric error metric for each cluster seperately and the memory requirement of simplification is linear to the output model size. Lindstrom and Silva (Lindstrom & Silva, 2001) also develop an out-of-core vertex clustering algorithm to handle cases where output model cannot fit into the main memory. Hoppe (Hoppe, 1998) and Cignoni (Cignoni et al., 2003) propose another category of methods that devide the model into smaller blocks, simplify each block and stitch the results for further simplification.

2.4 Level of Details Management

In this section we will review published techniques to modulate level of details in run-time. For LOD selection, a number of factors can be included to determine the desired level of resolution model, including distance, size, priority and perceptual factors (Luebke et al., 2002). Distance-based criteria are first used heavily in flight simulators back in 1980s and are still commonly used in popular graphics API and game engines today. Size-based techniques are similarly popular and it involves calculating approximated size of object after projected to the screen, using bounding spheres or axis-aligned bounding boxes. More sophiscated bounding volumes are

also investigated to provide better fits of the geometry, such as ellipsoids or oriented bounding boxes (Luebke & Erikson, 1997). In addition to distance or area selection criteria, the switching of LOD can be further modulated by assigning priorities to objects, as well as the hysteresis technique. A number of researchers have proposed priority ranking schemes such that more important objects are degraded less, such as the semantics-based solution proposed by Funhouser (Funkhouser & Séquin, 1993) which assign priority value to each object proportional to the object type's semantic importance. Hysteresis is a LOD technique that inroduces a lag in LOD transitions to reduce flickering effect between two levels of models when the view point hovers near transition threshold distance. This is done by slightly increasing the threshold for switching to a coaser object and slightly decreasing the threshold to switch to a finer object.

The transition between two LODs at the selection threshold can be noticeable and abrupt, which is often called "popping". To reduce popping effects, the common approaches used are alpha blending or geomorphs. Alpha blending blends two LODs together along the transition by gradually changing the opacity of the objects, such that the object to be switched in gradually becomes visible and the object to be replaced gradually fades out. Though providing smooth transitions, this technique has the disadvantage of rendering two versions of object simultaneously within the transition region. Geomorphing techniques blends two objects by morphing the vertices in one LOD to those in the other. A number of such geometric interpolation methods are proposed, such as the interpolation of vertex split and edge collapse in progressive mesh algorithm proposed by Hoppe (Hoppe, 1996).

CHAPTER 3

SYSTEM OVERVIEW

Our system is based on an automatic approach to reconstruct 3D models of large urban environments from aerial images. Though the reconstruction process is not the main focus of this work, We will briefly describe the reconstruction pipeline we employed in order to give a complete overview of our system. Then we will present our strategy to manage and render different intermediate data in the reconstruction pipeline, as well as the final products, which is a set of dense polygonal meshes representing reconstructed urban scenes.

3.1 Reconstruction

The input of reconstruction is a series of uncalibrated oblique images taken from low flight aircrafts. In a first stage, we perform a structure from motion (SfM) procedure to register the input images, orient and place their cameras and recover 3D structures. Structure from motion is a process that recovers camera poses and 3D points simultaneously. First, for each image a set of feature points are extracted. These feature points then matched to determine unique pairs of corresponding points among multiple images. In this step, instead of using sparse features in the image which may lead to problematic reconstruction, we rely on the quasi-dense matches propagation method proposed by Lhuillier and Quan (Lhuillier & Quan, 2005) to generate more uniform matches.

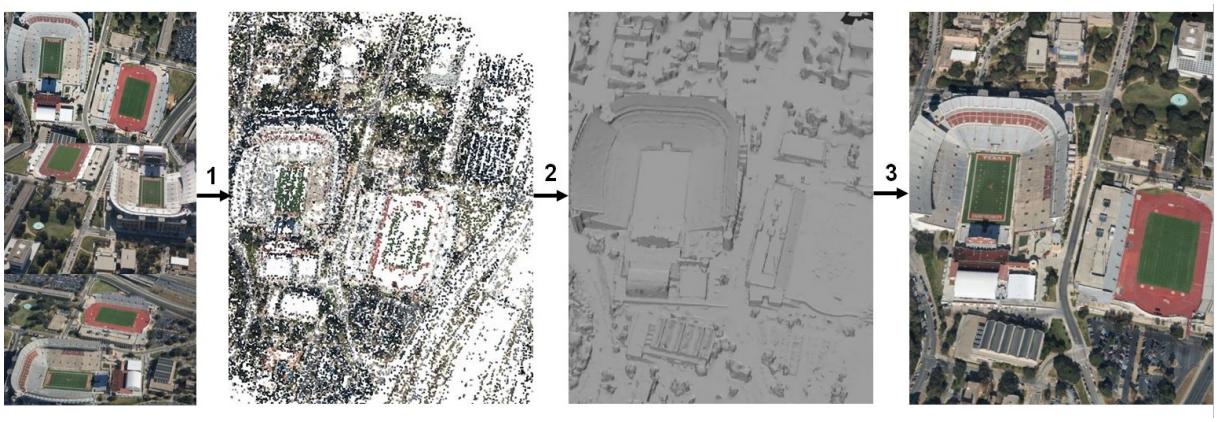


Figure 3.1: Reconstruction pipeline. 1) SfM to generate point clouds; 2) Multi-view stereo to extract and refine mesh; 3) Texture-map refined mesh.

While SfM delivers a set of registered images with their camera parameters and point clouds of 3D structures, the point clouds generated by SfM is rather sparse. In order to obtain more dense structure, a reconstruction that performs per-pixel matching for calibrated images is used, which is referred to as stereo reconstruction. While SfM relies on feature points that are detected robustly, for stereo reconstruction the searching for matched points is highly ambiguous and general smoothness constraints are needed to regularize the reconstructed surface. If some shape priors are known in advance, stereo reconstruction will be efficient and less noisy points will be generated.

After regularization and refinement of the extracted surface using various constraints, the resulting mesh is then subjected to a photo-realisitc texture mapping procedure. Because the models are geometrically imperfect, we use a method proposed by Sinha et al. (Sinha et al., 2008) to optimize textures combined from patches of multiple images using graph-cut. For each pixel in the texture, the assignment of color value from a selected image is done in order to minimize an object function, which penalize the incorrect assignment of an image to an individual pixel and its neighborhood.

3.2 Platform for Visualizing Reconstruction

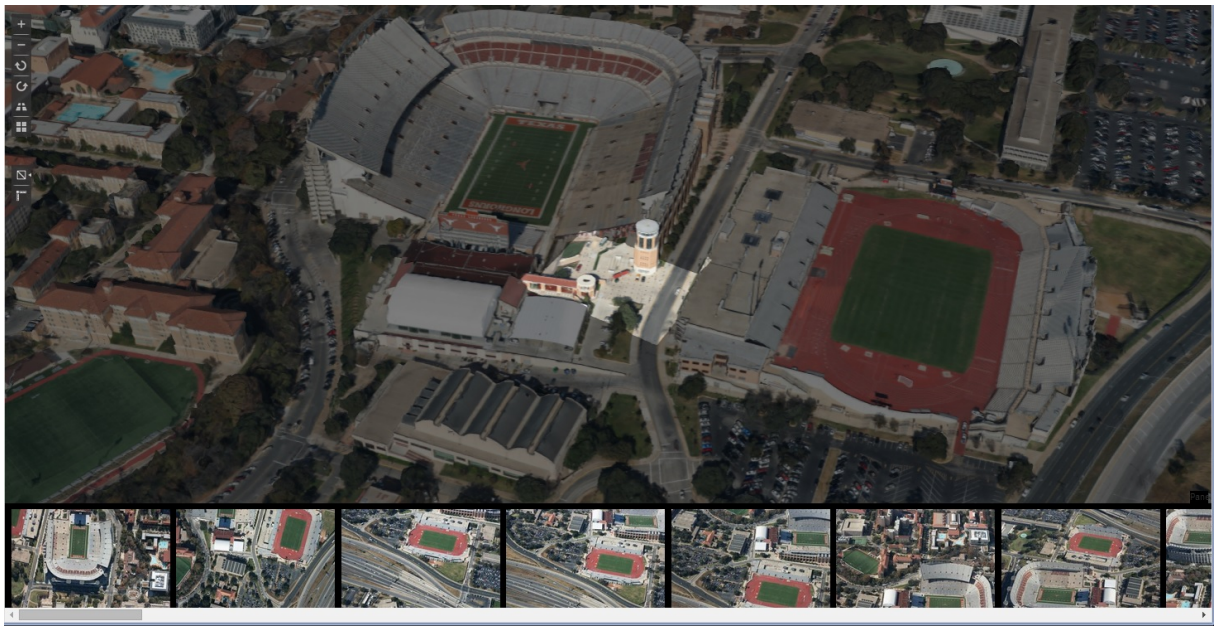


Figure 3.2: Input images are grouped and filtered by association to tiles, such that related images will only be shown if a corresponding tile is picked.

Given the reconstruction process described above, our system aims to efficiently visualize each intermediate result in the pipeline, namely the input image collection, the set of reconstructed 3D points and the polygonal meshes with or without textures. Due to the shear amount

of data, this is a challenging task both in data management and in the rendering of results. In the following sections, we will describe the characteristics of each data type and the methods we use to manage and render the data in our web-based platform.

3.2.1 Data Management

Despite the different types of data, our system uses a uniform data structure to handle all data types, which is a hierarchical tiling system. This uniform underlyging data structure divides the data into a regular grid of tiles and this grid is hierarchical in nature. Each tile is the unit of management. We will cover the details of this tiling system in Chapter 4.1.2.

Images

The input set of high-resolution aerial images for a particular urban area in moderate scale comes in tens of thousands in number and terabytes in size. It is unrealistic to browse the image collection as it is without appropriate filtering and selection methods. In order to assign logical and discrete grouping to the images, we associate each image with tiles in the final reconstructed scene that are visible in the particular image. This is done by first finding corresponding cameras obtained in SfM for a tile and then locating images associated with the camera. The relationship of images to tiles is stored in a relational database and the information of images as well as their thumbnails will be retrieved when a particular tile is picked, as is shown in Figure 3.2. It is worth mentioned that this browsing method is hierarchical because of the fact that the underlying tiles have hierarchy. In other words, picking a larger tile higher in level will retrieve more images relavant to the tile.

Point Clouds

The large scale SfM in the reconstruction pipeline generates large amount of reconstructed 3D points that are impossible to be visualized at once. To allow complete coverage of the entire point set at further distance as well as preservation of density at closed-up views, we adopts a hierarchical approach that generate different level of density of 3D points at different viewing distance. To obtain these levels of resolution, we use a resampling strategy on the 3D points to filter out redundant or less useful points (Fang & Quan, 2010). The details of resampling algorithm will be discussed in Section 4.2.

Meshes

The final product of the urban reconstruction process is a network of dense polygonal meshes covering the urban area. Compared to building models generated by manual modeling process,

the mesh produced by our approach is usually much denser, with higher number of primitives covering a same area. If many of these dense meshes are to be visualized at once to cover a larger area of urban scene, the total number of primitives will soon overwhelm the rendering capacity of graphics hardware. In order to provide elevated views of large area of urban scenes, we again use a hierarchical level of details method to simplify and deliver these data. Since the interactive visualization of reconstructed results is the major part of our application, this part will be covered in details in Chapter 4.

Objects

On top of the polygonal meshes, we can further segment the scene into objects of semantic meanings, for example individual buildings and trees. Then, instead of having a maps consisting of a dense meaningless network of meshes, the final scene can be composed of by semantic objects. These objects can be organized and displayed by the same hierachical tiling framework of our system, similar to the management of point clouds and meshes. Indeed, the data organization framework used in our system is generic, which can accomodate any type of concrete data as long as the data can be divided into tiles along a plane. Besides objectization, the system can also incooporate other types of data from various sources using the same framework, for example the overlaying of a geographic information system layer.

3.2.2 Rendering

The final delivery of data to end users is through an integrated platform built seamlessly into web browsers. The data resides in the server organized by database or file system, and is streamed progressively to the application according to priority defined by level of details calculation. The front end application, which is an interactive visualization of the reconstructed points or scenes, employs various optimizations to maximize the use of rendering resources and network bandwidth. The details of this part will be presented in Chapter 5.

CHAPTER 4

HIERACHICAL ORGANIZATION

4.1 Overview

Given the massive amount of data generated in large scale urban reconstruction, it is unrealistic to fit the original data with high fidelity and details into display, as the shear amount of data is out of the capacity for smooth animation and interactive frame rate. In order to bridge complexity and performance, level of details (LOD) techniques are employed, which regulate the amount of detail used to represent objects in the world based on the perceptual importance of the objects.

4.1.1 Discrete Level of Details

The basic frameworks of LOD techniques for rendering polygonal meshes can be classified into three major categories, depending on how it manages level of detail: discrete, continuous and view-dependent LOD. *Discrete LOD* creates multiple versions of the same object during an offline pre-processing stage, which typically applies simplification uniformly across the object. At run-time, the application switches between different versions of the pre-made models for every objects in the view, basing on how far away the object is from the camera or similar metrics. *Continuous LOD* defers from discrete LOD from when it computes the desired level of details. Instead of computing and storing discrete LOD versions offline, continuous LOD method reads from a data structure encoding continuous details at run-time and dynamically generate the desired model from such information, at better granularity. *View-dependent LOD* is built upon continuous LOD, using simplification algorithms which are dependent on the viewing direction of the camera. Since it selects the most suitable level of details for the current view, this method achieves the best granularity among all the three methods and leads to the best fidelity of the scene under the same rendering budget.

Although continuous and view-dependent LOD provides more precise and fine-grained details, they come at a significant price. Both schemes require extra computation at run-time to evaluate, simplify and refine the models, and due to the fact that models need to be continuously processed whenever the view is changed, the required recalculation frequency is higher. In addition, these methods need extra storage in the main memory to hold the data structure. Under circumstances where the application is limited in computing power and memory resource, the traditional discrete LOD method remains superior due to its low run-time computation costs. A further advantage of discrete LOD is the decoupling of mesh simplification and rendering,

which makes it possible to adopt time-consuming simplification algorithms and allows further optimization in terms of rendering format. In our application, we choose to adopt discrete LOD scheme because of the resource constraints in web-based environment.

4.1.2 Hierachical Tiling System

To organize discrete LOD models, we represent each city as a regular grid of data such that each tile of the city can be independently loaded and rendered. Simply dividing the map into equal-size regular tiles may surfice for small city block, but it will generate a huge number of tiles for large scene if tile size is small for good granuality. A hierarchical spatial data structure such as a quadtree is necessary to cope with the large scale of our data. The root node of this spatial quadtree is an rectangle containing the entire scene of the city. For each subsequent finer resolution level, the quadtree recursively divide the current node into four child nodes with equal size. Each tile in every level is represented by an individual mesh, and despite the fact that a parent tile is 4 times in size of each of its children, we aim to keep the polygon counts comparable for meshes of each level in our simplification process. In this way, a quad tree with m levels have the coverage of $2^m \times 2^m$ tiles in the finest level with a series of coarser tiles at successive power-of-two resolutions. In other words, an overview covering the whole city scene can be quickly provided by displaying the root tile, at the same time complex details can be gradually revealed by subdividing children tiles of finer resolutions when the view zooms in.

Before adopting the quadtree data structure, we tried to implement the system based on a flat hierarchy of grids. In other words, each tile in the grid was of the same size and different levels of resolution only differred in details but not size. The result was not quite satisfactory because even the coarsest level would contain the same number of tiles as the finest level. When an overview was needed, a large number of tiles would have to be loaded before the entire space is filled. In the hierarchical tiling scheme, the coarsest level consist of only one very large tile and thus the overhead in loading separated objects is significantly reduced.

4.1.3 Geographic Reference

The ultimate goal of our 3D map application is to visualize data of the entire globe. To achieve this, we need to incorporate an object space coordinate system where the absolute position of each spatial point can be determined with respect to a global geographic frame of reference. *Universal transverse Mercator* (UTM) system is such a map projection system which is widely accepted and used. This system partition the globe into 60 zones along the longitude direction, each with its local origin (ϕ_0, λ_0) at the intersection of the central meridan with the equator. A spatial point anywhere in the world is thus refered to by the UTM zone number and its local map projection coordinates (X_p, Y_p). The corresponding geodetic latitude and longitude (ϕ_p, λ_p) of

the point can be determined by inverse conversion from transverse Mercator projection. In our current application, the object space coordinates of our city models are linear to their corresponding local UTM coordinates and thus allow accurate conversion to their absolute geodetic coordinates.

4.2 Simplification and Resampling

The hierarchical quadtree organization of data is applicable to both 3D point clouds and reconstructed meshes. The strategy to generate different levels of resolution for both types of data, however, is different. The reconstructed 3D points are in essence unorganized, without topological concerns when simplification is applied. We can resample the points based on certain criteria to generate point clouds of less density. For polygonal meshes, on the other hand, we employ a vertex clustering algorithm to reduce vertices. The simplification is performed in an out-of-core manner due to the large size of our input models.

Mesh Simplification

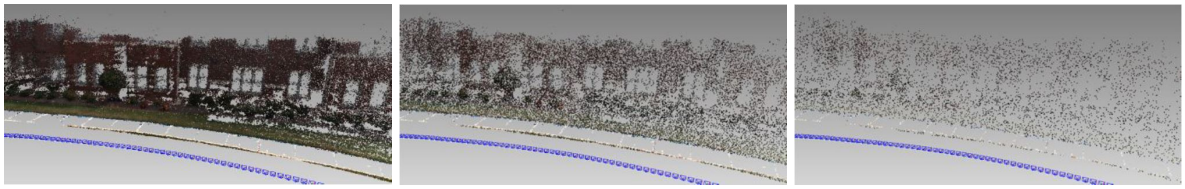
In this section, we will briefly describe the simplification process employed to generate different resolutions of triangular meshes. We use the out-of-core quadric metric vertex clustering method proposed by Peter Lindstrom (Lindstrom, 2000). The algorithm divides the initial mesh into a regular rectilinear grid, and each cell in the grid replaces all vertices inside it with a representative vertex. In order to compute this representative vertex, the quadric matrix $\mathbf{Q}$ for each triangle $(\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3})$ is first computed using:

$$\mathbf{Q} = \begin{pmatrix} \mathbf{A} & -\mathbf{b} \\ -\mathbf{b} & c \end{pmatrix} \tag{4.1}$$

$$\mathbf{n} = \begin{pmatrix} \mathbf{x_1} \times \mathbf{x_2} + \mathbf{x_2} \times \mathbf{x_3} + \mathbf{x_3} \times \mathbf{x_1} \\ -[\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}] \end{pmatrix} \tag{4.2}$$

The resulting quadric $\mathbf{Q}$ for the triangle is added to the corresponding cluster quadric matrix of each of the three vertices of $(\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3})$, depending on which cluster the vertex belongs to. After the quadrics of all the triangles is added, $\mathbf{Q}$ is block decomposed and the linear system $\mathbf{Ax} = \mathbf{b}$ is solved for the representative vertex position. This vertex clustering algorithm is performed in a out-of-core manner for the entire large mesh, where the vertices is stored on disk as a list of triplets of triangle vertices. While this vertex list is being read sequentially, only a table of quadrics and a list of output triangles reside in the memory. In this way, the memory consumption is only linear to the size of the output mesh and independent of the input.

Point Cloud Resampling



(a) (b) (c)

Figure 4.1: Resampling of reconstructed 3D points. The point cloud is resampled at 100%, 20% and 5% rates.

For point clouds, the problem of generating different resolution of LOD can be reduced to resampling points at different resample rates, thanks to the unorganized nature of point clouds. The resampling procedure filters out the least useful points in a greedy way. To select candidate points that are less useful, our criteria focuses on two aspects: first, poorly reconstructed points that are obtained by small baseline or small angles between reprojected rays for triangulation should be filtered out; second, the remaining points should cover the entire reconstructed scene and be in relatively uniform distribution across the scene. Mathematically, the uncertainty of a reconstructed point can be measured by the normal covariance matrix. The uniformness of point distribution can be measured by density of points in image or 3D space. The redundancy score of a point can therefore defined by a combination of uncertainty measured by covariance and density of points around it in both 2D and 3D space. Points with higher redundancy score are removed first in the resampling process.

4.3 Run-time Frameworks

Given the offline processing of mesh simplification, we still need an efficient framework to manage the replacement of LOD in the run-time. The criteria to select a particular level of resolution needs to be determined, as the choice will affect the final scene constitution and has direct impact on efficient use of resources. Loading and replacing different levels of tiles should be done in a seamless way to avoid popping. In addition, we employ a number of optimizations in the framework, including design of a suitable cache and adapted culling methods.

4.3.1 LOD Selection

For run-time management of discrete LOD, a basic question to ask is when to switch to a particular resolution. Intuitively, coaser meshes should be used for objects that are further away or smaller since fewer of their features will be visually noticeable. We originally employed a distance measure where the level of resolution for an object is determined by the distance

between the object and the camera. In a dataset with m levels of details, the resolution levels ($level_0 \ \ to \ \ level_m$) are assigned $m-1$ corresponding distance thresholds ($l_0 \ \ to \ \ l_{m-1}$), and the appropriate level to display for each object is computed as follows:

$$level(l) = \begin{cases} level_0 & where \ \ l > l_0 \\ level_i & where \ \ l_i < l \leq l_{i-1} \quad for \ \ 0 < i < m \\ level_m & where \ \ l \leq l_{m-1} \end{cases}$$

where l is the 3D Euclidean distance between the camera and the center of the bounding box of the object. This distance-based selection is extremely efficient as it only involves the computation of the euclidean distance between two points, which is basically a square root operation. Despite its simplicity, selecting LOD level based on distance is problematic as the range thresholds are sensitive to a number of changes. For example, a change in the scaling of the objects will affect the values of all thresholds; the same applies to changes in display resolution or the camera field of view. Additionally, using a single point to represent the position of an object is in itself inaccurate and will lead to noticeable discrepancies under certain situations.

The alternative way we use to select level of details is based on screen space size of the objects, which takes into account the entire shape of the object and is insensitive to changes in object scaling or projection parameters. For each object O, we use the vertices $\{\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_8}\}$ of the bounding box of O to serve as an approximation; the positions of these vertices are stored as world coordinates. The matrix $M_{ProjectionView}$ used for world space to screen space conversion is precomputed using the multiplication of the projection matrix and the view matrix, as $M_{ProjectionView}$ remains the same for all objects given the same camera position. The projection of O into screen space is thus reduced to eight vertex-matrix multiplications.

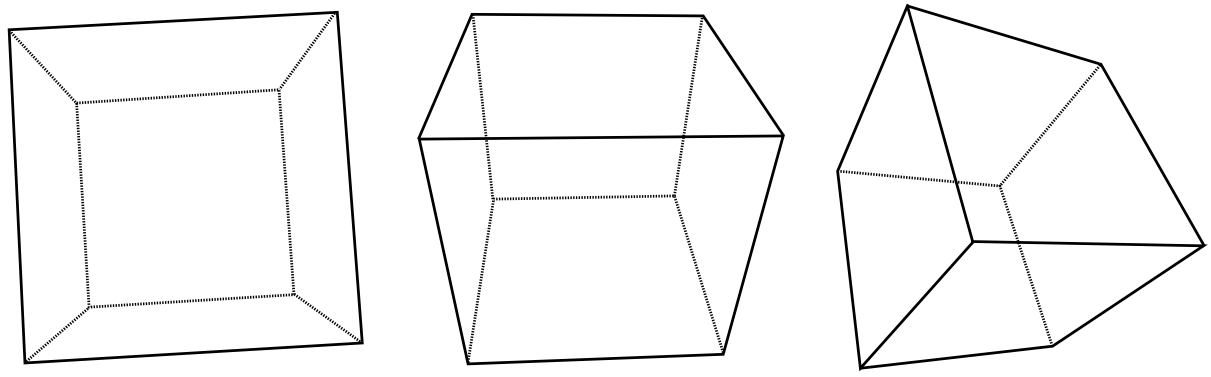


Figure 4.2: A bounding box may have one, two or three front facing faces. The projected area of the bounding box is equal to the sum of areas of its visible faces.

After the projection, we perform perspective division on each vertex to get the correct projected coordinates in Euclidean space. Then we proceed to calculate the area of the projected bounding box, which is equal to the sum of the areas of all the one, two, or three front facing

rectangles, as is shown in Figure 4.2. Specifically, each face of the projected box consists of two triangles with their vertices in anti-clockwise order $\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}\}$. The triangle area is computed as cross product of two ordered edges and the area will be discarded if it is negative, as it means that the triangle is back-facing:

$$S = \sum_{i=1}^{n} (S_{\Delta i})^{+}$$

$$S_{\Delta i} = \frac{1}{2}((\mathbf{x_1} - \mathbf{x_2}) \cdot (\mathbf{x_3} - \mathbf{x_2}))$$

The projected area of the bounding box is then compared to the thresholds for each level to determine the appropriate resolution for the object, similar to Equation 4.3.1. We only need to specify one threshold value manually though, since the adjacent levels of details in our quadtree data structure differ exactly 4 times in size, meaning we can scale the remaining level thresholds using a constant factor. In practice, we fix the area value for the finest resolution level across all datasets, and set the scaling factor such that the coarsest level will show approximately when the camera is at its furthest position allowed.

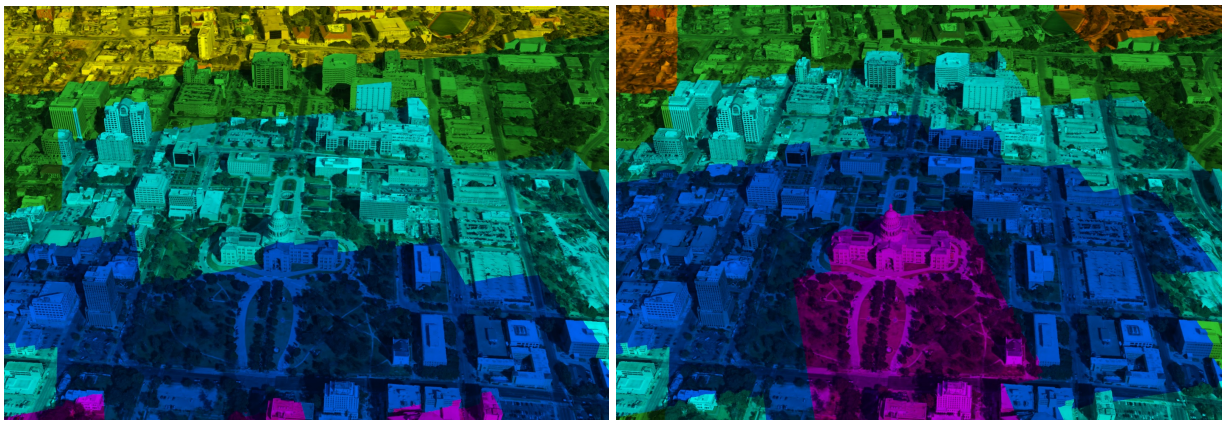


Figure 4.3: Result of the LOD selection criteria. Selecting LOD based on projected area of a tile tends to place finer level at the bottom of the screen (left). By modulating the selection with the distance from the screen center, finer levels are placed toward the center, which is usually the focus of users (right).

In addition to the calculation of screen size above, we further modulate our LOD selection criteria with perceptual considerations. As is described above, selecting LOD of the tiles based on their projected area on the screen have the benefits of higher accuracy and insensitivity in regards to object scaling and projection matrix. However, using a solely area-based selection criterion tends to render an image where the finer level lies in the bottom of the screen, as is shown in Figure 4.3. This is correct under our initial assumption where nearer objects should have finer LOD. However, despite the fact that nearer tiles occupy larger space on the final

scene, it is not necessary that these tiles are perceived more intensely by the user. The user usually focus his gaze on the center of the screen, and details of other area becomes less perceptible as their angular distance to gaze center decreases, due to the characteristics of human peripheral vision. As a result, we should try to place finer models at the center of the screen where their details are more noticeable. by incorporating the distance of each tile from the screen center into LOD selection.

4.3.2 Quadtree LOD Calculation and Loading

An important property of the spatial quadtree is that the parent and its child nodes cannot be visible at the same time, as the child nodes is located inside the parent. To be more precise, a node cannot co-exist with any of its recursive parent or recursive child, since they share part of the spatial content. This property is illustrated in Figure 4.4(a).

On the other hand, the visible nodes of a spatial quadtree should cover the entire scene, without empty space unoccupied by any node content. This implies that if one node reaches the LOD threshold for its level, effectively all three neighbors of this node or their recursive children should be visible in the final scene even if they do not pass the selection criteria, in order to fill up the entire space taken up by their parent. This property should be taken into account when calculating the desired resolution level for every node. It also affects the loading process where a temporary coarser tile is to be replaced by a finer tile, which is its child node in the quadtree. Under such situation, simply replacing the parent with one child node will leave holes in the space; instead, the parent should only be replaced when all of its four children finish loading.

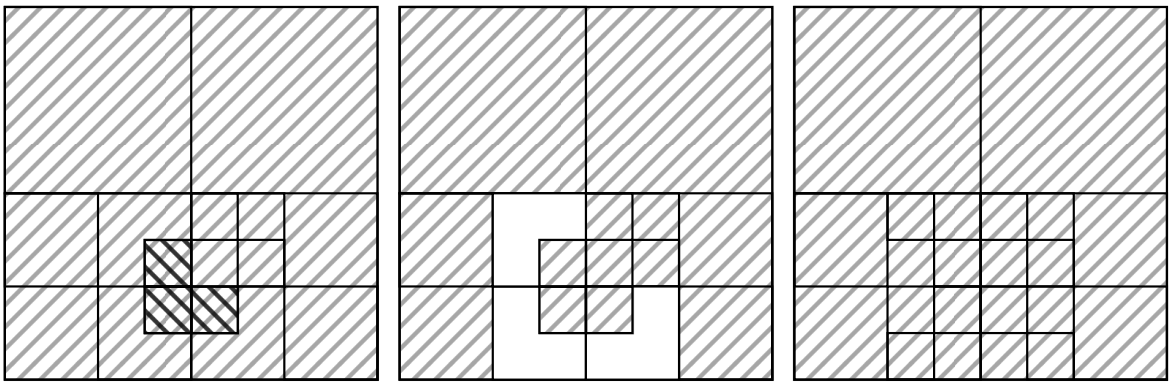


Figure 4.4: (a) A node cannot be displayed together with its parent or child node, otherwise they will occlude each other; (b) If some but not all children of a node are displayed, it leaves empty space unoccupied by any node; (c) An example of a complete scene without occlusion or empty space.

In addition to the special characteristics of spatial quadtree, we have yet another factor that

influences the design of our LOD calculation and loading algorithms, which is the asynchronous loading from server. As our system is web-based, the mesh and texture files for each tile are expected to locate in the server instead of in the local machine. Thus, network requests should be sent to retrieve each file, with potentially largely variable transfer time due to network bandwidth, as well as potential data loss. To cope with these properties, we separate calculation and loading into two decoupled process to have tighter control over loading.

Here we describe how we perform LOD calculation and loading on top of the spatial quadtree. At initiation, we build up the quadtree using a doubly linked tree data structure. Each node of the quadtree stores information about its level, tile coordinates, bounding box, pointers to its child nodes and pointer to its parent node. Additionally, to ensure random access to arbitrary tree node for such tasks like querying current LOD resolution, we build a hash table mapping the pointer locations of every node to its identifier, which consists of its level and tile coordinates. In every frame, if the position or orientation of the camera has changed, we will calculate the appropriate resolution level for each tile using a recursive procedure, which will be stored for look up in the subsequent asynchronous loading process. This calculation basically traverse the quadtree in a top-down manner and determines if the level of the current node is appropriate for final display using the selection criteria described in Section 4.3.1. The procedure is summarized in Algorithm 4.1. Please note that we use smaller level number to represent coaser tiles.

Algorithm 4.1 CalculateLOD(n)

Input: n, the current quadtree node;$M_{ProjectionView}$ and $frustum$, the current camera parameters;

Output: the desired resolution level for every node in the quadtree;

```
if frustum intersects the bounding sphere of n then
    p'_i = M_ProjectionView p_i        ∀p_i ∈ BoundingBox(O)
    S = Σ_{i=1}^{n} (S_Δi)^+
    if S < Threshold_l or n is leaf node then
        l = level(n)
        resolution(n) ← l
        return l
    else
        for all children n_ci of n do
            l_ci = CalculateLOD(n_ci)
        end for
        l_max = max{l_c1, l_c2...}
        resolution(n) ← l_max
        return l_max
    end if
else
    resolution(n) ← −1
    return -1
end if
```

Running Algorithm 4.1 will update the desired resolution level of every tree node for the current camera position. The time consuming step in this recursion is the calculation of projected area of a node; this step will only be called if the bounding sphere of a node as well as those of all its parents intersects the camera frustum, thus pruning out unnecessary calculations.

This algorithm can be called whenever the camera position or orientation is changed. In practice, we find that it is unnecessary to calculate LOD every frame if the user is continuously moving the camera, since the details of objects in motion is harder to detect. In our implementation, we force a fixed minimum time interval between each successive LOD calculations.

After the calculation of desired LOD resolutions, we proceed to loading. The loading process tries to gradually replace the old scene with a new one faithful to the current calculated levels, without creating temporary holes. To give a smoother transition, coarser levels should be shown first as they take lesser time to load, while waiting for the final finer levels to assemble. To avoid creating empty space unoccupied by any visible tiles, the proper timing to correctly remove previous unwanted tiles and to replace parent tile with children should be carefully controlled. To avoid having two different resolution tiles at the same spatial point, the removal of unwanted tiles should be both accurate and complete, such that no more and no less tiles are in the final scene then the desired calculations.

We use a queue to store the loading process and limit the maximum number of requests that are sent out at the same time. The loading process can be separated into two parts: a pre-loading part which checks nodes that needs to be loaded and populate the loading queue in a defined order, and a post-loading part which is called after the tiles finish loading and controls when the loaded tiles are ready to be added to the scene, replacing their parents. In the following description of these two parts, we ignore caching, which will be covered in Section 4.3.3.

The pre-loading part starts with the root node of the quadtree everytime LOD calculation changes. Theoretically we could traverse the tree and place all loading requests at one time, but we find that this often results in redundant items in the queue because the desired level changes more frequently then loading is able to catch up. Thus, after the first time, it is only called after a node is about to be added to the scene. Each time, it compares the level of the node to the desired level which is stored after the LOD calculation, through the lookup table. If the stored level is strictly less than the node level, meaning that neither this tile nor any of its children are needed, we recursively remove this tile and its children. If the stored level is strictly larger than the node level, we put all the next level children of this node to the loading queue. If the levels are equal, we only put the current node to the queue if this is the root node, since each time afterwards the current node should already be loaded and visible.

The post-loading part is called every time a node finishes loading. Its main functionality, which is determining when the loaded node is ready to be displayed, is executed as the following. If none of the node's recursive parents is currently in the scene, then this tile can be

safely added to the scene. However, despite the fact that this tile can be added to the scene, it is not necessary that it should be added, because of situations where finer resolutions substituting this tile are already available. We will skip the tile under the circumstances where its children are needed and all of them are ready to display. If any of the node's parents is in display, we check the immediate parent to see whether it is currently in display and all its children tiles are loaded; and if this is the case, all children tiles are added to the scene at once and the parent tile is removed. After this, it places loading requests for its next level children nodes, as mentioned above. Furthermore, for all the process above, we recursively remove all children of a node whenever the node is added to the scene. This process can be summarized in Algorithm 4.2.

Algorithm 4.2 DisplayTile($tile$)

Input: $tile$, the loaded tile, ready to be submitted to rendering pipeline;
Ensure: the tile is displayed without creating occlusion or holes in the scene;
  $parentVisible \leftarrow true$ if any of $tile$'s recursive parent is in display
  $skippable \leftarrow true$ if all immediate children of $tile$ are needed and loaded
  if $parentVisible == false$ **and** $skippable == true$ **then**
    skip
  else if $parentVisible == false$ **then**
    display $tile$
  else
    check immediate parent of $tile$ and display all its children if they are all loaded
  end if
  run pre-load procedure on $tile$

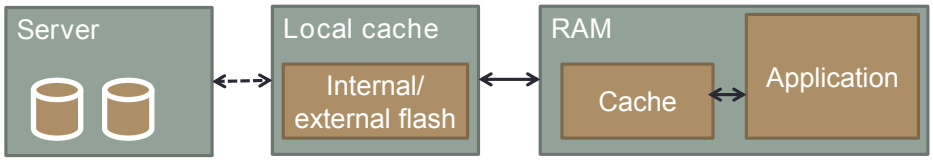
Throughout the loading process, we checks in every step to see if the current status is consistent with the most updated desired LOD table. For example, after a tile finishes loading, it may no longer be needed as it was before the load requests was sent. Similarly situation may arise when waiting for all next level children of a node to load. When inconsistence is detected, we abort the current progress and cache the results accordingly.

4.3.3 Cache

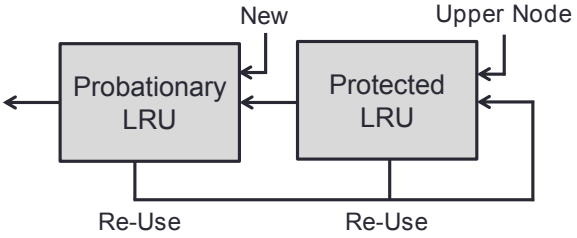
Since loading mesh and texture files from the server and decompress them into renderable formats cause significant overhead, we need a local cache that stores loaded and decompressed data ready to be submitted to the rendering pipeline. The replacement algorithm of this cache should be compatible with the quadtree structure and the order of tile replacement, to maximize cache hit chances. As is described in Section 4.3.2, the order of tree nodes in the loading queue are neither strictly depth-first nor strictly breadth-first, but rather a combination of both depending on the actual order that previous tiles are loaded. That being said, we still prefer a breadth-first traversal, as the top layers are repeatedly needed every time loading restarts due to the coarser to finer level replacement policy.

We design the cache with a strategy similar to a SLRU (Segmented Least Recently Used) cache. LRU cache, as the name suggests, removes the item that is accessed least recently to

make room for newly added item. A typical SLRU cache is divided into two seperate LRU caches, a probationary part and a protected part. A newly added item is first added to the most recently used end of probationary cache and a further hit of the item will promote it to the protected part. An item to be removed from the protected part is first inserted to the most recently used end of probationary part, giving it a second chance before it is removed. We modify the SLRU strategy to improve hit chances of the cache under normal sequence of quadtree loading. In particular, quadtree nodes in the upper levels in the tree go directly to the protected segment from cache misses, since these tiles have high chances to be accessed again when the view point changes.



(a)



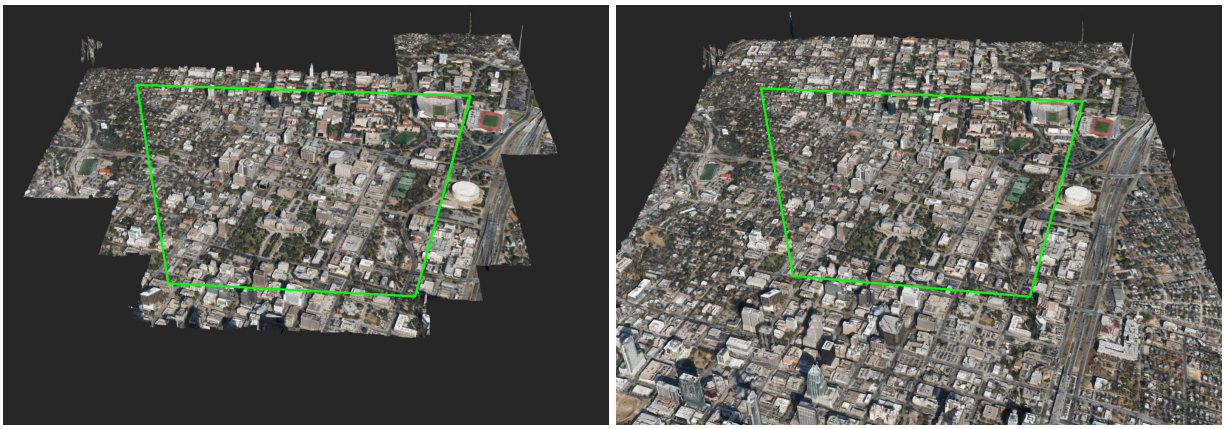
(b)

Figure 4.5: (a) Data transmission in our system. The cache in RAM is used to stored objects in decompressed renderable formats. (b) Our adapted SLRU design. Nodes that are upper in quadtree level are assigned a priority tag and misses of such nodes go directly to the head of the protected segment.

4.3.4 Culling

View frustum culling is an object based culling method that discards objects that are not in the current viewing volume. The view frustum is a cut pyramid defined by six clipping planes, which can be computed each frame from the projection and model view matrices. If a tile is entirely outside of the frustum, it is essentially invisible and we will cull it away.

To accurately cull as many tiles as possible while avoiding excess computations, we combine view frustum culling with LOD calculation step in Section 4.3.2. In the quadtree LOD calculation step, we tries to fill the entire spatial quadtree with different resolution tree nodes even if the tree node indeed lies outside the view frustum. Specifically, we first define a set of visible nodes and their corresponding resolution, and then fill the empty space left behind



(a) (b)

Figure 4.6: Results on a typical dataset with (a) or without (b) view frustum culling. The original view port is marked in green.

with appropriate neighboring nodes. The potentially invisible tiles are within these space filling neighboring nodes, and since their bounding boxes have already been tested against the view frustum in LOD calculation, we label those tiles whose four bottom vertices of bounding boxes all fall outside of the frustum as invisible. These tiles are culled away in view frustum culling.

However, as LOD calculation is not performed every frame due to its computation cost, relying culling on this step will lead to delayed response when the user move the camera. For example, if the user pans slightly to a previously invisible area, the tiles will be empty if LOD calculation does not catch up, leading to interruption of perceived realism of the virtual environment. A potential means to alleviate this problem is by slightly enlarging the view frustum in calculation, which will make a thin circle of areas adjacent to currently displayed tiles readily visible. However, this solution does not eliminate the effect if the camera moves fast enough to pass the enlarged frustum, especially in camera rotation. The method we use is an additional fast, cheap culling test that only adds culled tiles back to the scene to compensate these situations, which is basically a test of bounding sphere intersection of frustum.

We test the performance of our view frustum culling algorithm using three of our datasets. We perform a walkthrough passing through random areas in the data and record the triangle counts. As is shown in Table, the culling step is able to reduce primitive counts significantly, meaning significant performance gains.

Dataset	Without Culling	With Culling
Austin Downtown	4.44 million triangles	3.08 million triangles
Entire Area of Austin	6.71 million triangles	4.45 million triangles
Googleplex	2.09 million triangles	1.45 million triangles

Table 4.1: Average number of total triangles for the walkthrough of three sample datasets using view frustum culling.

In addition to view frustum culling, we also enable backface culling in the rendering pipeline to skip rear facing polygons, as the city models are completely opaque. The movement of camera is restricted in distance and angle with respect to scene center, thus it will not enter inside areas where back facing polygons are culled. Another commonly used culling technique, occlusion culling, is not performed in our application due to low depth complexity of the scene. This is because the camera angle is restricted to at least 30 degrees from the floor plane, i.e. street level views are not allowed.

CHAPTER 5

SYSTEM IMPLEMENTATION

5.1 Front End Architecture

The front end web application displays the detailed urban models within a styled canvas with user interaction functionalities to allow exploration. It can be a stand alone web page, or it can be readily integrated into customized web applications. In addition to supporting viewing and exploration on desktop computers, a counter part designed specifically for mobile devices is also developed.

The front end architecture is built on top of a popular light-weight rendering engine named Three.js. Among the basic components is the Context object, which handles the render window and exposes the canvas and WebGL interface. The Scene object manages the currently visible models and processes them for rendering. The data contained in the Scene object is continuously changing depending on view points and level of details calculations, though the Scene object does not have knowledge of the underlying structure of data and does not handle this part. In the scene there is a Camera object which controls what can be seen. The camera can be switched to either perspective or orthographic and we support corresponding functionalities for both. The Camera object has a Controller, which responds to user interaction events and defines the navigation within the scene.

The contents of the city environments are managed by the hierarchical tiling system described in Section 4.1.2. When the view point has changed, the contents to be updated are calculated using information stored in the quadtree as is described previously. The pending tiles are then checked against local cache and will be inserted into the Scene object if they are in the cache. If not, a loading manager will submit asynchronous request to fetch model data on the server, following priority of the tiles and limiting load requests per frame to a fixed number, to increase interactivity.

Different from many virtual 3D city systems where the contents are terrains and a separate set of building models, the model data for final reconstruction results in our system is a set of dense polygonal meshes and the terrain and buildings are combined in the same model files. The models are originally in wavefront obj format and then converted to utf8 binary format using the compression algorithm described in Section 5.1.4. The binary file as well as texture files can be loaded asynchronously into the viewer and decompressed to generate vertex buffer objects, which will be directly handled by the Scene object.

The point cloud data is esseantially managed and loaded the same way as mesh data, since the quadtree tiling structure applies to both types of data. In terms of implementation, both types are extension of a Tile object, which holds information of each quadtree node such as pre-calculated bounding box and handles LOD calculation and loading logic. The difference between mesh tiles and point tiles is mainly in rendering, as the 3D points are rendered as billboards. In our system, switching the views from point cloud to mesh representations only involves change in renderable data.

Our system is easily extendable to support a Geographic Information Systems (GIS) layer to display points of interest and other geographics information. We have implemented a basic version of GIS layer overlayed on the 3D map. The information on roads and buildings are taken from open street map database, which is a publicly available world map. To retrieve such GIS information, a request containing the latitude and longitude bounding box and the zoom level is sent to the server following each view point change, and results from queries of map database will be returned. To render GIS information on top of the underlying 3D map, we use two methods depending on the data type. For lines and polygons such as roads, we first render the 2D map on a HTML5 canvas and use the resulting image to texture a quad that is placed slightly above the ground plane. For points of interest, we use billboards for better readability in all orientations.

5.1.1 User Interface

The user interface of our system is minimalistic as the majority of the screen real-estate is devoted to displaying the virtual urban environment. As is shown in Figure 5.1, the user interface contains a series of navigation options, switches for viewing models with or without textures, and other helper functionalities such as measuring distance or picking tiles.

We have tailored the navigation interactions to best facilitate exploration of the 3D urban scenes. Basic navigations in 3D include rotation, panning and zooming. The camera controller uses an arcball model for rotation, which spins the scene around its center when the user drags the mouse. For panning, the user can drag the scene precisely along the floor plane; by precisely we mean that the distance panned is calculated according to the distance dragged, such that the point under the cursor remains the same in the scene. This is implemented by first calculating the coordinates of starting and ending points under the cursor using ray intersection with the floor plane. Then the camera position is translated in world space according to the distance and direction of the end point with respect to the start point.

In addition to navigation using mouse inputs, we also provide corresponding animated yawing, pitching and zooming through buttons. These interactions are animated smoothly from the starting view point to the final view point, to provide a visual clue to the user of the changes

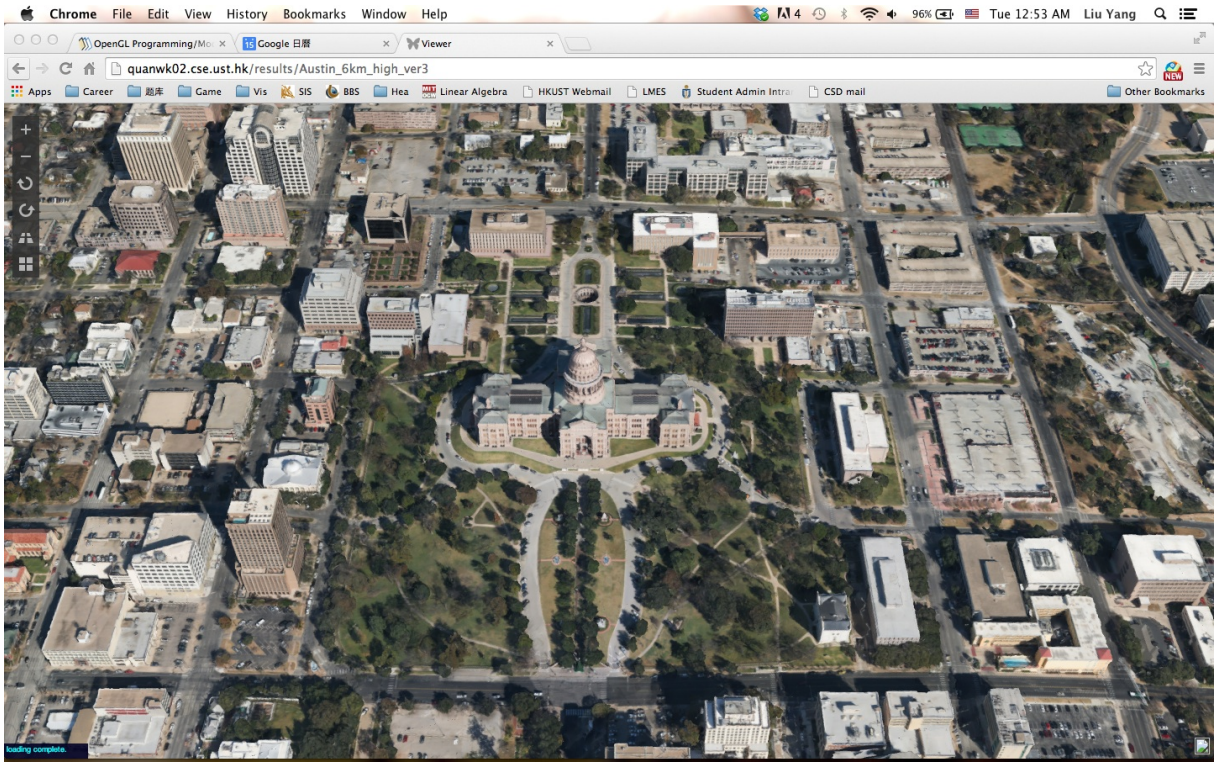


Figure 5.1: User interface of our system.

made. To make a better user experience, such animations are accelerated at first and gradually decline to static when approaching the destination view point. To do this, we simply make the per-frame change in camera position dependent on the remaining changes to be made, which means that the camera will move less in the trail if its current position are closer to the destination.

As we confine the view point to be within a certain distance and angle from the center, the otherwise free navigations will not pass a certain boundary if it exceeds the allowed zone. To signal the user about reaching such boundaries in an intuitive way, we adopt an analogous animation which repels the camera in the opposite direction as if it has hit a spring. To be more precise, when the user drag pass the boundary, the camera motion will slow down proportionally to the distance away from the boundary, similar to the situation where a spring is harder to pull when the extended length is longer; when the user release the mouse, the camera will shrink back to the boundary location as if it is on a spring that is just released. This animation is applicable to all zooming, panning and rotation cases, and it avoids the use of additional texts or sounds to indicate view point boundary while being fairly intuitive.

As we mentioned previously, the user can switch between a perspective camera and a orthogonal camera to view the 3D scene. While the perspective camera allows free exploration of the virtual city from a wide range of angles, the main purpose of the orthogonal camera is to give a view from top like the traditional 2D map. The camera controller is the same ex-

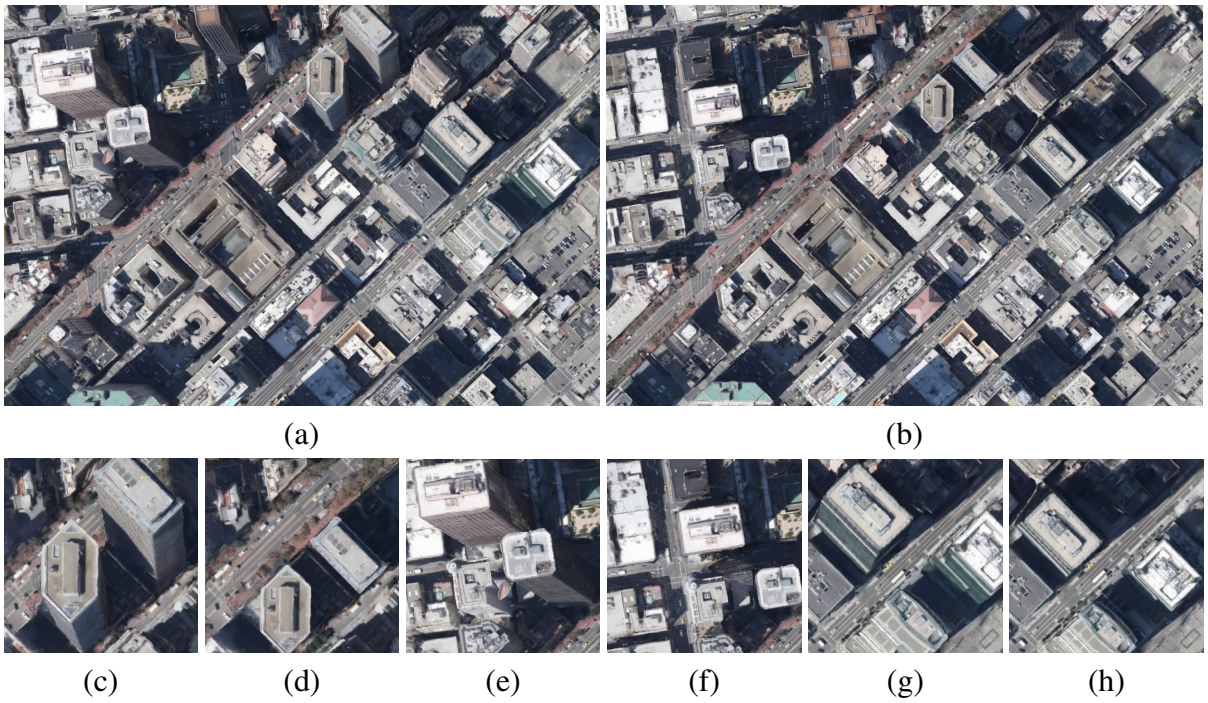
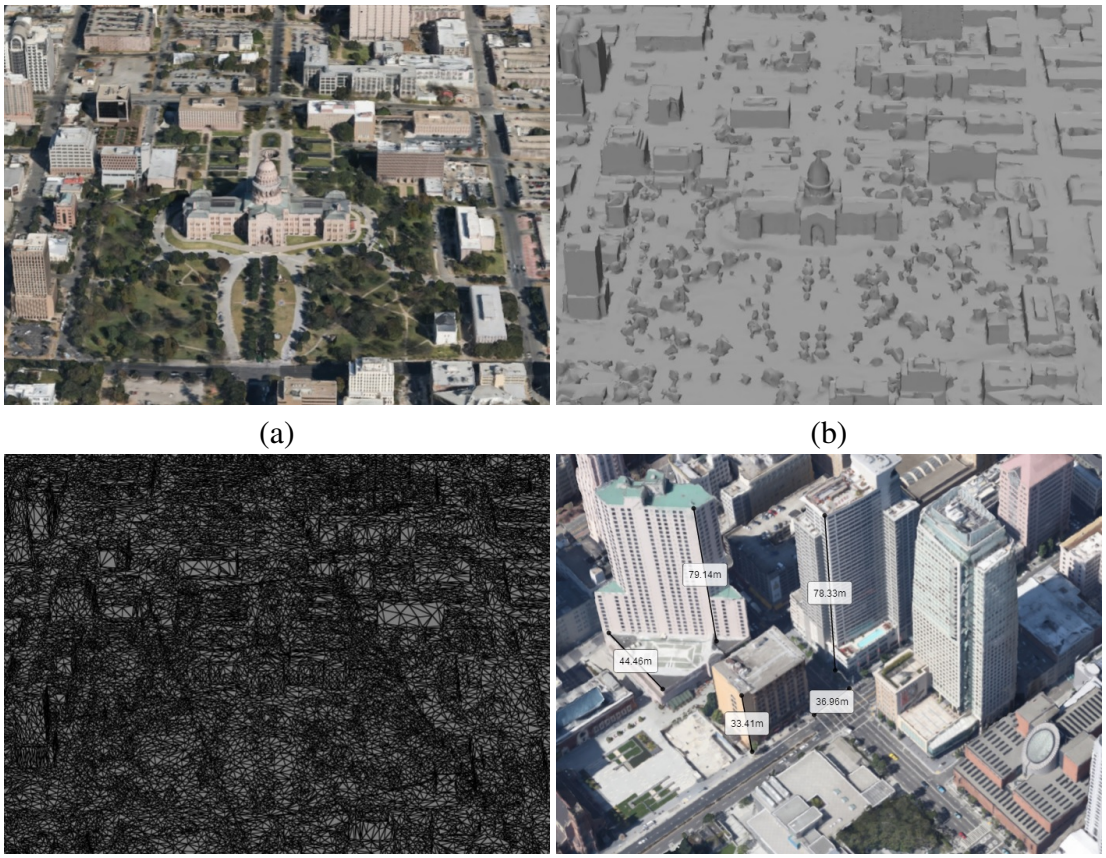


Figure 5.2: Users are able to switch between a perspective(a) or orthographic(b) camera. The orthographic view can serve as an accurate approximation of the two dimensional map. The bottom row shows side by side comparison of selected areas in perspective (c) (e) (g) and orthographic (d) (f) (h) view.

cept that pitching rotation is not allowed. The user can switch between cameras at any point and the new camera position will be calculated correspondingly. To provide a smoother visual clue about transition between these two camera types, we also creates a unique animation to morph the scene from perspective to orthographic projection or vice versa. Such animation is implemented through the gradual change of camera field of view. For instance, to animate from perspective camera to orthographic camera, the field of view is decreased each frame while maintaining the same width and height of view port, until it reaches a small enough value which projects the view with imperceptible difference from orthographic projection.

Despite the animated navigations described above, the user interface also contains additional features that are more interesting to a smaller set of audience. To give specialists a better inspection of underlying city models, we provide switching between texture, surface or wire-frame view, as is shown in Figure 5.3. Since the measurements of real world distance from map may be useful to professionals like surveying and mapping engineers, we also implements a feature which allows the user to pick two points and gives real world distance between them, by first getting world space coordinates of the two points using ray intersection of mesh and then converting the distance using geospatial reference. This feature is shown in Figure 5.3(d).



(a) (b)

(c) (d)

Figure 5.3: Users are able to switch between texture(a), surface(b) or wireframe(c) view of the urban scene. In addition, a measuring utility is provided that allows users to measure real world distance between two points in the scene(d).

5.1.2 Support for Mobile Devices

One of the major benefits of integrating 3D urban maps directly into standard browsers is the ability to work cross platform. This platform independence extends to mobile devices with appropriate browser and hardware support. However, as rendering massive city models are typically resource demanding, the application needs to be specifically modified to address the limited memory and smaller screen of mobile devices.

The user interface on mobile is redesigned for convenience and aesthetic reasons, as is shown in Figure 5.4. It only contains the basic navigation features, as we assume the mobile version targets general non-expert audience.

The original navigation using mouse are replaced by several multi-touch gestures. In particular, we define the following gestures for intuitive exploration on touch screens: single finger drag for panning, double tap for zooming in, pinch for zooming, two fingers rotate for yawing and two fingers drag for pitching. Although we assign three different tasks to two finger gestures, they are distinctively distinguishable from one another by the following criteria. For pinch, the distance between fingers are increasing rapidly while their relative angle does not

Figure 5.4: User interface on mobile devices.

change much. For rotation, the angles between fingers are changing rapidly but the distance between them remains approximately constant. For drag, neither the relative distance nor the angle change much, but the midpoint position does. The gesture recognition is done shortly after the user presses multiple fingers on the screen and will remain the same until the user releases his fingers. In other words, zooming will not change to yawing halfway through the gesture, for example, to avoid unnecessary confusion due to sensitivity.

To account for the much limited RAM in mobile devices, we have made changes to level of details calculation as well as texture files for mobile. For level of details calculations, we adjust the selection thresholds such that a certain level of resolution needs much larger screen space to be displayed, resulting in selection of coarser tiles overall. Additionally, we skip the finest level entirely to save memory. For texture files, since the video memory on the mobile device is also much limited and textures usually take up a large part of such memory space, we further down-sample the texture images for mobile usage. Though these changes affect the graphics quality of final views, they are acceptable due to the fact that mobile phone screens are also much smaller which make coarser quality less noticeable.

Despite the modifications made, the application running in mobile browsers are basically the same as that on desktop computers. Indeed, our system will automatically choose the version based on the information about client browser and platform.

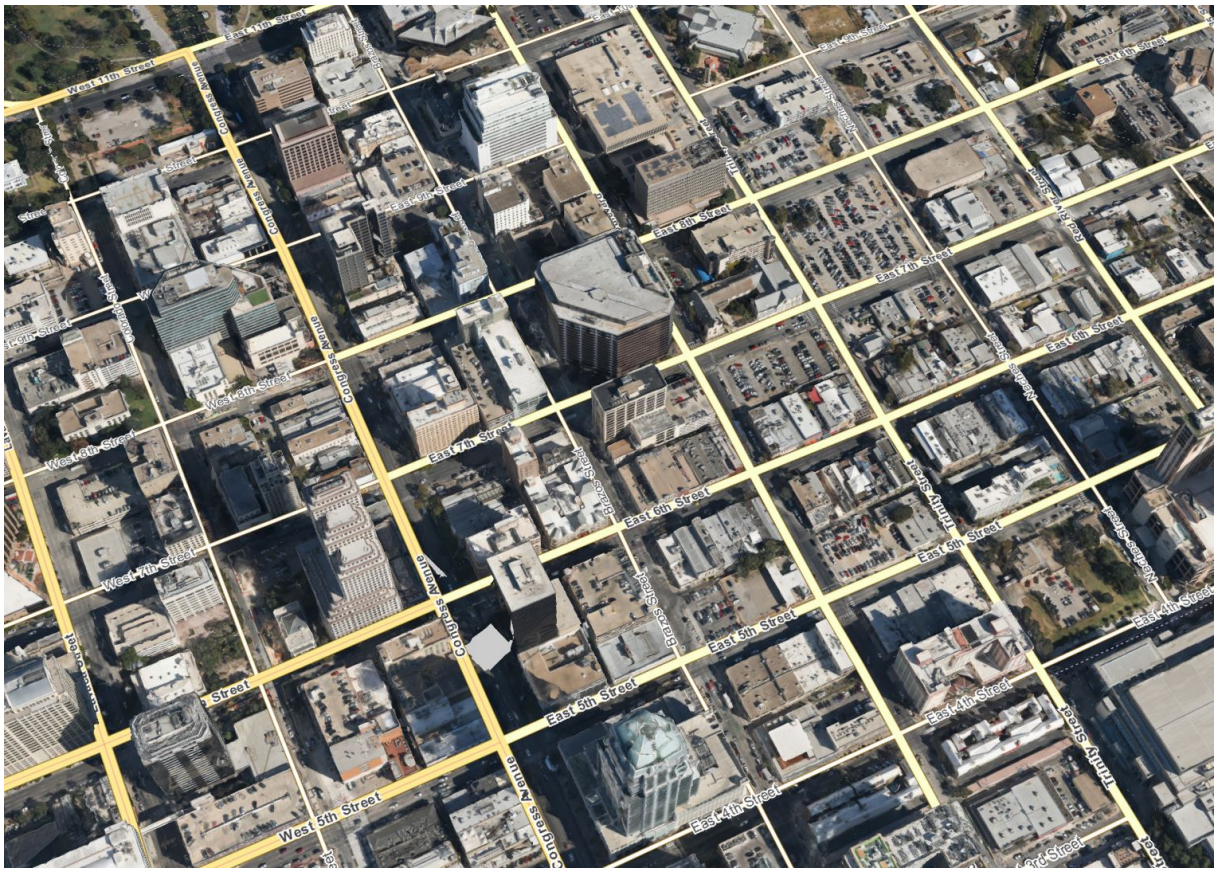


Figure 5.5: A demonstrative implementation of overlaying a GIS layer on our reconstructed 3D map.

5.1.3 Support for Geographic Information System

We have incoorporated a geographic information system (GIS) layer in our implementation. For data acquisition, we use the open source vector map data from the Open Street Map project. As is described in Section 4.1.3, the 3D map data used in our system is GPS aligned and is converted to a geographic coordinate system, i.e. UTM. After applying the same coordinate conversion to vector map, it is able to perfectly align with our model.

For rendering the GIS layer, currently we use a basic method where the vector map is rendered in a separately on a hidden canvas and then the image is used to texture a transparent rectangle that approximate the floor plane. The result is shown in Figure 5.5. Although this method is able to produce a visually appealing map on cities that are relatively flat, it cannot handle datasets where the terrain is largely varying in height, as the texture-mapped floor plane will be occluded by elevated potion of the terrain. Even if the floor plane containing GIS layer is not occluded by for example disabling depth test, when the view point is slanted, noticeable discrepancy appears between alignment of 3D map structures and 2D vector map labelings. Indeed, the alignment will only be correct in orthographic view where the depth of the 3D map is removed. To eliminate this problem, the vector map and points of interests should contain

height information, making them effectively three dimensional. We are now working on extracting the height information for vector map by querying the orthographic projected mesh map and later rendering the vector map as 3D.

5.1.4 Optimizations

To handle large and complex city models, despite the use of level of details techniques described in Chapter 4, we employs compression method for the mesh data file to cope with low network bandwidth. Besides, as long computation time hinders immediate handling of user inputs and eventually causes the browser to freeze, we employ means to avoid excessive long computation in the single threaded environment of browser.

Compression

To reduce data file size and increase loading speed for given network bandwidth, we compress the mesh file into UTF-8 based binary code using the method of Arthur Blume et al. (Blume et al., 2011). The algorithm is able to compress raw bytes of model file by a magnitude of ten and it can be applied in addition to GZIP, a popular HTTP compression built into web servers and clients. The method is based on UTF-8, an encoding scheme availible on the web. UTF-8 has the property of variable-length encoding, which means that values smaller than one byte are literally encoded as one byte while larger values take two or three bytes with some prefix coding, as is shown in Table 5.1:

First Integer	Last Integer	Encoded Length
0	127	1 byte
128	2,047	2 bytes
2,048	65,536	3 bytes

Table 5.1: UTF-8 uses variable-length encoding to encode 16-bit values using between one and three bytes.

Exploiting this property, the method then uses delta encoding to capture the difference between adjacent values in the file, which tends to have much smaller magnitudes. However, delta compression yields signed values while UTF-8 is unsigned, leading to the problem that small negative numbers will be shifted to very large positive numbers. The algorithm then uses ZigZag encoding to interleave small magnitudes such that small negative numbers will be mapped to small positive numbers, shown below as three steps of bit manipulations:

$$ZigZagInteger = (Integer \gg 15) \wedge (Integer \ll 1) \tag{5.1}$$

Decoding of unsigned numbers to the original signed counterparts is also done in bit manipulations:

$$Integer = (ZigZagInteger \ll 1) \wedge (-(ZigZagInteger\&1)) \tag{5.2}$$

This delta encoding scheme is favorable particularly for indexed triangles list, as the optimization of triangle ordering for vertex cache also tries to put close triangles together in order to maximize cache hit rates, which means that delta encoding will likely generate small values between adjacent vertices. In other words, geometry data that is optimized for rendering will also lead to good compression result using this algorithm.

Preventing Prolonged Computations

Computation complexity of the client side in browser applications is limited by certain features of browser implementations. The call stack sizes are rather limited, for example Google Chrome version 25 has a stack size limit of approximately 25,000, making deep recursion in our quadtree structure undesirable. The single threaded nature of Javascript prevents prolonged computation as it significantly decreases responsiveness and eventually freezes the browser.

We make a series of modifications to tackle these problems. First, the quadtree related computations are rewritten to iterative version instead of recursion. This and other potentially long executing computations are broken into small chunks and subjected to a task scheduling manager, which executes one chunk at a time and allows handling of interaction callbacks between consecutive chunks. In this way, user interaction and other callbacks will not be delayed The number of synchronous network requests as well as texture switches are limited to a maximum number per frame to further increase responsiveness.

5.1.5 User Group and Permission

A common concern for web-based applications is permission and copy right issues. We design the server side architecture to protect private datasets and prevent illegal download of model data. In order to access the viewer, the user needs to obtain a user ID and password, which is stored in our user database with associated permission to dataset groups. Every time a data file is accessed either through application requests or directly from URL, we verify the permission of user to the URL and will deny access if the verification fails. Although this checking is done for every GET request sent from the client, in practice this does not have noticeable overhead to the performance of the viewer.

5.2 Results

Our system has already been applied and tested on a dozen of urban reconstruction datasets of large to massive scale. All of the datasets are tuned to achieve interactive frame rates on client machines of moderate hardware capacities. Mobile versions of these datasets are able to run on devices with maximum 1GB memory. For demonstration purpose, we present four representative datasets, focusing on the ability of our system to efficiently organize and render massive scale of data. The computer used to collate the results presented in this section is a Intel Core i3 3.30 GHz processor with 4GB memory and a GeForce GT 640 graphics card. The statistics of the selected datasets are summarized in Table 5.2.

The first set is the reconstruction result of over one hundred square kilometers area covering the city of Austin. The raw mesh files of the entire set exceeds 4 terabytes, after level of details simplifications, the size comes to approximately 70 gigabytes. Compression brings the size further down to 6.5 gigabytes, but it is still out of the capacity of ordinary consumer laptops. Figure 5.6 shows the views of different scale of the city, from a bird eye view of the entire reconstructed area, to a close-up view of the details of individual buildings. Due to hierachical simplification and streaming, the overview covering the entire city consists of approximately 8.35 million triangles taking up roughly 800 megabytes of space, which can be streamed in a few seconds under good bandwidth condition. The subsequent navigation which continuously zoom to individual buildings in street level is completely smooth and responsive, with an average of 30 frames per second on the test computer.

Other datasets presented in this thesis include the campus of National University of Singapore(NUS), the downtown area of San Francisco and the city of Huesca. Though much smaller in size compared to the first set, these examples still benefit from the techniques for large scale data. For NUS dataset, both the point cloud view and the textured mesh view are demonstrated in Figure 5.7.

Dataset	Austin	NUS	San Francisco	Huesca
Num. of Images	54080	944	4682	754
Size of Images	2.85 TB	50.9 GB	253 GB	40.7 GB
Size of Raw Mesh File	4.31 TB	1.79 GB	391 GB	531 GB
Size of Compressed Mesh File	6.49 GB	0.25 GB	0.68 GB	0.71 GB
Size of Textures	10.2 GB	0.41 GB	0.94 GB	1.12 GB
Tile number	320×320	64×64	128×128	128×128

Table 5.2: A summary of the statistics of Austin, NUS, San Francisco and Huesca datasets.

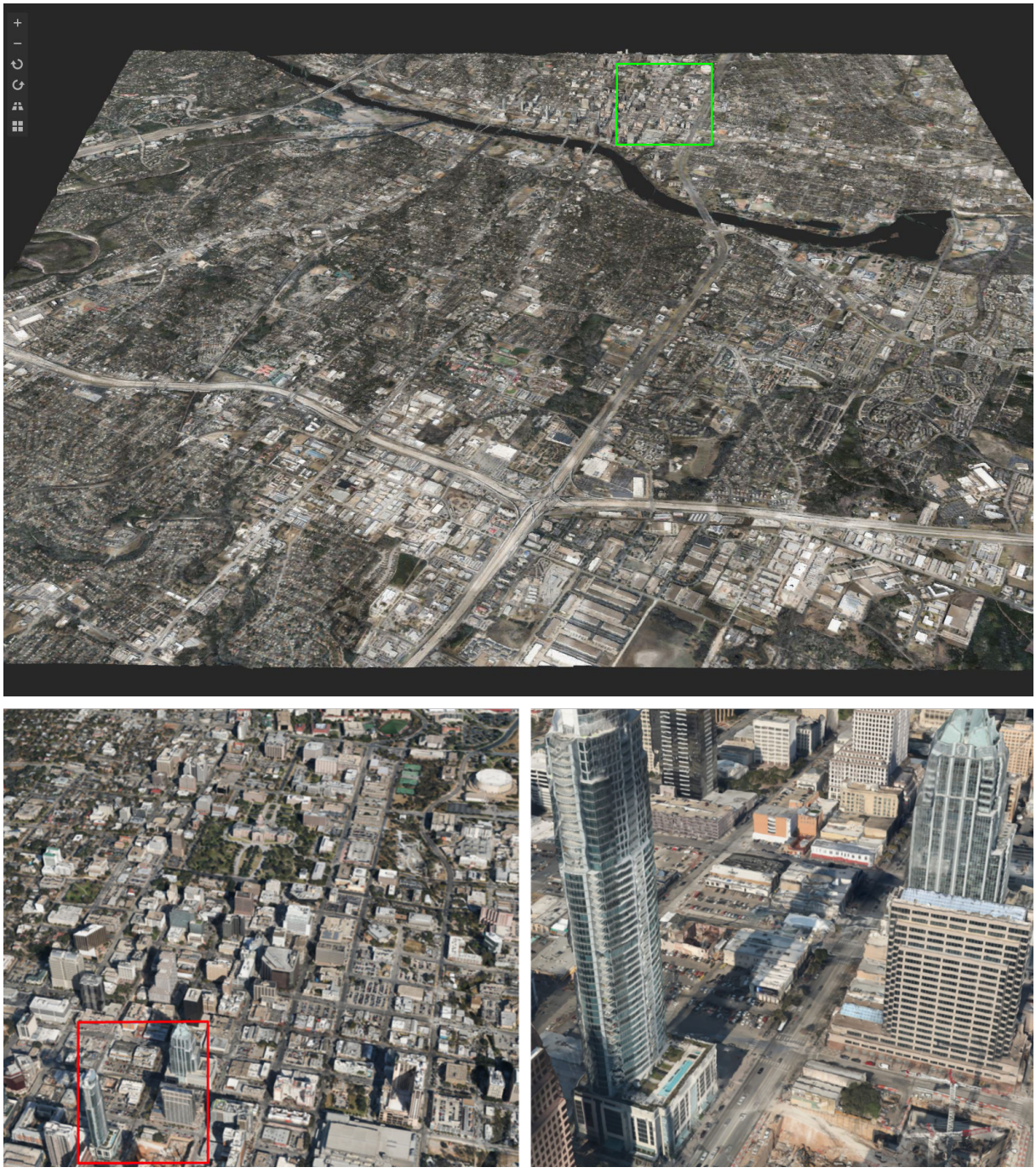


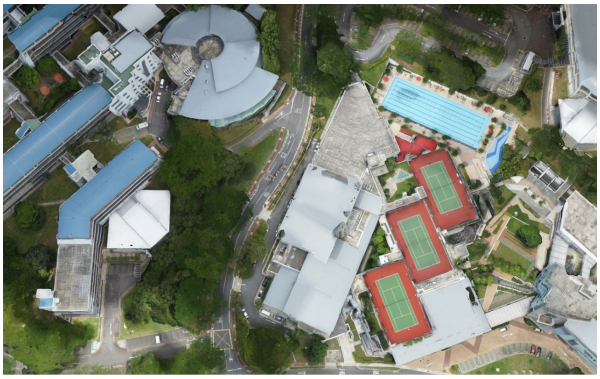
Figure 5.6: The Austin dataset. Top column shows a bird eye's view of a majority of the reconstructed urban area. Bottom left figure shows a close-up view of the green rectangular area at the top, which is an elevated view of austin downtown. Bottom right figure is a further close-up view of the red rectangular area on the left, down to the scale of individual buildings.

(a)

(b)

(c)

(d)



(e)

Figure 5.7: The National University of Singapore dataset. Top column shows an overview of the campus (a). The middle column shows point cloud data (b) and its corresponding textured mesh data (c). The bottom column displays two close-up views of buildings and structures in the campus (d)(e).

(a)

(b)

(c)

Figure 5.8: The San Francisco downtown dataset. Left figure shows an overview of the city (a). The middle and right figures display two close-up views of buildings and structures in the city (b)(c).

(a)

(b)

(c)

Figure 5.9: The Huesca dataset. Top column shows an overview of the city (a). The bottom column displays two close-up views of buildings and structures in the city (b)(c).

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis we present a new visualization engine for interactive exploration of intermediate and final results of large scale urban reconstruction. The scale of the data is handled by hierachical level of details to provide perservation of original complexity at close inspection along with coverage of complete overview at elevated view points. Various optimization methods tailored to the underlying hierachical data, such as cache and culling, are employed to speed up rendering. The system is implemented as a web-based platform, integrating interfaces for viewing input image collections, reconstructed 3D point clouds and the polygonal meshes with or without textures. Special attentions are paid to support the seamless integration of the platform on compatible mobile devices, which include support of multitouch gestures and alterations made to address low video memory and bandwidth.

The system has been tested with numerous datasets, ranging from reconstruction of downtown areas to larger scale covering the entire city. Our strategy is proved to handle such large scale data with satisfactory speed, quality and responsiveness on various client machines with different capacities.

There are multiple venues for future work. The first possible area of extension will be digital mapping, which is the incoorporation of a GIS module to handle processing and rendering of GIS information overlayed on the 3D urban scenes. In the current system, only a basic GIS layer is implemented to demonstrate the ability to incoorporate such an component. One of the possible improvements comes in balancing the occlusion in rendering 2D map lines on top of 3D structures. For example, when a road runs directly behind a row of tall buildings in oblique views, the buildings effectively occlude useful geographic text information along the road, but drawing the road on top of the buildings will be counter-intuitive and confusing. Finding the balance between completeness of information and intuitiveness will be an interesting topic to pursue.

Another interesing extension in digital mapping will be GIS in 3D. Currently, most, if not all, 3D maps still use 2D geographic information. It will be interesting to add height information to geographic points of interest and display them on 3D maps. This will be useful, for example, to label offices and shops on different floors of a skyscraper in order to provide spatial distinctions. It is entirely impossible on traditional 2D maps, as all these points of interests have the same latitude and longitude coordinates and they have to be displayed at the same spatial location on 2D maps. To support such 3D geographic labeling, we need to find good ways to intuitively edit

such information on 3D maps, at the same time develop methods to display such information without occlusion or confusion.

Apart from improvements in 3D digital mapping and GIS, management and post-processing of large scale urban reconstruction data will also be an interesting area for furture work. Our system can integrate an additional component to allow selecting, downloading and uploading after offline modification of certain tiles. This will be useful for running selective post-processing algorithms on particular reconstructed areas. It will also be useful to integrate user interactions for semi-automatic reconstruction algorithms on the platform.

Bibliography

Agarwal, S., Snavely, N., Seitz, S. M., & Szeliski, R. (2010). Bundle adjustment in the large. In *Proceedings of the 11th European Conference on Computer Vision: Part II*, ECCV'10, pp. 29–42 Berlin, Heidelberg. Springer-Verlag.

Bay, H., Ess, A., Tuytelaars, T., & Van Gool, L. (2008). Speeded-up robust features (surf). *Computer Vision and Image Understanding*, *110*(3), 346–359.

Bing Maps (2014) `http://www.bing.com/maps/`. Accessed: 2014-07-10.

Blume, A., Chun, W., Kogan, D., Kokkevis, V., Weber, N., Petterson, R. W., & Zeiger, R. (2011). Google body: 3d human anatomy in the browser. In *SIGGRAPH Talks*, p. 19.

Cignoni, P., Montani, C., Rocchini, C., & Scopigno, R. (2003). External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, *9*(4), 525–537.

Fang, T., & Quan, L. (2010). Resampling structure from motion. In *Proceedings of the 11th European Conference on Computer Vision: Part II*, ECCV'10, pp. 1–14 Berlin, Heidelberg. Springer-Verlag.

Fischler, M. A., & Bolles, R. C. (1981). Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, *24*(6), 381–395.

Frahm, J.-M., Fite-Georgel, P., Gallup, D., Johnson, T., Raguram, R., Wu, C., Jen, Y.-H., Dunn, E., Clipp, B., Lazebnik, S., & Pollefeys, M. (2010). Building rome on a cloudless day. In *Proceedings of the 11th European Conference on Computer Vision: Part IV*, ECCV'10, pp. 368–381 Berlin, Heidelberg. Springer-Verlag.

Funkhouser, T. A., & Séquin, C. H. (1993). 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*, SIGGRAPH '93, pp. 247–254 New York, NY, USA. ACM.

Garland, M., & Heckbert, P. S. (1997). Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pp. 209–216 New York, NY, USA. ACM.

Google Maps (2014) `http://maps.google.com`. Accessed: 2014-07-10.

Hagedorn, B., Hildebrandt, D., & Dllner, J. (2009). Towards advanced and interactive web perspective view services. In *4th international 3D GeoInfo workshop* Ghent, Belgium. Springer.

Hoppe, H. (1996). Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pp. 99–108 New York, NY, USA. ACM.

Hoppe, H. (1998). Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings of the Conference on Visualization '98*, VIS '98, pp. 35–42 Los Alamitos, CA, USA. IEEE Computer Society Press.

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., & Stuetzle, W. (1993). Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pp. 19–26 New York, NY, USA. ACM.

Hu, J., You, S., & Neumann, U. (2003). Approaches to large-scale urban modeling. *IEEE Computer Graphics and Applications*, *23*(6), 62–69.

Lhuillier, M., & Quan, L. (2005). A quasi-dense approach to surface reconstruction from uncalibrated images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *27*(3), 418–433.

Lindstrom, P. (2000). Out-of-core simplification of large polygonal models. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pp. 259–262 New York, NY, USA. ACM.

Lindstrom, P., & Silva, C. T. (2001). A memory insensitive technique for large model simplification. In *Proceedings of the Conference on Visualization '01*, VIS '01, pp. 121–126 Washington, DC, USA. IEEE Computer Society.

Low, K.-L., & Tan, T.-S. (1997). Model simplification using vertex-clustering. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pp. 75–ff. New York, NY, USA. ACM.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, *60*(2), 91–110.

Luebke, D., & Erikson, C. (1997). View-dependent simplification of arbitrary polygonal environments. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pp. 199–208 New York, NY, USA. ACM.

Luebke, D., Watson, B., Cohen, J. D., Reddy, M., & Varshney, A. (2002). *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA.

Musialski, P., Wonka, P., Aliaga, D. G., Wimmer, M., van Gool, L., & Purgathofer, W. (2013). A survey of urban reconstruction. *Computer Graphics Forum*, *32*(6), 146–177.

Nebiker, S. (2003). Support for visualization and animation in a scalable 3d gis environment motivation, concepts and implementation.. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XXXIV-5*(W10).

Nokia Maps (2014) `http://here.com/`. Accessed: 2014-07-10.

Poullis, C., & You, S. (2009). Photorealistic large-scale urban city model reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, *15*(4), 654–669.

Rossignac, J., & Borrel, P. (1993). Multi-resolution 3d approximations for rendering complex scenes. In Falcidieno, B., & Kunii, T. (Eds.), *Modeling in Computer Graphics*, IFIP Series on Computer Graphics, pp. 455–465. Springer Berlin Heidelberg.

Royan, J., Gioia, P., Cavagna, R., & Bouville, C. (2007). Network-based visualization of 3d landscapes and city models. *Computer Graphics and Applications, IEEE*, *27*(6), 70–79.

Schroeder, W. J., Zarge, J. A., & Lorensen, W. E. (1992). Decimation of triangle meshes. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pp. 65–70 New York, NY, USA. ACM.

Sinha, S. N., Steedly, D., Szeliski, R., Agrawala, M., & Pollefeys, M. (2008). Interactive 3d architectural modeling from unordered photo collections. *ACM Transactions on Graphics*, *27*(5), 159:1–159:10.

Snavely, K. N. (2009). *Scene Reconstruction and Visualization from Internet Photo Collections*. Ph.D. thesis, Seattle, WA, USA.

Snavely, N., Seitz, S. M., & Szeliski, R. (2006). Photo tourism: Exploring photo collections in 3d. *ACM Transactions on Graphics*, *25*(3), 835–846.

Snavely, N., Seitz, S. M., & Szeliski, R. (2008). Modeling the world from internet photo collections. *International Journal of Computer Vision*, *80*(2), 189–210.

Triggs, B., McLauchlan, P. F., Hartley, R. I., & Fitzgibbon, A. W. (2000). Bundle adjustment - a modern synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*, ICCV '99, pp. 298–372 London, UK, UK. Springer-Verlag.

Vu, H.-H., Labatut, P., Pons, J.-P., & Keriven, R. (2012). High accuracy and visibility-consistent dense multiview stereo. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *34*(5), 889–901.

Willmott, J., Wright, L. I., Arnold, D. B., & Day, A. M. (2001). Rendering of large and complex urban environments for real time heritage reconstructions. In *Proceedings of the 2001 Conference on Virtual Reality, Archeology, and Cultural Heritage*, VAST '01, pp. 111–120 New York, NY, USA. ACM.

Wu, C., Agarwal, S., Curless, B., & Seitz, S. (2011). Multicore bundle adjustment. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pp. 3057–3064.