VISUALIZATION OF URBAN ENVIRONMENTS

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Türker Yılmaz
June, 2007

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Assoc. Prof. Dr. Uğur Güdükbay(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Bülent Özgüç

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Volkan Atalay

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Assoc. Prof. Dr. Veysi İşler

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

VISUALIZATION OF URBAN ENVIRONMENTS

Türker Yılmaz
Ph.D. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Uğur Güdükbay
June, 2007

Modeling and visualization of large geometric environments is a popular research area in computer graphics. In this dissertation, a framework for modeling and stereoscopic visualization of large and complex urban environments is presented. The occlusion culling and view-frustum culling is performed to eliminate most of the geometry that do not contribute to the user's final view. For the occlusion culling process, the shrinking method is employed but performed using a novel Minkowski-difference-based approach. In order to represent partial visibility, a novel building representation method, called the slice-wise representation is developed. This method is able to represent the preprocessed partial visibility with huge reductions in the storage requirement. The resultant visibility list is rendered using a graphics-processing-unit-based algorithm, which perfectly fits into the proposed slice-wise representation. The stereoscopic visualization depends on the calculated eye positions during walkthrough and the visibility lists for both eyes are determined using the preprocessed occlusion information. The view-frustum culling operation is performed once instead of two for both eyes. The proposed algorithms were implemented on personal computers. Performance experiments show that, the proposed occlusion culling method and the usage of the slice-wise representation increase the frame rate performance by 81 %; the graphics-processing-unit-based display algorithm increases it by an additional 315 % and decrease the storage requirement by 97 % as compared to occlusion culling using building-level granularity and not using the graphics hardware. We show that, a smooth and real-time visualization of large and complex urban environments can be achieved by using the proposed framework.

Keywords: Stereoscopic visualization, slice-wise representation, space subdivision, octree, occlusion culling, occluder shrinking, Minkowski difference, from-region visibility, urban visualization, visibility processing.

ÖZET

YERLEŞİM ALANLARININ GÖRÜNTÜLENMESİ

Türker Yılmaz
Bilgisayar Mühendisliği, Doktora
Tez Yöneticisi: Doç. Dr. Uğur Güdükbay
Haziran, 2007

Bilgisayar grafiklerinde geniş geometrik ortamların modellenmesi ve görüntülenmesi popüler bir araştırma alanıdır. Bu tezde, geniş ve karmaşık şehir ortamlarının üretilmesi ve stereoskopik olarak görüntülenmesi için bir çerçeve sunulmaktadır. Kullanıcının göreceği görüntüye katkıda bulunmayan geometrinin çoğunun elenmesi için, kapatılan alanların atılması ve bakış piramidi dışında kalan alanların ayıklanması yöntemleri uygulanmaktadır. Kapatılan alanların atılması işlemi için daraltma yöntemi, yeni bir Minkowski farkına dayanan yaklaşım ile uygulanmaktadır. Kısmî görüntülemeyi sağlayabilmek için, dilimsel temsil adı verilen yeni bir bina temsil yöntemi geliştirilmiştir. Bu yöntem sayesinde, kısmî görünürlük, depolama ihtiyacında muazzam azaltmalar sağlanarak temsil edilebilmektedir. Elde edilen görüntü listesi grafik işlemci ünitesi tabanlı bir algoritma aracılığıyla görüntülenmektedir. Stereoskopik görüntüleme, gezinti esnasında hesaplanan göz pozisyonlarına dayanmakta ve görüntü listeleri tespit edilmiş kapatılan alanların bilgisi kullanılarak elde edilmektedir. Stereoskopik görüntüleme için bakış piramidi dışındaki nesnelerin ayıklanması işlemi, her iki göz için iki yerine bir kez uygulanmaktadır. Önerilen algoritmalar kişisel bilgisayarlarda kodlanmıştır. Performans deneyleri, kapatılan alanların atılması yöntemi ile dilimsel veri yapısı kullanımının, standart görüntülemenin kullanıldığı bina seviyesindeki kapatılan alanların ayıklanması yöntemine göre performansı; görüntü karesi hızı olarak % 81 arttırdığını; grafik işlemci ünitesi tabanlı yöntem kullanımının da buna % 315 ilave artış sağladığını ve depolama ihtiyacını % 97 azalttığını göstermektedir. Önerilen çerçevenin kullanılmasının, büyük ve karmaşık şehir modellerinin düzgün ve gerçek zamanlı görüntülenmesini sağladığı gösterilmiştir.

Anahtar sözcükler: Stereoskopik görüntüleme, dilimsel veri yapısı, uzay alt bölümleme, sekizli ağaçlar, kapatılan alanların ayıklanması, kapatanların daraltılması, Minkowski farkı, bölgeden görüş, şehir görüntüleme, görünürlük işleme.

Canan'ıma...

Acknowledgement

At the end of this study, I am very grateful that we have succeeded. I say "we", because this could not have been accomplished without the support of some people.

I am very grateful to my supervisor, Assoc. Prof. Dr. Uğur Güdükbay, for his invaluable support, guidance and motivation. I learned a lot from him, especially the endurance needed for this kind of study.

I would like to thank my thesis committee members Prof. Dr. Bülent Özgüç, Prof. Dr. Özgür Ulusoy, Prof. Dr. Volkan Atalay and Assoc. Prof. Dr. Veysi İşler for their invaluable comments to improve this thesis.

I especially want to thank to the other half my heart, my love and my wife, Canan Yılmaz. Without her, not a part of this work could have been accomplished. She gave all her support and motivation to me all the time. She and our two sons, Cantürk and Caner are my all motivation sources.

I want to thank to my father Sami Yılmaz, brother Tibet Yılmaz and my sister Cemile Tanju Duygun for their invaluable support. Finally, I cannot forget mother Aliye Yılmaz, for her endless belief in my success and high hopes. She rest in peace and God bless her.

This work is supported by the Scientific and Research Council of Turkey (TÜBİTAK) under Project Codes 198E018, 104E029, 105E065 and with a Ph.D. scholarship. Al Model is courtesy of Viewpoint Datalabs International, Inc. The heptoroid model is courtesy of the University of California, Berkeley. Vienna2000 Model is courtesy of Peter Wonka and Michael Wimmer. Glasgow Model is courtesy of ABACUS, University of Strathclyde. I want to thank to Oğuzcan Oğuz for his efforts for helping in the development of the City Modeling feature. Thanks to Medeni Erol Aran for helpful comments during the development phase.

Contents

List of Figures

List of Algorithms

List of Tables

List of Symbols and Abbreviations

API	:	Application Programming Interface
CPU	:	Central Processing Unit
FPS	:	Frames per Second
GPU	:	Graphics Processing Unit
IOD	:	Interoccular Distance
LCS	:	Liquid Crystal Shutter
LOD	:	Level of Detail
OC	:	Occlusion Culling
OCTREE	:	A tree data structure, in which each node has eight children.
OPENGL	:	Open Graphics Library, which is one of the most commonly used API.
PARALLAX	:	Distance between the stereo pairs on the screen.
PVS	:	Potentially Visible Set. It is the list of objects or other primitives that is to be rendered, when the user is in a specific view-cell.
QUADTREE	:	A tree data structure, in which each node has four children.
TIN	:	Triangular Irregular Network
VBO	:	Vertex Buffer Objects
VFC	:	View Frustum Culling
VIEW-CELL	:	A small area in the scene, within which the primitives around it can be classified with respect to their visibility status.
VR	:	Virtual Reality

Chapter 1

Introduction

Modeling and visualization of large and complex environments is a popular research area in computer graphics. Recent developments in processors and graphics cards, the amount of available memory and the development of computer graphics modeling and rendering techniques facilitate to run high quality simulations. Applications cover a large spectrum from visual simulations, military training and city planning to video games.

Modern graphics workstations allow rendering of millions of polygons per second. No matter how much graphics hardware evolves, human being is going to crave for what is impractical for those hardware to render at interactive frame rates. Therefore, it has become a race between hardware developers and researchers, to render more detailed graphics by using the lower bounded algorithms that can be achieved at present time.

In general, geometry processing is the main bottleneck of all graphics applications. Even high-end graphics workstations have the ability to draw only a very small fraction of triangles needed to draw large complex scenes at interactive frame rates. Furthermore, virtual reality applications need twice the processing power as needed for their monoscopic counterparts. Therefore, it is crucial to send only the visible parts of the geometry to the rendering pipeline. Besides, if processing power needed exceeds the capacity of the hardware, it is necessary to approximate these parts up to a certain threshold, in order to achieve interactive frame rates.

The advances in graphics hardware allow detection of occluded regions of urban geometry, even with complex 3D buildings. Visual simulations, urban combat simulations and city engineering applications require highly detailed models and realistic views of an urban scene. Occlusion detection using preprocessing is a very common approach, because of its high polygon reduction and its ability to handle general 3D buildings.

Visualizing urban environments is one of the most challenging areas in computer graphics, mainly because of the unorganized geometry and their complex nature. Attempts to reduce this complexity include either preprocessing or assuming simpler geometry for the buildings in the urban environment or both. And since virtual reality applications need twice the processing power of their monoscopic counterparts, it is crucial to send only the visible parts of the geometry to the rendering pipeline. For interactive walkthroughs of large building models or city like scenes, a system must store in memory and render only a small portion of the model at each frame. The most important challenge is to identify the relevant portions of the model, swap them into memory by using a robust database access and render at interactive frame rates, as the user changes position and viewing direction.

In order to send only the related portions of the scene, thereby allowing the hardware to render the scene at interactive frame rates (17 and above frame rates per second), there are mainly three types of culling methods to get rid of the irrelevant portions of the geometry. One of them is *view frustum culling* (VFC) that discards the objects that are out of the field of view. *Occlusion culling* eliminates the parts that are occluded by front objects. The last one is *back-face culling*, which discards those polygons whose normals are facing away from the viewer. Back-face culling works for convex objects. View frustum culling is performed by the evaluation of the plane equations that form the view frustum. Back-facing polygons are eliminated if the dot product of the viewing direction and polygon normal is greater than zero. Back-face culling is mostly implemented in hardware in most of graphics boards. One specific work about back-face culling is [64], where the authors have some improvements compared to the hardware implementation.

In this dissertation, we propose a framework for the stereoscopic visualization of urban environments using a conservative visibility determination algorithm and several other optimization schemes, such as using graphics processing unit (GPU) for rendering and VFC to speed up the rendering process. The proposed VFC culling scheme is for the stereoscopic rendering. The main attention is given to the occlusion culling process, where the most performance gain is achieved.

The visible geometry in a typical urban walkthrough mainly consists of partially visible buildings. Most occlusion-culling algorithms, in which the granularity is buildings, process these partially visible buildings as if they are completely visible. To address the problem of partial visibility, we propose a storage scheme, called *slice-wise representation*, that represents buildings in terms of slices parallel to the coordinate axes. We observe that the visible parts of the objects usually have simple shapes. This observation establishes the base for occlusion culling where the occlusion granularity is individual slices. The proposed slice-wise representation has minimal storage requirements. We also propose to shrink general 3D occluders in a scene to find volumetric occlusion.

Generally the techniques for speeding up the rendering process is applied separately for each eye during a stereoscopic visualization. In our approach, VFC for stereoscopic visualization is performed once, instead of two for both eyes. The usage of the slice-wise representation is utilized for the GPU and high performances are achieved in stereo, by accessing the predetermined visibility information directly.

The proposed framework is tested on several urban models ranging from 500K to 46M polygons. Empirical results show that, significant increase in frame rates and decrease in the number of processed polygons can be achieved using the proposed slice-wise occlusion-culling with GPU-based rendering and the VFC approach for stereoscopic visualization, as compared to an occlusion-culling method, where the granularity is individual buildings and regular VFC approach is applied for the stereoscopic visualization.

1.1 Contributions

The contributions of this dissertation can be listed as follows:

- an automatic city modeling approach and its algorithms, which is able to model a city given that the ground plans are available in electronic formats, (see Appendix A),
- a navigation space extraction algorithm, which determines the view-cells to be used for the occlusion culling process, (see Chapter 3),
- a novel storage scheme, which takes advantage of the special topology of buildings and exploits real-world occlusion characteristics in urban scenes by subdividing the objects into slices parallel to the coordinate axes and allowing partial visibility to be stored in a very low amount of information, (see Chapter 4),
- an occluder-shrinking algorithm to achieve conservative visibility, which is the first demonstrated attempt that can also be applied to general nonconvex occluders, (see Chapter 4),
- a simple view-frustum culling approach, in which only one application becomes enough from a suitable culling location calculated with respect to the two eye coordinates, instead of two for the stereoscopic visualization, (see Chapter 5),
- the utilization of the GPU for the occlusion culled scenes, in the context of the developed slice-wise representation, and improved rendering performance of the urban scenes by using the GPU, (see Chapter 5).

The contributions presented in this dissertation has been published in several journals and conferences. Below is the list of publications for the contributions of the dissertation:

- T. Yılmaz, U. Güdükbay, and V. Akman."*Modeling and Visualization of Complex Geometric Environments.*", Chapter 1 in Geometric Modeling: Techniques, Applications, Systems and Tools, pages 3–30, Kluwer Academic Publishers ISBN 1-4020-1817-7, 2004.

- T. Yılmaz and U. Güdükbay. "Extraction of 3D navigation space in virtual urban environments.", In *Proc. of the 13th European Signal Processing Conference (EUSIPCO'05)*, Antalya, Turkey, 2005.
- O. Oğuz, M. E. Aran, T. Yılmaz, and U. Güdükbay. "Bina tahsis planlarından 3-boyutlu Şehir modellerinin üretilmesi ve görüntülenmesi.", In *IEEE Sinyal İşleme ve Uygulamaları Kurultayı (SIU'06)*, Antalya, Turkey, 2006.
- O. Oğuz, M. E. Aran, T. Yılmaz, and U. Güdükbay. "Automatic production and visualization of urban models from building allocation plans.", In *Proceedings of the Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'06–Technical Posters Section)*, Brazil, 2006.
- T. Yılmaz and U. Güdükbay. "Conservative occlusion culling for urban visualization using a slice-wise data structure.", doi:10.1016/j.gmod.2007.01.002, *Graphical Models (to appear)*, 2007.
- T. Yılmaz and U. Güdükbay. "GPU-based stereoscopic urban visualization", (submitted to the Visual Computer).

1.2 Outline of the Dissertation

In the next Chapter, we give the related work for the stereoscopic visualization of urban environments. In Chapter 3, we describe the navigation space extraction algorithm for urban models. In Chapter 4, we describe our slice-wise representation, and the occluder shrinking process used for determining occlusion in urban environments. In Chapter 5, the utilization of the GPU for the slice-wise representation and the contributions for the stereoscopic visualization are described. In Chapter 6, we give detailed comparisons and the results of our empirical study. Finally we conclude the dissertation in the last Chapter. We also present the City Modelling approach, which we also develop as a possible feature that can be incorporated to the framework presented in this dissertation. We describe our approach to City Modeling in Appendix A.

Chapter 2

Related Work

In this chapter, we give the related work on the subject, in terms of building modeling, navigation space extraction, occlusion culling and GPU-based stereoscopic visualization.

2.1 Building and City Modeling

One promising approach to the reconstruction of city models is the use of computer vision based techniques on aerial imagery to extract the buildings and streets [40]. Another approach is to use range scanning with the help of laser airborne scanners. There are also vehicle borne data acquisition systems with management and interactive rendering software for interactive rendering of large urban areas [18]. While these methods produce excellent city models with high accuracy, they require extra information, such as building plans and ground views, and post-processing to accurately model individual buildings in a detailed way. There are also techniques for automatic generation of high quality building models from Lidar data [82]. Hu et al. [55] give a very good survey of different approaches to large-scale urban modeling.

In order to model streets, context-free grammars, mainly L-systems, are used [34, 79]. Derivation of detailed building models using split grammars is demonstrated to be highly successful [102]. Split grammars are a composition of set grammars

and shape grammars [90]. Split grammars split or transform 3D shapes to sub-shapes that are included in the volume of the parent shape. The derivation process ends when the terminal shapes representing the building are derived. This derivation is steered by the attributes; thus specific building designs and architectures could be achieved [71]. A parameter matching system is invoked during the derivation process that allows the user to specify multiple high-level design goals and controls randomness to guarantee a consistent output. Control grammars, which are context free grammars, handle the spatial distribution of design ideas not randomly, but in an orderly way that corresponds to architectural principles.

The proposed building construction algorithm makes use of the previously developed methods, enhances them in different ways and creates an adopted version for the use in stereoscopic urban visualization framework [75, 76]. We present our approach in Appendix A.

2.2 Navigable Space Extraction

Cellulization of navigation space, thereby providing way to create visibility lists for a specific region is very crucial, because the preprocessed occlusion culling algorithms need these cells in order to calculate visibility. For walkthroughs of architectural models, cellulization is easy because rooms naturally comprehend to cells [41]. However, for walkthroughs of outdoor environments like urban sceneries, cellulization is accomplished mostly in model design time [85], by using semi-automated ways [35] or by using building footprints where the complexity of the models is limited [36, 86, 98, 100].

Generally, navigation space extraction for building interiors is not necessary, because rooms of the architectural model naturally correspond to cells, where it is not important to cellulize the rooms again as in [7, 41, 84]. In [41], the cell-to-cell visibility is defined, where a portal sequence is constructed from a cell to the others if a sight line exists, thereby making a whole cell navigable. In [100], the user is assumed to be navigating on the ground. Besides, the city model used

was built using footprints, where the navigability information becomes explicit.

Sometimes, it is quite sufficient to determine the navigable area during model design time. In [85], the developed walkthrough system accepts streets or paths as navigable, where a triangle is defined as either a street or a path triangle. This means that in order to navigate over a triangle, it must be a part of a street or a path and determined manually. Besides, only triangles are accepted for view-cells. Both of these properties make extending user navigation into the 3D space very challenging, although the algorithm for occlusion culling that the authors develop is suitable for this extension.

In [35], the user is assumed to be at two meters above streets. Besides, the created model has straight streets, making navigation space determination straightforward. Likewise, in [36, 98, 100] the authors also implement navigation assuming the user is on the ground, where the navigable space information is explicit and in 2D.

As a summary, except [31], where 3D navigation is performed using parallel computing, almost all other algorithms perform 2D navigation where extraction of navigation space is straightforward and model complexity is limited into some extend. Hence, a simple and yet powerful navigation space determination in 3D becomes vital for 3D navigation applications. The method proposed for the navigable space extraction, automatically detects and constructs the navigation space for complex urban scenes [105]. If 3D navigation is not required, the resultant navigation space structure can also be used for the navigation that is bounded to the ground.

We present our approach for the extraction of the navigable space in urban environments in Chapter 3.

2.3 Occlusion culling

Occlusion-culling algorithms detect the parts of the scene occluded by other objects and do not contribute to the overall image; these parts should not be sent to the graphics pipeline. In the special case of urban environments, most geometry

is hidden behind other buildings; occlusion culling therefore provides significant gains in performance. In addition, most of the buildings are partially visible for different views during a walkthrough. Thus, identifying occluded parts of the buildings quickly and representing partial visibility is of vital importance.

Scene representation has a crucial impact on the performance of a visibility algorithm in terms of memory requirement and processing time. Many data structures have been adopted for scene and object representation such as octrees [83], or scene graph hierarchy [77]. Scene graph usage that provides fast traversal algorithms is particularly popular [5, 89]. However, these are useful mainly for the definition of object hierarchies. Their usage in determining visibility may require them to be augmented with additional information, thereby increasing their storage requirements. In addition, the natural object structure is modified in some applications. In [84], the triangles that belong to many nodes of the octree are subdivided across the nodes for easy traversal. In [11], the objects could be divided into subobjects to create a balanced scene hierarchy, if necessary.

2.3.1 Object Space versus Image Space Algorithms

The idea of an efficient visibility culling algorithm is to calculate a conservative and fast elimination of those parts of the scene that are definitely invisible. *Object space algorithms* are the ones that geometrically make computations on the scene and decide whether the objects are visible or not, e.g. [25, 26, 27, 52, 61, 85, 92]. There is considerable work done for the visualization of urban scenes composed of 2.5D buildings –buildings constructed using their footprints. Most of them are object space methods; these iterate over the scene objects and decide whether or not they are visible [52, 61, 85]. The general approach of the previous work is to select some polygons to act as virtual occluders and check if they occlude any objects seen from the viewer by applying some sort of planar geometry. To reduce the cost of checking, occludees are usually approximated by bounding volumes.

Mostly, the target data for occlusion culling algorithms, affects the way the algorithms are designed. For building interiors or ship like scenes, most visibility algorithms decompose the model into cells [25, 41, 43, 92]. Occlusion region

can be specified by object space occlusion culling algorithm using supporting planes [27]. These cells are connected by portals and inter-cell visibility is computed, which is done in a preprocessing step. Since the walls of the buildings or doors of ships occlude a large amount of the geometry behind them, making precomputation in order to compute the potentially visible sets (PVS) and later using this information to cull the invisible objects is a novel approach for this type of data [38, 41, 43, 91, 92]. This scheme has the main disadvantage of requiring a huge secondary storage for the PVS information. There are many algorithms developed to compress the data that is needed for PVS information [9, 80, 81].

Under this classification, object space methods can be regarded as output sensitive algorithms. Output sensitive algorithms are the ones whose runtime depend only on the size of its outputs, not on the size of the inputs.

In the case of *image space algorithms*, the fundamental idea is to perform visibility computation for each frame by scan conversion of some potential occluders by checking if the projections of the bounding volumes of the occludees fall entirely within the image area covered by the occluders [10, 19, 30, 36, 46, 47, 48, 95, 98, 99, 101, 108]. Some of them classify the scene into both scene data structure and image replaceable parts, namely near and far fields. This kind of occlusion culling methods are very similar with radiosity calculations [21]. In image-based simplification methods the whole scene parts are replaced with an *impostor* –a generated image of the scene [33, 88]. Unfortunately, one impostor is usually valid for a few frames and has to be updated frequently. Other approaches use textured depth meshes that incorporate depth information for efficient impostor update. One of the important advantage of image space algorithms is that the target data can be very complex in which object space algorithms are not very successful at and the occluded objects are within a very tight estimation range. Common deficiencies of image space algorithms are that they are mostly hardware dependent and the screen resolution is fixed, which may yield rasterization errors if the resolution is increased.

2.3.2 Online versus Offline Occlusion Culling

Occlusion culling is performed either during visualization (online) or before visualization (offline). Online algorithms calculate the visibility during run-time [101]. However, the scalability is limited if no simplifying assumptions are made. To overcome this, geometry-reduction techniques such as view-dependent simplification schemes can be incorporated [7, 37].

Off-line algorithms calculate visibility with respect to a given region. This facilitates the discretization of the scene and the navigable area is divided into cells, which we call as *view-cells*. These algorithms are able to determine occlusion and store the visibility list, which is valid for a limited region. This way, the preprocessed information can be calculated and stored for later use. The occlusion power of the off-line algorithms is inversely proportional with the size of the view-cell.

2.3.3 From-point versus From-region Occlusion Culling

Occlusion culling algorithms can be classified as *from-point* and *from-region*. From-point algorithms calculate visibility with respect to the position and viewing direction of the user, whereas from-region algorithms calculate visibility, which is valid for a certain area or volume. One of the most advantageous property of the from-region algorithms is that the visibility can be precomputed and stored for later use. However, it has the disadvantage of large storage requirements, which we intend to overcome by developing the slice-wise representation.

2.3.4 Conservative, Approximate and Exact Occlusion Culling

The occlusion culling algorithms can be classified as *conservative*, *approximate* and *exact* [24]. Conservative algorithms may classify some invisible objects as visible but never call a visible building invisible. Instead of traversing an object's internal hierarchy for fine tuned visibility, most conservative algorithms either

accept the entire object as visible or reject it. These algorithms may even accept invisible buildings as visible. For urban environments, which have less hidden geometry behind the objects, occlusion culling with a few large occluders is a popular approach. The navigable area is again subdivided into cells in many approaches and for each frame a small set of (about 5-30) occluders that are likely to occlude a big part of the model is selected. The reason why these algorithms select only a small set of occluders is that it becomes very time consuming to calculate the occlusion of every occluder. The selection schemes differ among the algorithms with respect to errors introduced into the resultant image, accurateness of the selection, tightness of the conservativeness and the data that is needed to be stored with this PVS [6, 7, 35, 61].

Approximate occlusion-culling algorithms, such as [59, 61, 72], render the visible primitives up to a specified threshold, i.e., some of them may not be sent to the graphics pipeline although they are visible. There are also approaches to occlusion culling that use parallel processing methods, such as [11, 31, 100].

Another class of algorithms is the *exact* visibility algorithms, which provide accurate visibility lists at the expense of degrading the rendering performance and increasing storage requirements. An example of this class is [73], where the authors represent triangles and the stabbing lines in a 5D Euclidean space derived from a Plücker space and perform geometric subtractions of the occluded lines from the set of potential stabbing lines. In [13], the authors compute visibility from a region by using a hierarchical line space partitioning algorithm. They map the oriented 2D lines to points in dual line-space and test the visibility of a line segment with respect to the occluders yielding to a visibility from a region.

2.3.5 Environment Specific Occlusion Culling

There are occlusion culling algorithms developed for specific environments, such as indoor scenes [41], outdoor environments like urban walkthroughs [15, 32, 35, 100], and general environments –environments having no semantic object definition [7, 10, 20, 73]. In all of the algorithms the navigable area is clustered in a way to provide the fastest occlusion culling possible. For indoor scenes, the

navigation area is naturally clustered into rooms and specific techniques were developed such as portal usage [41]. For the case of urban walkthroughs, the navigable area is clustered or *cellulized* so that precomputations can be performed with respect to a limited area. Most of the algorithms developed for general environments are also applicable to others with little or no modifications such as [20], but the best performance is achieved by using the algorithms in their target environments.

Some applications are only suitable for the environments where there are large occluders and a large portion of the model is behind these occluders. These algorithms strongly rely on temporal coherence. The traversal cost and other overheads increase as the occluded regions decrease, thereby limiting the scalability [66, 84, 107]. Visibility determination by traversing a scene hierarchy requires the quick selection of occluders; or the occluders should be selected beforehand to decrease the time required for this process. Performing occluder selection is a difficult task [36, 52, 62, 85], because it must be completed in a limited time and there are many factors affecting the occluder selection process, such as the projected area of the occluder, triangle counts, transparency factors and holes. A survey of occlusion-culling can be found in [24].

2.3.6 Occluder Shrinking for From-region Occlusion Culling

Occluder shrinking is a common approach for the detection of the occluded regions in urban scenes. Using occluder shrinking, it is possible to determine occlusion from a specific point and use it for the entire view-cell region, because the occluders are shrunk by the maximum distance that a user can go in the view cell. Wonka et al. shrink occluders by using a sphere constructed around 2.5D occluders, as shown in Figure 2.1. In [32], instead of a sphere, the authors calculate erosion of the occluder using a convex shape, which is the union of the edge convex hulls of the object. These two approaches are applicable to 2.5D urban environments. Exact shrinking can only be carried out by using Minkowski differences of the view-cell and the object [4], and using the volume constructed inside the

object. In order to shrink occluders, we developed a Minkowski-difference-based method, which is able to shrink general 3D objects and use them as occluders, (see Chapter 4 and [106]).

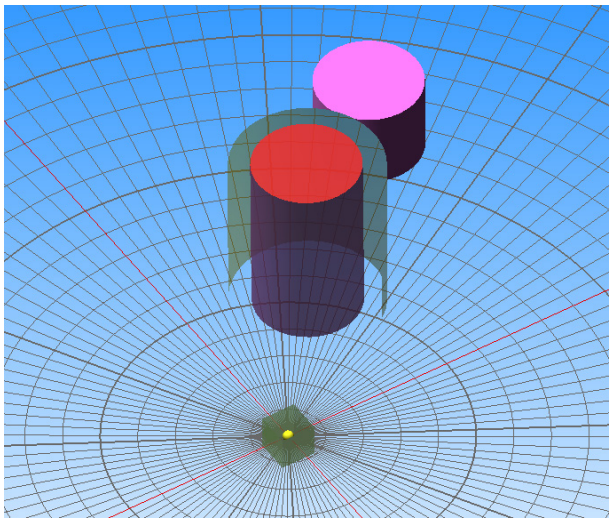


Figure 2.1: Occluder Shrinking: if the tested object (the purple one) is hidden from the shrunk version of the occluder (the red one), then it is also occluded from any point within the view cell (the green cube).

The purpose of creating visibility lists for each view-cell is to improve scalability. Time consuming operations are done beforehand. This results in a large amount of data to be stored. There are many different approaches to compressing the resultant data, such as [9, 20, 78, 80, 81]. Our slice-wise representation significantly decreases the amount of information that needs to be stored.

The proposed slice-wise structure is able to create a tight visibility set of slices of objects for any kind of occlusion-culling algorithm. The visibility set thus produced is tighter than those that measure occlusion at the building level, but more conservative than the exact ones that operate at the polygon level: it groups polygons by exploiting visibility characteristics in a typical urban walkthrough.

The monoscopic part of the proposed urban visualization framework can be compared with the previous state of the art work as follows:

- It does not make any assumption on the architectures of the buildings. Unlike [14, 15, 63, 66], our occlusion culling algorithm handles all kind of

3D occluders as in [10, 57, 72, 73], not just 2.5D buildings generated by extruding the city plans.

- The occlusion culling algorithm is based on occluder shrinking performed in the object-space. It is a from-region method, as in [14, 20, 32]; however, our algorithm is capable of shrinking all kinds of 3D objects by calculating the Minkowski difference of the occluders and the view-cell. We can shrink the nonconvex 3D occluders as a whole.
- All of the previous approaches use some kind of data structure to speed up scene and object traversal. We also use quadtree-based scheme for culling large portions of the scene. However, we make use of our proposed slice-wise structure to determine visible parts of each building to gain more rendering time by eliminating those invisible portions. Instead of traversing and storing a large amount of data for the representation of visible portions, we store only three bytes for each building and access them in constant time.
- Unlike [59, 60, 61], which are *approximate* occlusion culling algorithms and [13, 73], which are *exact* occlusion culling algorithms, our algorithm is *conservative*, like [10, 14, 15, 52, 57, 63, 66, 72, 107].
- We use hardware occlusion queries to determine occlusions, as in [20, 72, 107]. We calculate the visibility with respect to the centers of the calculated view-cells [105]. Since we use occluder shrinking, the PVS calculated for the center of the view-cell is valid throughout the whole view-cell.

2.4 GPU-based Stereoscopic Urban Visualization

In order to achieve good stereoscopic visualization, a good monoscopic correspondent must first be achieved. Therefore, we initially deal with the problem of speeding up monoscopic visualization by using powerful occlusion culling and VFC algorithms.

One of the biggest disadvantage of off-line occlusion culling algorithms is the

difficulty of storing the visibility information for run-time use, especially when the scene is large, containing tens of millions of polygons. Since visibility information must be stored for each view-cell, the number of view-cells can total hundreds of thousands. In this dissertation we present a storage scheme for buildings, called the *slice-wise representation*; this facilitates the storage of partial visibility information for urban walkthroughs [106]. It can significantly reduce the size of PVS storage when compared to other commonly used storage schemes, such as octrees. The partial visibility information can be represented with 50 % reduced polygons and 80 % speed up in frame rates when compared to occlusion culling using building level granularity. The high reduction in storage requirements for partial visibility allows the visualization of large and complex urban models.

2.4.1 GPU Usage

GPU usage is very common in today's researches. Hardware vendors provide great elasticity in order to help programmers create new algorithms. GPU usage is becoming commonplace, not only in rendering but also in performing tasks such as collision detection [45], data base sorting [44], and others [65].

A vertex buffer object (VBO) is a powerful feature that allows the user to store data in high-performance memory on the server side of OpenGL Application Programming Interface [74]. Using regular OpenGL functionality to draw primitives necessitates transferring data back and forth from the client side (CPU) to the server side (GPU). The VBO feature provides a mechanism for encapsulating the data within "buffer objects" rather than having to transfer them from the server side; this increases the rate of data transfers.

The slice-wise representation perfectly fits into the GPU architecture. This is possible by the use of VBOs and accessing the triangles to be drawn with the help of the vertex arrays constructed for the buildings.

2.4.2 Stereoscopic Visualization

2.4.2.1 About Stereoscopy

Stereoscopic visualization is used in many applications such as simulators and scientific visualizations. It uses specifically designed hardware –four frame buffers for the stereoscopic display. One of the most commonly used pieces of hardware is the time-multiplexed display system that is supported by liquid crystal shutter (LCS) glasses and virtual reality (VR) gears. Detailed information about these systems can be found in [53] and [54].

Stereoscopic viewing requires a display technique that allows each eye see the image generated for it. Most of the applications support stereoscopic display by generating the two images for the left and right eyes completely separately. The application must be able to generate 40-50 or more images per second to achieve a frame rate that approximates the same real time visualization as the monoscopic correspondent [49]. Obviously, when a monoscopic application is converted to stereo without any improvement, the frame rate decreases by half.

Most of the applications support stereoscopic display by completely generating the two images for the left and right eye views separately. Parallel processing is very suitable for this type of stereoscopic visualization. Except large-scale simulator applications such as flight simulators, there are not many applications for low-end systems, especially personal computers, that allow the user to navigate freely over the data.

2.4.2.2 Stereoscopic Image Perception

Up to 19^{th} century, mankind was not aware that there was a separable binocular depth sense. Through the ages, people like Euclid and Leonardo understood that, we see different images of the world with each eye. It was Wheatstone [97] who explained to the world that there is a depth sense named as *stereopsis*, which is produced by retinal disparity. Wheatstone explained that the mind fuses the two planar retinal images into one with stereopsis (*solid seeing*).

A stereoscopic display is an optical system, whose final component is the human brain. It functions by presenting the mind with the same kind of left and right views that the person sees in the real world [104].

2.4.2.3 Retinal Disparity

In order to explain the presence of the retinal disparity one can try this experiment: hold your finger in front of your face. When you look at your finger and try to see the finger in detail, your eyes start to converge on your finger. That is, the optical axes of both eyes cross on the finger. There are sets of muscles, which move the eyes to accomplish this by placing the images of the finger on each fovea, or central portion of each retina. If you continue to converge your eyes on your finger, paying attention to the background, you will notice that the background appears to be doubled. Now try to focus on the background and you will see that when your see the background in detail, your finger, because of the retinal disparity, will now appear to be doubled. If we could take the images that are on your left and right retina and somehow superimpose them as if they were aside, you would see two *almost* overlapping images –left and right perspective viewpoints–, which is what physiologists call disparity. Disparity is the distance, in horizontal direction, between the corresponding left and right image points of the superimposed retinal images. The corresponding points of the retinal images of an object on which the eyes are converged, will have zero disparity.

Retinal disparity is caused by the fact that each of our eyes sees the world from a different point of view. On the average the eyes are two and a half inches or 64 millimeters apart for adults [22]. The disparity is fused by the brain into a single image of the visual world. The minds ability to combine two different, although similar, images into one image is called fusion, and the resultant sense of depth is called stereopsis.

2.4.2.4 Parallax

A stereoscopic display is able to display parallax values, which is the distance between left and right corresponding image points and may be measured in inches

or millimeters. This makes stereoscopic display different from a monoscopic display. Disparity in the eyes produces parallax, and this provides the stereoscopic cue.

Electro-stereoscopic displays provide parallax information to the eye by using a method related to that employed in the stereoscope. In a stereoscopic display, the left and right images are alternated rapidly on the monitor screen. When the viewer looks at the screen through shuttering eye-wear, each shutter is synchronized to occlude the unwanted image and transmit the wanted image. Thus each eye sees only its appropriate perspective view. If the images (the term *fields* is often used for video and computer graphics) are refreshed fast enough (often at twice the rate of the monoscopic display), the result is a flickerless stereoscopic image. This kind of a display is called a *field-sequential stereoscopic display.*

When you observe an electro-stereoscopic image without eye-wear, it looks like there are two images overlayed and superimposed. The refresh rate is so high that you cannot see any flicker, and it looks like the images are double-exposed.

Parallax and disparity are similar entities. Parallax is measured at the display screen and disparity is measured at the retinal. When wearing eye-wear, parallax becomes retinal disparity. Retinal disparity produces parallax, and parallax in turn produces stereopsis. Parallax may also be given in terms of angular measure, which relates it to disparity by taking into account the viewers distance from the display screen. Since parallax is the entity that produces the stereoscopic depth sensation, we give a classification of the kinds of parallax one may encounter in stereoscopic viewing.

2.4.2.5 Types of Parallax

Four basic types of parallax are defined [22]: zero parallax, positive parallax, divergent parallax and negative parallax. In *zero parallax*, the homologous image points of the two images exactly correspond or lie on top of each other. The eyes of the observer is separated with the interpupillary or interocular distance (IOD) that is on the average two and a half inches. When the observer is looking at the display screen and observing images with zero parallax, this means that the eyes

are converged at the *surface* of the screen. In other words, the optical axes of the eyes cross at the plane of the screen. In *positive parallax*, the axes of the left and right eyes are parallel. This happens in the visual world when looking at objects that are at a great distance from the observer. For a stereoscopic display, when the distance between the eyes (*IOD*) equals the parallax, the axes of the eyes will be parallel, just as they are when looking at a distant object in the visual world. Experiences show that having parallax values equal to IOD, or nearly IOD, for a small screen display will produce discomfort [22]. The visualization with an uncrossed or positive value of parallax between IOD and zero, will produce images appearing to be within the space of the *cathode ray tube* (CRT), or behind the screen.

Another kind of parallax is *divergent parallax*, in which images are separated by some distance greater than IOD. In this case, the axes of the eyes are diverging. This divergence does not occur when looking at objects in the visual world, and the unusual muscular effort needed to fuse such images may cause discomfort. There is no valid reason for divergence in computer-generated stereoscopic images. Objects with *negative parallax*, appear to be closer than the plane of the screen, or between the observer and the screen. The objects with negative parallax are said to be within viewer space [104].

2.4.2.6 Focusing and Convergence Relationship

The left and right image fields must be identical in every way except for the values of horizontal parallax. The color, geometry, and brightness of the left and right image fields need to be the same or to within a very tight tolerance, or the result will be *eye fatigue* for the viewer. If a system is producing image fields that are not suitable in these respects, it will never be able to produce good-quality stereoscopic images. Left and right image fields congruent in all aspects except horizontal parallax are required to avoid discomfort [68].

The eyes converge dynamically on the objects in the real world, depending on the distance of the objects. However in stereoscopic visualization, it is assumed

that the eyes converge on the screen, not on any specific object and this convergence does not show up any change. This differentiation of real world and stereoscopic visualization causes some people depart from their natural feeling and they may experience an unpleasant sensation when looking at stereoscopic images, especially images with large values of parallax. Experiences show that it is better to use the lowest values of parallax possible for a good depth effect in order to help to reduce viewer discomfort. On the other hand, the parallax value specification and visual discomfort should be adjusted so that the visual discomfort is minimized, while providing a good depth effect.

The goal when creating stereoscopic images is to provide the deepest effect with the lowest values of parallax. This is accomplished in part by reducing the IOD. As a rule, parallax values should not exceed 1.6° [93]. Also the distance of the viewer from the screen should be taken into account when composing a stereoscopic image.

2.4.2.7 Crosstalk (Ghosting)

Main problems incurred with stereoscopic visualization include the ghosting effect and the resultant eye disturbance problems. The ghosting effect or crosstalk in a stereoscopic display results in each eye see an image of the unwanted perspective view. It is the faded image seen by the untargeted eye. This effect is undesirable because it may cause eye fatigue and other visualization problems. Much research is devoted on reducing this disturbing effect. In a perfect stereoscopic system, each eye sees only its assigned image. In particular, there are two reasons for crosstalk in an electronic stereoscopic display: late decaying of the phosphor (afterglow), and shutter leakage [17, 50, 67, 69, 70]. The phosphor persistence causes a faded image to be seen when the image for the other eye is being displayed on the screen [103]. A third reason of ghosting is non-matching perspective projection for both eyes [104]. This may occur when a point is projected for an eye but not projected for the other.

In an ideal field-sequential stereoscopic display, the image of each field, made up of glowing phosphor, would vanish before the next field was written, but that is

not what happens. After the right image is written, it will persist while the left image is being written. This phosphor persistence results in one image to last in time. Late decaying of it while switching the eyes causes a faded image be seen, when the image for the other eye is being displayed on the screen [103]. Thus, an unwanted fading right image will persist in the left image (and vice versa). The term ghosting is used to describe perceived crosstalk. Stereoscopists have also used the term *leakage* to describe this phenomenon. The perception of ghosting varies with the brightness of the image, color, and parallax and image contrast. This effect is especially experienced when the background color is dark and the image just drawn has high intensity colors. Images with large values of parallax will have more ghosting than images with low parallax. High-contrast images, like black lines on a white background, will show the most ghosting. Given the present state of the art of monitors and their display tubes, the green colored phosphor has the longest afterglow and produces the most ghosting effect [103].

2.4.2.8 Speeding-up Stereoscopic Visualization

Earlier work on speeding up stereoscopic rendering generally made use both of the mathematical characteristics of an image that change when the eye-point shifts horizontally, and a recognition of the characteristics that are invariant with respect to eye-point, such as the scan-lines toward which an object projects [53]. In [39], the authors present a sterescopic raytracing algorithm that infers a right-eye view from a fully ray-traced left-eye view; this algorithm is further improved in [3]. In [1], a non-ray-tracing algorithm is described that speeds up second-eye image generation in the processes of polygon filling, hidden surface elimination and clipping. Methods that take advantage of the coherence between the two halves of a stereo pair for ray traced volume rendering are discussed in [2]. In [51], the authors present an algorithm using segment composition and linearly-interpolated re-projection for fast direct volume rendering. Hubbold et al. [56] propose extending a direct volume renderer for use with an autostereoscopic display in radiotherapy planning. In [49], the authors present a framework to speed up stereoscopic visualization of terrains represented as height fields by generating the view for one eye from the other with some modifications; this speeds

the process up by approximately 45 %, as compared to generating two eye views separately from scratch.

2.4.2.9 Other Problems with Stereo

Resolving occlusion in stereoscopic imagery is known as stereo matching problem and an important issue. Occlusion regions in stereoscopic views are spatially coherent groups of pixels that appear in one image and not in the other. These occlusion regions are caused by occluders, in which there is a very little information for the occluded part, when seen from the occluded eye direction. In [16], stereo matching problem is tried to be solved for still images. There are also many other research done for stereo matching that uses image processing methods, like [12, 28, 42, 58] all of which work on image processing methods, which are out of our consideration.

Chapter 3

Navigable Space Extraction

In order to develop navigation systems for urban sceneries, extraction and cellulization of navigable space is one of the most commonly used technique providing a suitable structure for visibility computations. Cells for the navigable area are needed, because the precomputations for the visibility are valid only for a specific area and these areas, called *view-cells* should be determined beforehand. Urban models, except for the ones where the building footprints are used to generate the model, generally lack navigable space information. Because of this, it is hard to extract and discretize the navigable area for complex urban scenery.

Urban visualization strongly requires culling of unnecessary data in order to navigate through the scene at interactive frame rates. There are efficient algorithms for view-frustum culling and back-face culling. However, occlusion culling algorithms are still very costly. Especially, object-space occlusion culling algorithms strongly need precomputation of the visibility for each view-point and for each viewing direction.

Almost all occlusion culling algorithms calculate occlusion with respect to ground walks, thereby eliminating the need for a 3D navigable space. However, for a general fly-through application, a cellulized navigable space can provide a suitable environment for a precomputable visibility information.

The algorithm presented in this chapter, calculates and extracts the navigable space for urban scenery, where the models of buildings are highly complex. The

buildings may have balconies, pillars, fences or holes where it is possible to see through them. No assumptions or restrictions are made on the model. The extracted navigable space looks like a jaggy sculpture mold and it is used in the cellulization process required by the occlusion culling algorithms. Besides, for the urban data acquired from different sources, which may contain errors, our approach provides a simple and efficient way of discretizing both navigable space and the model itself. The extracted space can instantly be used for visibility calculations such as occlusion culling in 3D space. Furthermore, terrain height field information can be extracted from the resultant structure, hence providing a way to implement urban navigation systems including terrains.

Current occlusion culling algorithms, which use preprocessing for occlusion determination, need large amount of data to store the visibility lists for each viewpoint. One of the most promising result of our navigable space extraction method is that, it becomes suitable to develop other general structures, which yields natural occlusion determination for urban scenes and decrease drastically the amount of the data that is needed to be stored.

3.1 Navigable Space Extraction Algorithm

Figure 3.1 shows the data structures used in the navigable space extraction process; these include the structures to represent the objects in a scene and the structure to store the navigable space. `geomobject` structure stores the name of the object and the number of triangles making up of this object. It holds pointers to the bounding box of the object, the very first triangle of the triangle list, the parent of the octree defining the navigable space found within the box of the object and next object in the order. The scene file is read from the storage and a list of triangles are tied to each object with necessary vertex information defined in `tri` and `vertex` structures. The `octree` and `seed` structures are used later, while extracting the navigable area and discretized objects.

```
struct vertex {
    float x;
    float y;
    float z;
};
struct tri
{
    struct colors color;
    struct vertex v1;
    struct vertex v2;
    struct vertex v3;
    struct vertex normal;
    struct tri *next;
};
struct octnode
{
    int level;
    char no;
    float minx, miny, minz, maxx, maxy, maxz;
    char type;     //1:parent, 2:inner, 3:leaf
    struct octnode * parent;
    struct octnode * n[8];
    char empty;
};
struct geomobject
{
    char name[12];
    int number_of_triangles;
    struct boundingbox * b_box;
    struct tri * first;
    struct octnode *octtree;
    struct geom_obj* next;
};
struct seed
{
    int xoff,yoff,zoff;
    int tag_fill;
};
```

Figure 3.1: The data structures used in the navigable space extraction.

3.1.1 Extraction Process

We need to mention that the input data formats do not have significant importance on the efficiency of the algorithm, because our approach is nearly independent of the input data format. The only assumption is that the scene must be composed of triangles. One of the most common data format is the *dxf* data format created by *Autodesk, Inc.* The data structure used to store this file is a forest type data structure, equipped with suitable fields designating the parameters of the other algorithms.

The navigable space extraction algorithm mainly consists of two phases: the seed test, and the contraction and the octree construction phase. In the first phase, the bounding boxes of objects are calculated and a seed box is travelled around each object to find the blocks that touch its surface. Filled seeds are later passed to a contraction algorithm, in which the octree structure for the navigable area is constructed and the mold of the object becomes extracted. It should be noted that, it is possible to find all holes and passages inside the objects within a user specified threshold using this approach. The flow diagram of our algorithm is shown in Figure 3.2.

After reading the scene database from the input file, the algorithm first calculates the bounding boxes of each object in the scene. Object discrimination is done while constructing the scene file and each object (i.e., building) is defined with a header and triangles are inserted into the list according to the object names, which is a property of the `dxf` file format. The bounding boxes are calculated in a straightforward manner and stored in the relevant structures. Seed testing and contraction parts of the algorithm take place in these bounding boxes and all space out of these boxes are accepted to be navigable.

3.1.2 Seed Testing

The seed testing phase is based on a box with a size of a user-defined threshold. We call this size as *threshold* because it defines the roughness of the extracted mold of the object. The time needed to extract the navigable area strictly depends

on the size of the seed box.

We start by reading the scene data. The next thing to do is to calculate the bounding boxes of each object in the scene. The object discretization algorithm is based on grid cells with a user-defined size threshold. This threshold defines the roughness of the extracted mold of the object. The algorithm travels inside the bounding box of the object to find the occupied grid cells. A grid cell and a triangle may intersect in three ways, which are shown in Figure 3.3.

The first case is where any vertex (or vertices) of the triangle is inside the cell. This case is the easiest to determine, in which a range test gives the intended result (Figure 3.3 (a)). The second case, none of the vertices of the triangle is inside the cell but the triangle plane intersects the edges of the cell, is handled by performing ray plane intersection test (Figure 3.3 (b)). In the algorithm to detect this case, the main idea is to shoot rays from each corner of the cell to each coordinate axis direction. The last case (Figure 3.3 (c)), where the triangle penetrates the cell without touching any of its edges is handled in a similar way, but this time the rays are shot from the vertices of the triangle and checks are made against the surfaces of the cell. This process is repeated until all locations in the bounding box of the object is tested. A sample discretization for an object in 2D is shown in Figure 3.4 (a). With this approach, it is possible to use all holes and passages through the objects as part of the navigable area (see Figure 3.4 (b)).

The discretization of the object structure by testing each unit cube with the triangle structure (See Figure 3.3) is essential with respect to two aspects: one is the definition of the object hierarchy, and the other is creating an object structure, which is an alternative to current octree-like structure.

3.1.3 Extraction of the Navigable Space

Although the uniform subdivision provides the occupied cell information, which is enough to determine the navigable space, its memory requirement is high. In order to overcome this problem, an adaptive subdivision is applied to the bounding box of the object to extract the navigable area as an octree structure. This is done using the occupied cell information provided by the uniform subdivision.

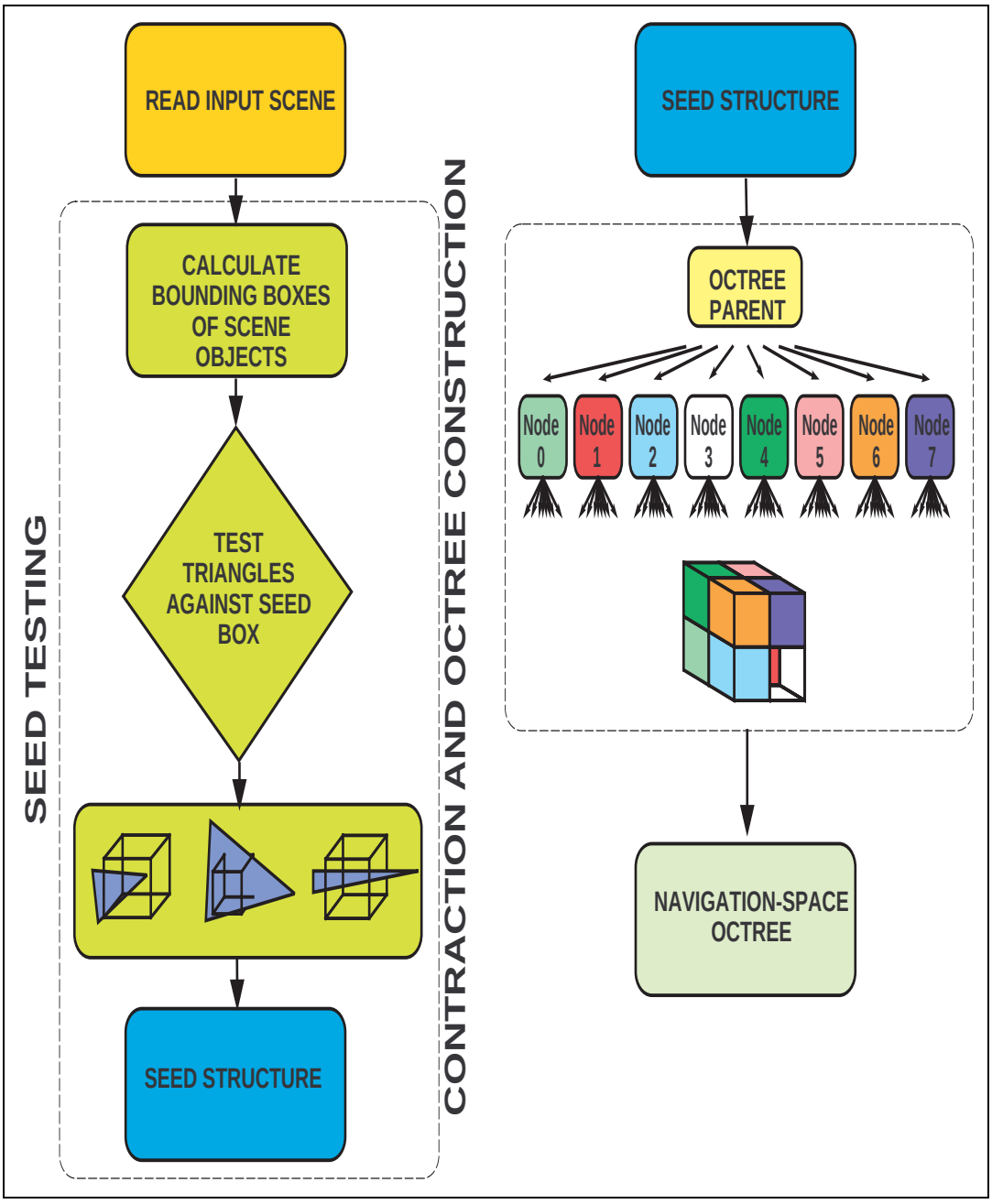


Figure 3.2: Flow diagram of the navigable space extraction algorithm.

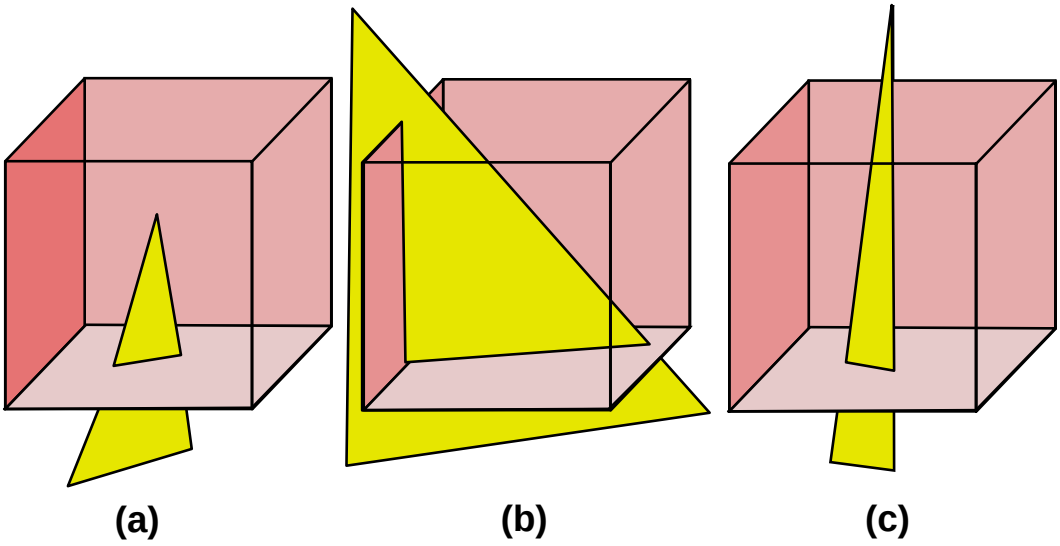
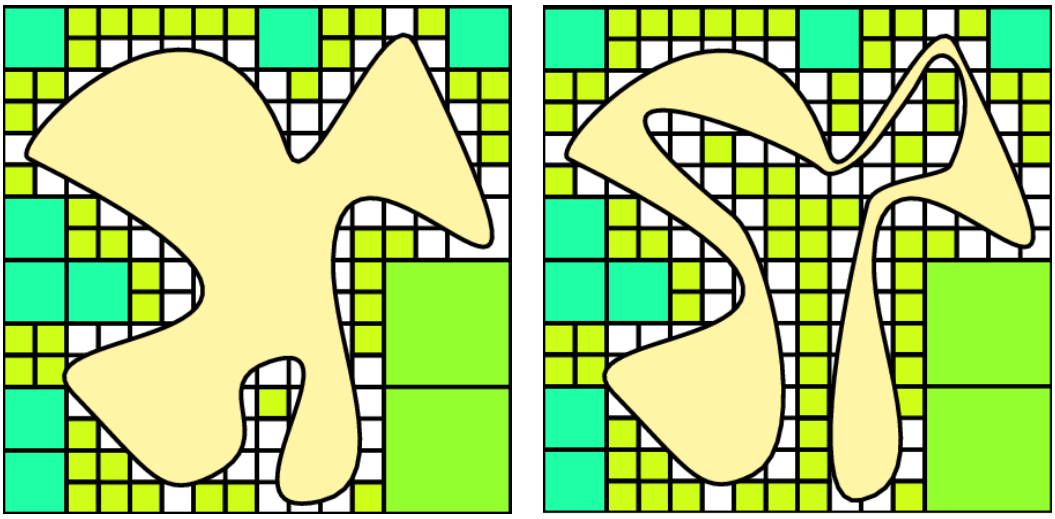


Figure 3.3: Test cases: (a) any vertex is inside the cell; (b) the vertices of the triangle is not inside the cell, but the cell edges intersect with the triangle surface (See Algorithm 3.1); (c) the triangle edges intersect with the surfaces of the cell. The idea behind this testing is to determine each unit cube, which has an interaction with at least one triangle. This will help us to create the slice-wise representation, which is specifically designed for urban scene occlusion culling.

An example of the created structure is shown in Figures 3.5 and 3.6.

The navigation octrees for each object are tied up to the spatial forest of octrees that corresponds to the whole scene. The empty area outside the objects in the scene is also a part of the navigable space.

We did not make any assumptions on the type of scene objects, or on their respective locations, while determining the navigable space information. The objects may have any type of architectural property, such as pillars, holes, balconies etc. Our algorithm indiscriminately finds the locations not occupied by any object part (i.e., triangle). This property makes our approach very suitable for the models that are created from different sources such as *LIDAR*, because the only information needed is triangle information, which most model formats have, or otherwise the primitives that are convertible to them.



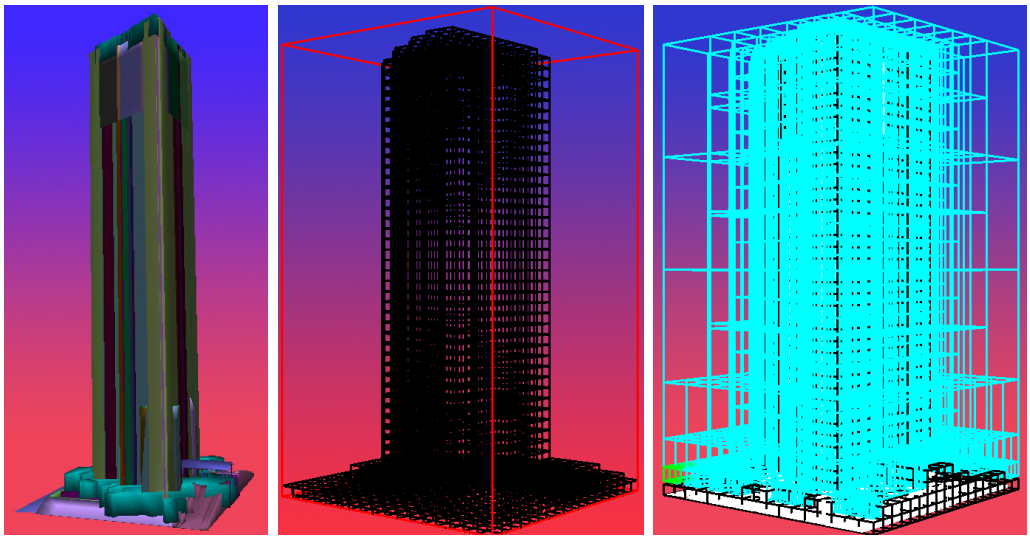
(a) (b)

Figure 3.4: (a) Discretization of an object in 2D for easy interpretation. It is normally performed in 3D. The object is represented with uniform grid cells (filled boxes show the complement of the object space, which corresponds to the navigable area represented as an octree using adaptive subdivision to reduce memory requirements.) (b) Any holes and passages can safely be represented as part of the navigable area.

3.1.4 Contraction and Navigable Space Octree Construction

After the seed test phase is finished, we have a discretized version of the scene objects, where we exactly know the spatial locations occupied by the triangles of the object within a certain threshold. Although the occupied seed cell information is enough for us to determine the navigable space (i.e. its dual space), its memory requirement is very high. Therefore, we need to contract this empty area and determine the navigable space using another structure requiring less memory space. An octree structure is used for this purpose.

The octree structure constructed is shown in Figure 3.2. The algorithm for the octree construction simultaneously contracts the space not occupied by the seed cells into larger blocks of space as much as possible, thereby eliminating the need to keep filled cell information for every small seed. This is done as follows: The algorithm first sets the bounding box of the object as the parent of the navigable



(a) (b) (c)

Figure 3.5: Navigation octree construction: (a) the original object; (b) cells of the object, where its triangles pass through; (c) the created navigation octree, in which navigable space information is embedded. In the figure, the black lines show the occupied cells and the green lines show the boundaries of the navigation octree for the object.

space octree and checks if there is any filled cell within the range of the node. If there is any filled cell, then it recursively subdivides itself into eight octants and repeats the same procedure for the newly created nodes until the size of the node decreases below the size of the seed box or there is nothing left but empty cells. This structure is tied to the spatial forest of octrees after all the scene objects are processed. It should be noted that the numbering scheme as seen in the Figure 3.2, provides neighboring information of the octree nodes. The constructed octree allows the navigable space to be traversed hierarchically.

3.1.5 Resultant Structure

The algorithm is concluded after all the scene objects are processed. The resultant octree structure represents the navigable area, where bounding boxes are tied up to spatial forest of octrees. An example of the created structure is shown

```
for each triangle of the object do
    define plane of triangle;
    for each corner of the cell do
        shoot rays towards the other neighboring corner;
        if the ray hits the plane of triangle then
            calculate the intersection point;
            translate the point to the origin;
            for each edge of the cell do
                if the horizontal line to the right of the intersection point
                intersects the triangle odd number of times then
                    report as INSIDE;
                else
                    report as OUTSIDE;
    for each neighboring corners of the cell do
        if any two neighboring corners have intersection on the triangle then
            return INTERSECT;
    return NOTINTERSECT;
```

Algorithm 3.1: The pseudo code for the detection of the second case, where a triangle passes through a cell but none of the vertices of the triangle is inside the cell and the triangle plane intersects with the cell edges.

in Figure 3.5. After this, the user exactly knows the locations in 3D, where navigation is possible. The user will also know where the objects are and a hierarchical subdivision of them will also be provided, as described below.

3.2 Creating Object Structure

In addition to creating the navigable space information of the scene database, it is very easy to create the octree for the scene itself, where further calculations on them can be performed. One important process that can be applied to the hierarchy is occlusion determination, where hierarchical calculations are strongly needed.

After the contraction process is performed and the octree for navigable space constructed, the octree construction algorithm is repeated once more seeking full seed cells to discretize the object. This time the contraction part of the algorithm

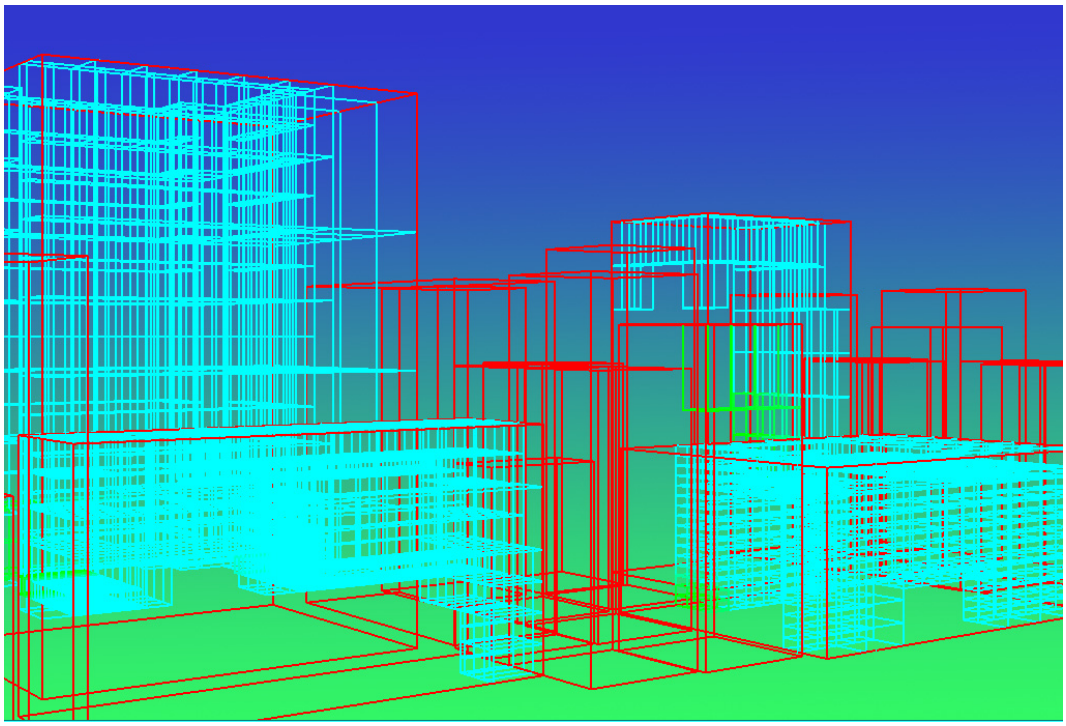


Figure 3.6: The octree forest of a scene created by the navigable space extraction algorithm having 24 buildings.

finds the cells which contain geometry in it, whereas we did it for empty cells while constructing the octree for the navigable space. The same procedure is applied to each object and scene objects are tied to the spatial forest of object octrees. We are left with the two forests of octrees, one with the navigable space information and one with the object hierarchy, which are useful for 3D navigation and scene object processing, respectively.

Chapter 4

Occlusion Culling using Slice-wise Representation

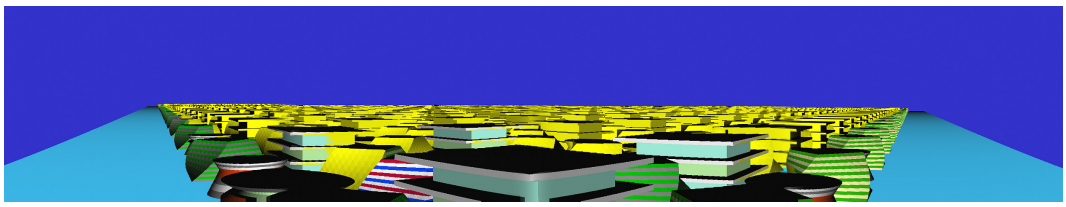


Figure 4.1: Slice-wise occlusion culling sends approximately 51 % fewer triangles to the graphics pipeline and increases frame rate by 81 %, as compared to occlusion culling using building-level granularity for this model. The yellow colored sections show occluded regions, which are discarded from the graphics pipeline. In addition, the slice-wise representation decreases Potentially Visible Set (PVS) storage requirement drastically.

The efficiency of a visibility algorithm is vitally important for making an urban visualization system usable on ordinary hardware. View-frustum culling and back-face culling are ways to speed-up the visualization and there exist efficient methods for them. However, occlusion-culling algorithms are still very costly.

In occlusion-culling algorithms, where the granularity is individual buildings, an object could be sent to the graphics pipeline even if a small portion of it becomes visible. In most cases, this would result in unnecessary overloading of the hardware, especially if the objects are very complex. An efficient approach is needed to create a tight visibility set without causing further overheads. Although it is

feasible to traverse the nodes in the hierarchy of an object to see which parts are visible, it is usually impractical to store the visibility lists.

In this chapter, we present the *slice-wise representation*. This is a simple storage scheme, which takes advantage of the special topology of buildings within an urban scene. It automatically exploits real-world occlusion characteristics in urban scenes by subdividing the objects into slices parallel to the coordinate axes (Figure 4.1). Other object hierarchies such as octrees and regular grids can well be used to partition the objects; even individual triangles can be checked. However, the storage requirement of Potentially Visible Sets (PVSs) limits the scalability of their usage. The PVS storage requirement of the proposed slice-wise structure is very low (three bytes for each viewpoint and partially visible building). An index is stored for a partially visible building, indicating the visible slices along each coordinate axis.

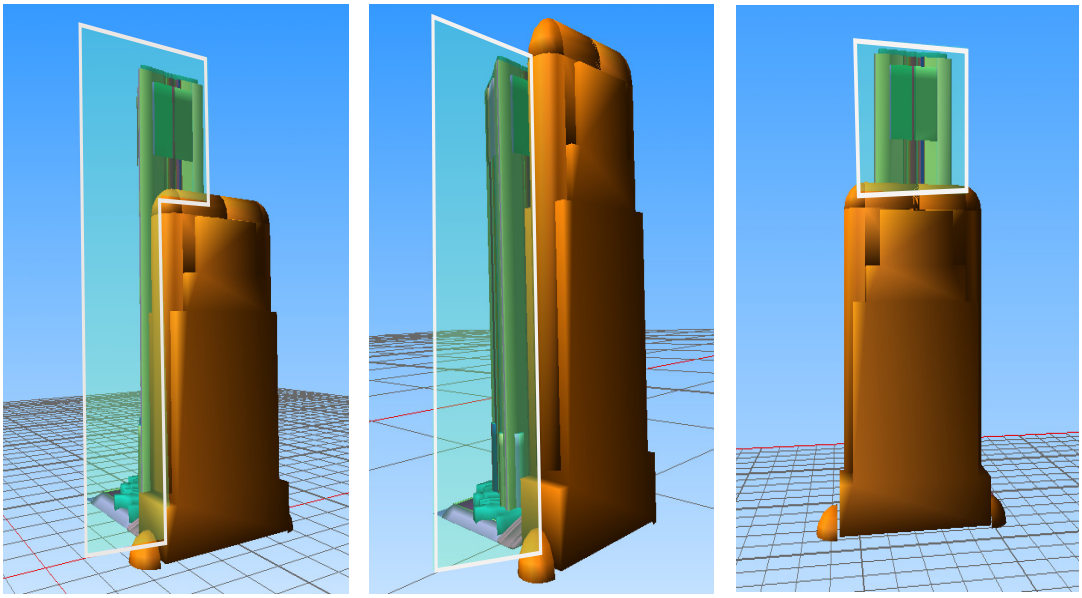
In this chapter, we also present an occluder-shrinking algorithm to achieve conservative from-region visibility. Conservative occlusion-culling can be performed by shrinking the occluders and performing the visibility tests using the shrunk versions of the occluders. To our knowledge, this is the first demonstrated attempt that can also be applied to general nonconvex occluders as a whole.

4.1 Slice-wise Object Representation

4.1.1 Object Visibility Forms

Our slice-wise approach is based on the observation that while a person is navigating through a city, the visible parts of the objects usually have one of the following three forms (see Figure 4.2):

- The visible part looks like an *L-shaped* block in different orientations if a building is occluded in part by a smaller occluder, as in Figure 4.2 (a).
- The visible part looks like a *vertical rectangular* block, from the left or right of the building if the occluder seems taller than the occludee (see Figure 4.2 (b)).



(a) (b) (c)

Figure 4.2: Visibility forms during urban navigation: (a) *L-shaped* form; (b) *vertical rectangular* form; (c) *horizontal rectangular* form. In each part of the figure, the visible part of the occludee is the green transparent area.

- If the occluder is a large one and appears to be shorter than the occludee, it usually hides the lower half of the building. In this case the visible portion looks like a *horizontal rectangular* block, as in Figure 4.2 (c).

Most of the occlusion can be represented by the proposed slice-wise structure using these forms. However, there are of course some other cases that the occlusion cannot be perfectly represented. These may include a configuration such as both sides of the building are occluded resulting in a middle part visibility and the top of the building is visible in addition to the middle part visibility. In any of these cases the occlusion culling algorithm tries to capture the occlusion as much as possible in one of the three visibility forms. For example, the middle part visibility is regarded as *vertical rectangular*, the top and middle part visibility is regarded as *L-shaped* visibility.

Obviously, a visibility-culling algorithm could be developed without characterizing visible parts of the buildings. However, this might send unnecessarily large

number of polygons to the graphics pipeline. If we could find a way to exploit these visibility characteristics, we could reduce both the number of polygons sent to the graphics pipeline and the storage requirements for the PVSs. This is what we achieve with the proposed slice-wise structure.

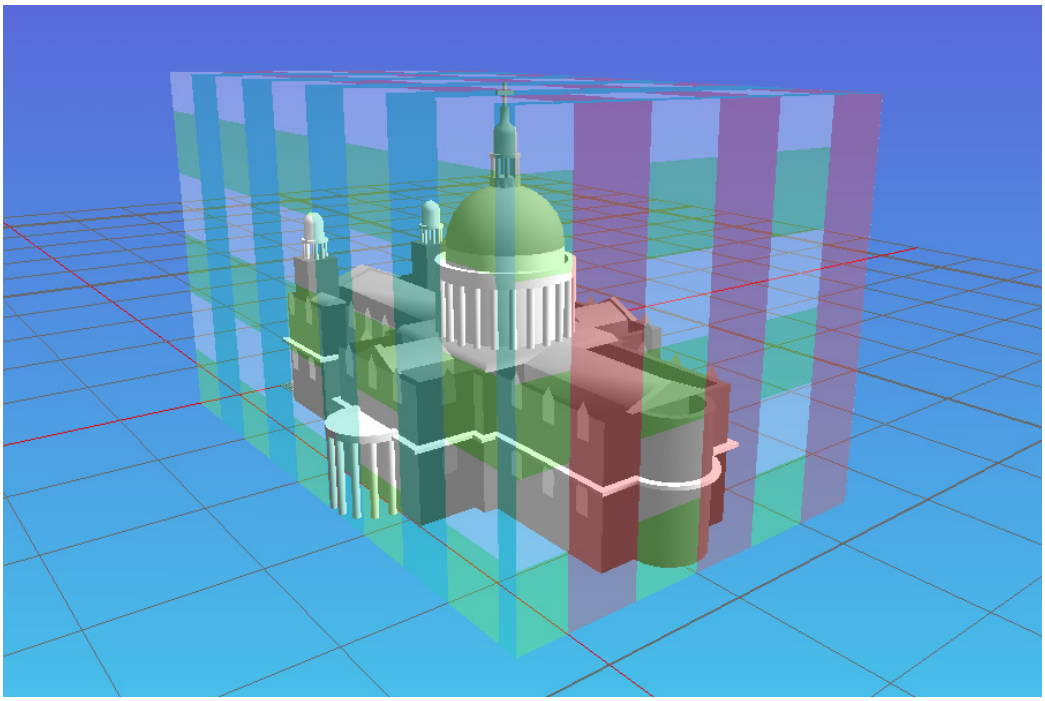
4.1.2 Slicing Objects

The aim of the proposed slice-wise structure is to create tight PVSs for urban scenes. Slice-wise representation is obtained by subdividing an object into axis-aligned slices and determining the triangles that belong to each slice. The slicing process is composed of two steps: *subdivision* and *slice creation*. In the subdivision step, each object is uniformly subdivided and the grid cells occupied by each triangle are determined using the approach presented in Chapter 3. Next, the occupied cells in the uniform subdivision are combined into axis-aligned slices for each coordinate axis. The process of slicing an object is shown in Figure 4.3. The resultant data structure is shown in Figure 4.4.

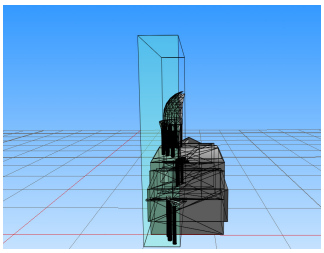
4.1.3 Visibility Representation Using Slices

Defining the visible portions requires determining the visible slices. As shown in Figure 4.5, we only need to store one visible-slice index for each axis. The combination of these indices facilitates the representation of the visibility characteristics. The index number depends on the occluded sections of the occludee. For vertical slices, if the right part of the object is occluded, then the index is stored with a "+" sign indicating that the slices, numbered from left to right, will be accepted as *visible* including the slice with the index number. For the other case, the index is stored with a "−" sign, telling that the slices are *invisible* including the slice with the index number. In the case of horizontal slices, the numbers are assigned from bottom to top and the first visible slice number is stored as the index. Visible-slice indices are assigned to each axis after they are checked for occlusion.

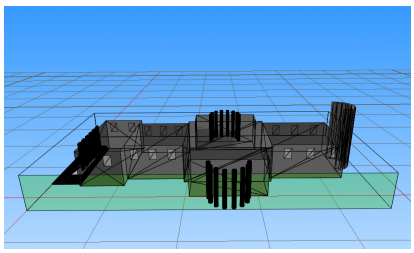
In addition to facilitating the exploitation of different visibility characteristics for



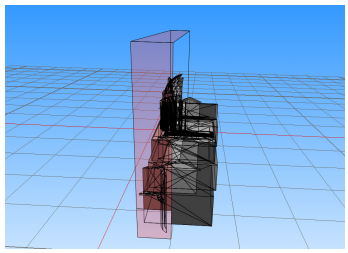
(a)



(b)



(c)



(d)

Figure 4.3: The process of slicing an object determines the triangles that belong to each slice. (a) a complete view of the object where the positions of slices are shown; (b) an x–axis slice;(c) a y–axis slice; (d) a z–axis slice.

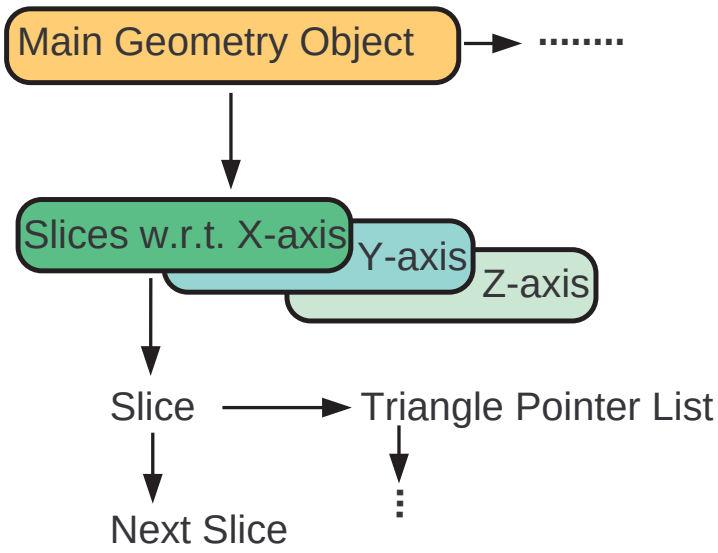


Figure 4.4: The scene data structure produced by slicing operations.

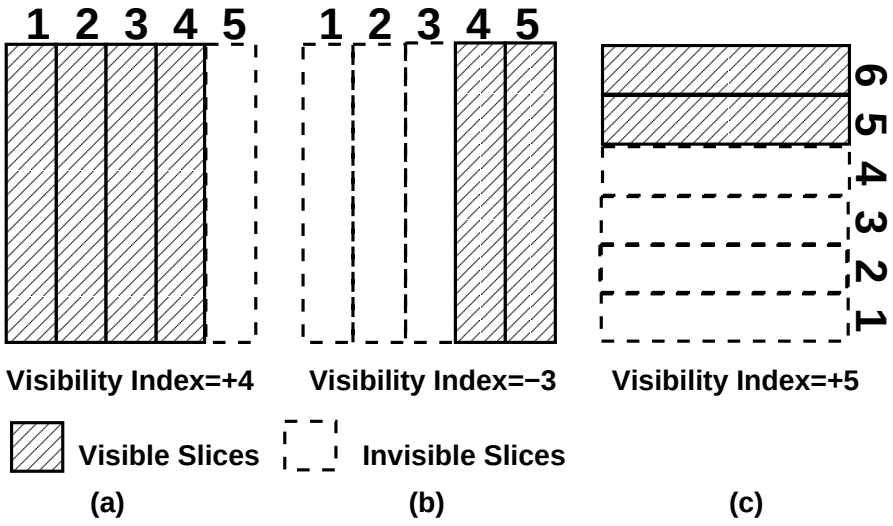


Figure 4.5: Defining visibility indices for objects: the visible slice indices are determined for each axis during occlusion determination. (a) If an object is partially occluded from the right, the index of the last visible slice is stored with a "+" sign. (b) If the object is partially occluded from the left, the index of the last invisible slice is stored with a "−" sign. (c) If the object is partially occluded from the bottom, we store the index of the first visible slice.

tight visibility processing, the benefits of slicing objects are:

- Each triangle is encoded in at least 3 slices in different axes. Therefore, we can use slices on any axis during visualization. We choose the axis with maximum occlusion. Choosing the maximally occluded axis allows us to tighten the visible set as much as possible.
- The memory required for the slice-wise approach is minimal. In order to define the visibility, three bytes, one for each axis, are used for each object and for each view-cell. This representation greatly decreases the storage requirements for PVSs.
- Slicing the objects provides a fast way to access visible portions of an object. Unnecessarily traversing a tree-like data structure is prevented by directly accessing the visible slices and hence triangles of an object.

4.1.4 Comparison with Other Storage Schemes

For precomputed visibility, the size of the data stored for the view-cells may become so large that the total size of the PVSs is much larger than the size of the scene. Aside from a few studies [20, 43, 78], the problem of big PVS storage problem has not been given enough importance [24].

We compare the proposed structure with octrees and regular grids in terms of the memory requirements. In Figure 4.6 and the corresponding Table 4.1, we depict subdivision depth and the number of nodes needed for each subdivision. The number of nodes for octrees refers to regularly subdivided octree including the bits needed for the previous levels. In an adaptively subdivided octree, the number of nodes is below these levels. However, giving exact costs and approximations on adaptive version is very difficult. Instead, we give an informal comparison of the results of the empirical study with octrees and triangle level PVS sizes in Chapter 6. A comprehensive study of the costs for various construction schemes of octrees is presented in [8].

PVS storage costs are depicted for various storage schemes in Figure 4.7 and the corresponding Table 4.2. In this comparison, we assume that all objects are *visible*

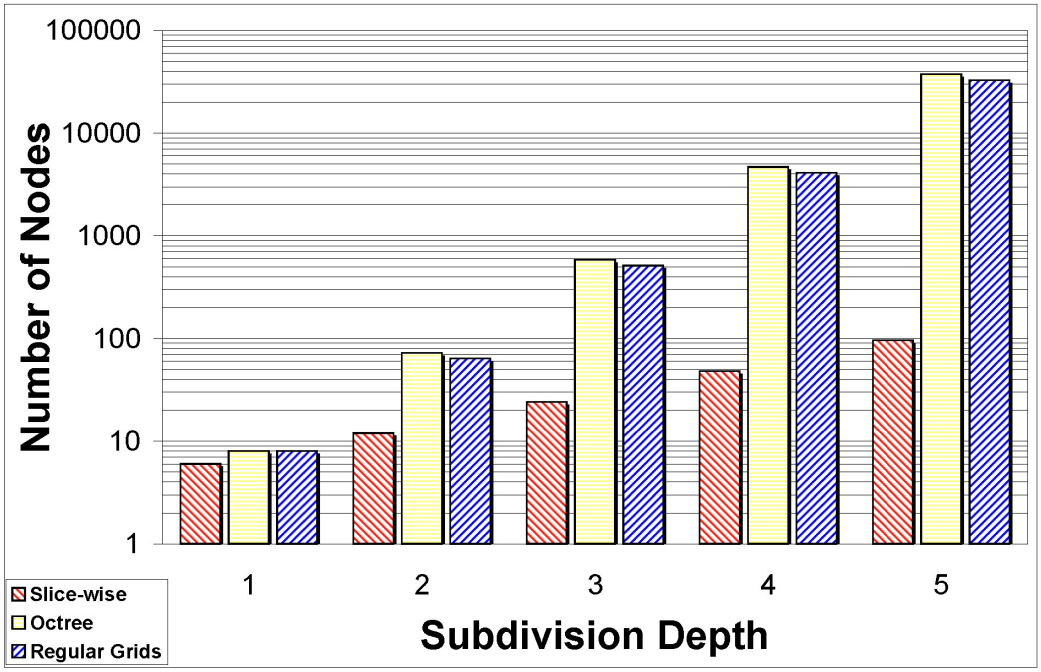


Figure 4.6: The comparison of the number of nodes needed for each subdivision scheme. The values are in logarithmic scale.

Table 4.1: The comparison of the number of nodes needed in slice-wise, octree and regular grids.

Depth	Slice-wise	Octree	Regular Grids
1	3x2=6	8	8
2	3x4=12	64+8=72	64
3	3x8=24	512+72=584	512
4	3x16=48	4,096+584=4,680	4,096
5	3x32=96	32,768+4,680=37,448	32,768

Table 4.2: PVS storage comparison (Bytes/View-Cell)

Triangles	Objects	Depth	Slice-wise	Octree	Regular Grids	Triangle Level
100K	10	1	30	80	80	12.5K
		2		720	640	
		3		5,840	5,120	
		4		46,800	40,960	
		5		374,480	327,680	
1M	100	1	300	800	800	125K
		2		7,200	6,400	
		3		58,400	51,200	
		4		468,000	409,600	
		5		3,744,800	3,276,800	
10M	1000	1	3K	8K	8K	1,250K
		2		72K	64K	
		3		584K	512K	
		4		4,680K	4,096K	
		5		37,448K	32,768K	

or *partially visible*. We also assume that each node of octree and regular grids can be identified with 1 bit and we discard additional information to be stored along them, such as corner coordinates. The pointers to polygons are not taken into account because they are needed in all types of structures. Additionally we provide the data needed to do occlusion culling at the polygon level, assuming that the visibility of each triangle is encoded in bits and each object is composed of 10K polygons. However, since the scene size is indicated by the number of visible triangles and objects, the data needed to be stored for polygon-level occlusion culling may be much larger than those given in the figure and using an additional data structure becomes necessary. The figure shows that the slice-wise structure requires much less space to store the PVSs; this is an indispensable part of most preprocessed occlusion-culling algorithms. Other compression schemes may be used to further decrease the amount of data to be stored.

Using individual triangles and testing for occlusion is a good way to create the tightest possible visibility set for any point in the scene and at first it may sound better than the approach presented here. However, the PVS storage issue becomes a big problem and limits the scalability. The slice-wise structure creates a good

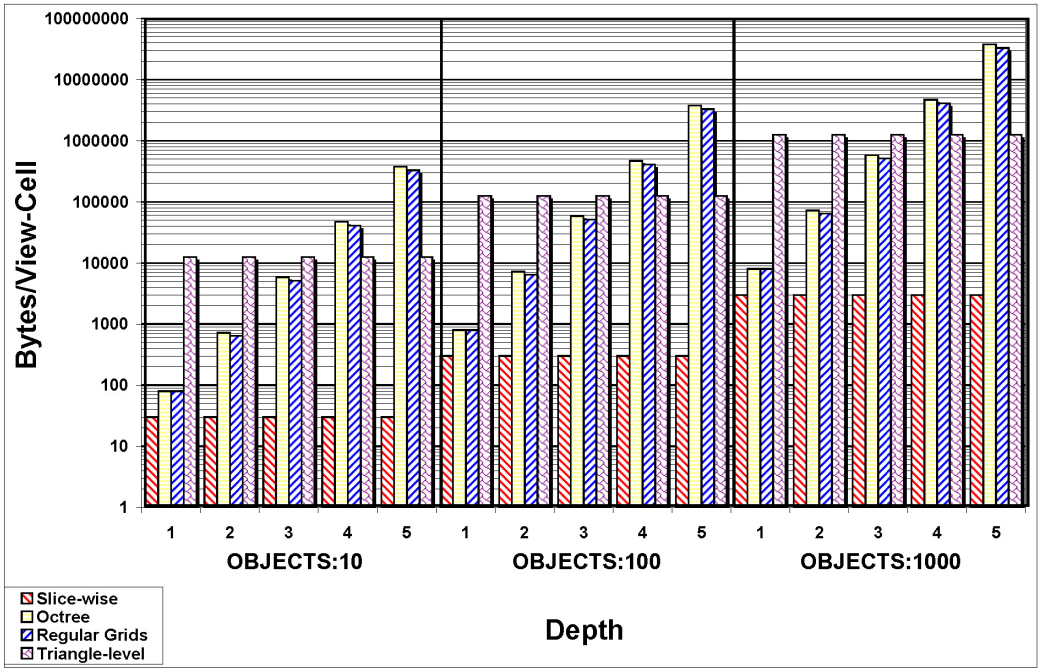


Figure 4.7: The comparison of the PVS storage requirements for each subdivision scheme. We assume that each object is composed of 10K polygons. The slice-wise representation provides a huge decrease in the use of memory. The values are in logarithmic scale.

balance between PVS storage and running time.

It is also possible to define some semantic properties and store occlusion information with respect to this information. For example, in [66], the authors define floors for the buildings, called as *2.5D+ε*; these buildings have more vertical complexity than 2.5D buildings. Their occlusion culling algorithm tests triangles and determine the visibility on a floor basis. However, this prevents its application to real city models obtained from sources, such as airborne laser scanners. Our algorithm is capable of handling all types of buildings, do not need floor information and determines occlusion with respect to three axes. In this way, it captures

a tighter PVS than the floor-based visibility calculation. Since it requires more processing time, it is applied as a preprocessing step.

The slice-wise representation is very suitable for determining occlusion of cube-like objects. A building in an urban environment is one example to these classes. Due to the intention of creating a balance between PVS storage and running time, online visibility determination is very is not scalable. However, it automatically displays objects as a whole without any cost. Therefore, the worst case behavior of the slice-wise representation is as if not using it at all. Due to the setup costs, most approaches can only be used in environments with much occlusion. This property makes the slice-wise representation suitable even for the environments with less occlusion.

4.2 Slice-based From-Region Visibility

We want to show the applicability of the proposed slice-wise representation and efficient ways for the usage of it in a typical occlusion culling framework. The occlusion culling framework is from-region and conservative.

Figure 4.8 shows the framework for urban visualization using the slice-wise representation. It mainly consists of a preprocessing phase and a navigation phase. To test the effectiveness of the slice-wise representation, we developed a conservative from-region visibility algorithm. To achieve conservative occlusion culling, we made use of the shrinking idea first proposed by Wonka et al. [100]. Our shrinking algorithm can be applied to any kind of scene object, including nonconvex ones.

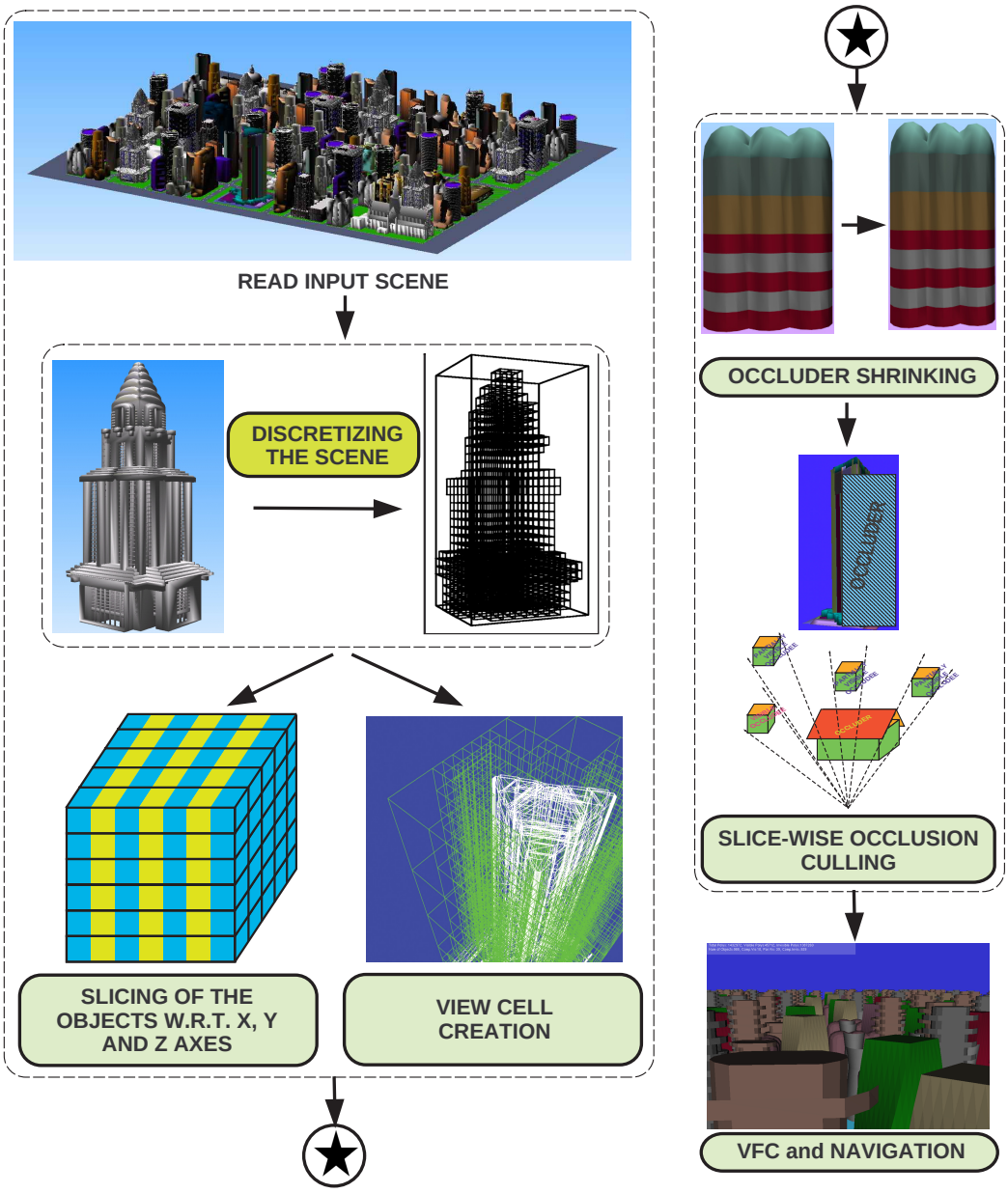


Figure 4.8: The urban visualization framework: in the first phase, we read the scene and calculate the bounding boxes of objects. Next, we discretize each object by checking if each triangle intersects with a predefined threshold-sized cube. After discretizing the object, we check the cubes for fullness to create slices and create the tree of octrees of the bounding-boxed object. These are used during preprocess as view-cells. After creating the shrunk versions of the objects, these slices are checked for occlusion and a tight visibility determination is performed for each grid location. The phases in dashed blocks are performed in the preprocessing phase. The View Frustum Culling (VFC) is also done during navigation.

4.2.1 Occluder Shrinking

The purpose of shrinking is to achieve conservative occlusion culling by sampling from discrete locations. It is possible to determine occlusion from a point and retain conservativeness for a limited area because the occluders are shrunk by the maximum distance that can be traveled in the view-cell. In order to achieve conservativeness, it is necessary to shrink occluders, so that behind the occluder is visible even if the user moves to the farthest possible location in the view-cell.

Wonka et al. [100] shrink occluders by using a sphere constructed around 2.5D occluders. Decoret et al. [32] generalize the shrinking by a sphere to erosion by a convex shape, which is the union of the "edge convex hulls" of the object. This is performed to create tighter visibility sets and to increase the occlusion region of the objects. They compute the shrunk versions of objects using an image-based algorithm at each view cell using a voxelized representation. This makes it very difficult to apply the presented image-based approach to general 3D objects.

4.2.1.1 Shrinking General 3D Objects

The exact shrinking can only be performed by calculating the Minkowski differences of the object and the view-cell [4, 87] and using the volume constructed inside the object as the shrunk version (see Figure 4.9).

Consider a general 3D object O and a set of vectors X of a view-cell. The dilation of O by X, also known as the Minkowski sum of both sets is defined by the equation:

$$O \oplus X = \{M + x \mid M \in O, x \in X\} \tag{4.1}$$

Here, X is commonly called the *structuring element* [32]. Thus, the inner volume, which composes the shrunk shape S of the object, can be defined as:

$$\begin{aligned} O \ominus X &= \{S \mid \forall x \in X, S + x \in O\} \\ &= \{S \mid \{S\} \oplus X \subset O\} \\ &= \{S\} \subseteq \{O \ominus X\} \end{aligned} \tag{4.2}$$

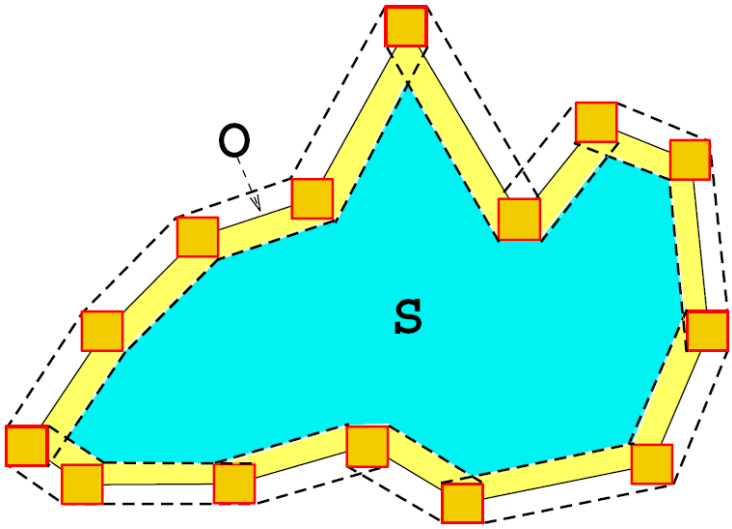


Figure 4.9: Shrunk volume extraction using Minkowski difference calculation. O denotes the object and S denotes the shrunk volume of the object.

Although there exist methods for exact Minkowski sums [94], it is practically very hard to find the exact Minkowski differences of general 3D objects. Our aim is to find a shrunk version of an object and retain the conservativeness of the view-cell visibility. Therefore, we need not be exact for the Minkowski difference calculations, i.e. we do not want to obtain a correctly calculated model as is shown for the summation case in [94]. Thus, we try to find an approximation to the difference, which will also satisfy the conservativeness of the occlusion-culling process. In this way, we can shrink any complex 3D object.

4.2.1.2 Shrinking Using the Minkowski Difference

We shrink an object by moving the vertices in the reverse direction of their normals. Although architecturally we do not make any assumption on the buildings, in order to achieve a good shrunk version of a model, closed mesh and connectivity in the geometry provide a suitable environment. Our aim is to use an approximate Minkowski difference of the object and the view-cell and still achieve conservative occlusion culling. It should be noted that the vertices are not moved with a constant distance (see Figure 4.11).

The traveled distance of a vertex during shrinking affects the final position of a

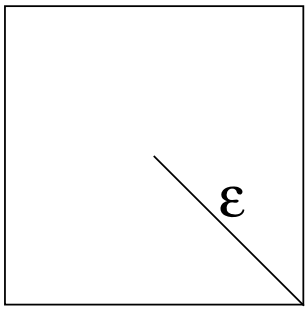
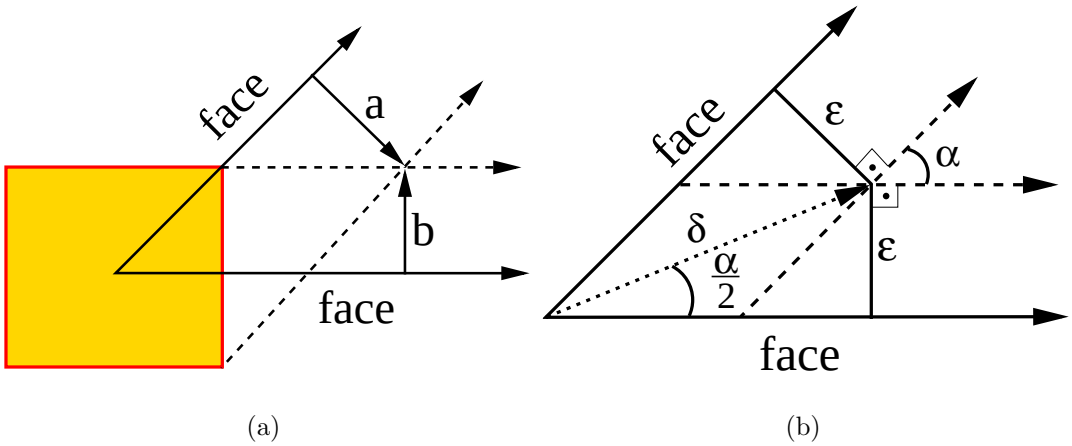


Figure 4.10: A sample view-cell, in which the user can move at most with ε distance.

face. For the exact Minkowski difference calculation, the movement distances of the faces towards inside vary with respect to the orientation of the view-cell, as shown in Figure 4.11 (a).



(a) (b)

Figure 4.11: Shrinking using the Minkowski difference: (a) In an exact Minkowski difference calculation, the movements of the faces towards inside are different with respect to the view-cell position. The two faces move with distances a and b. (b) The face movements are assumed to be at least the distance ε of Figure 4.10 to guarantee conservativeness. In this case, the vertex movement distance becomes δ. For easy interpretation, only an instance of the process where two faces sharing a vertex is shown.

The hard part in calculating exact shrinking using the Minkowski difference concept is calculating the distances a and b for any orientation and posture of the view-cell and the faces of the object, shown in Figure 4.11 (a). Instead we provide an approximation for it (see Figure 4.11 (b)). In order to guarantee the

conservativeness and derive a formulation, the faces should be moved at least the distance ε, which is the longest movement distance within a view-cell. We call this movement distance as δ. In this case it becomes a function of ε.

Theorem 1 *Let O be the occluder, S be its shrunk shape, and ε be the maximum travel distance from the center of the view-cell. If the minimum shrinking distance of O is greater than or equal to ε, the determined visibility from the center of the view-cell by using the shrunk shape of the occluder provides a conservative estimate for the whole view-cell (see Figure 4.12).*

Proof: According to Equation 4.2, the shrunk shape S is $\{S\} \subseteq \{O \ominus X\}$. Let X_d be the vectors of length d, which is smaller than ε and is the correct distance calculated using the Minkowski difference, that is $d = a$ or $d = b$ (see Figure 4.11 (a)). Hence, $\{S_d\} \subseteq \{O \ominus X_d\}$. Since ε is the maximum distance to be moved, then $\{a, b\} \leq \varepsilon$. Then, the volume of $\{S_d\}$ is greater than or equal to the volume of $\{S_\varepsilon\}$. Consequently, if $\{S_d\}$ is conservative, then $\{S_\varepsilon\}$, having a smaller volume, is definitely conservative.

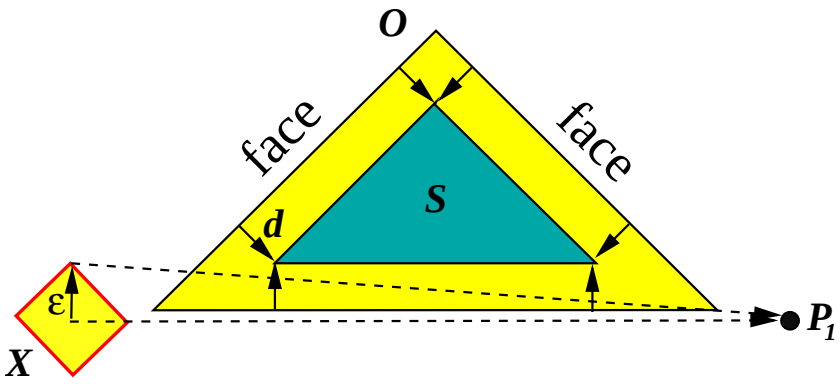


Figure 4.12: If a point P_1 is visible from the center of the view-cell, then it should also be visible with respect to its shrunk version S, even if the user moves to the farthest distance available in the view-cell, ε. If the inner movement distance d of the faces for the shrunk shape calculation is greater than or equal to ε, the conservativeness is guaranteed and the point P_1 becomes visible with respect to the shrunk version S. The reader is referred to [100] for the proof of the other case: if a point is occluded with respect to the shrunk version S, then it is also occluded with respect to its original version O, within an ε neighborhood.

4.2.1.3 Calculating Shrinking Distance for the Vertices

Using the notations of Figures 4.11 (b) and 4.13,

$$\frac{\alpha}{2} = min\left\{90 - arccos\left(\frac{\mathbf{N}_v \cdot \mathbf{N}_i}{|\mathbf{N}_v| \ |\mathbf{N}_i|}\right)\right\}, i = 1, 2, \ldots, n \tag{4.3}$$

where n is the number of faces sharing that vertex, $\mathbf{N}_v$ is the vertex normal, and $\mathbf{N}_i$ is the face normal. Then the shrinking distance of the vertex becomes $\delta = \varepsilon / sin(\frac{\alpha}{2})$. In order to calculate δ, we calculate the minimum angle between the vertex normal and all the neighboring face normals, since it yields to the longest distance and guarantees conservativeness. The calculated shrinking distance becomes a conservative bound on the real Minkowski differences of the model and the view-cell.

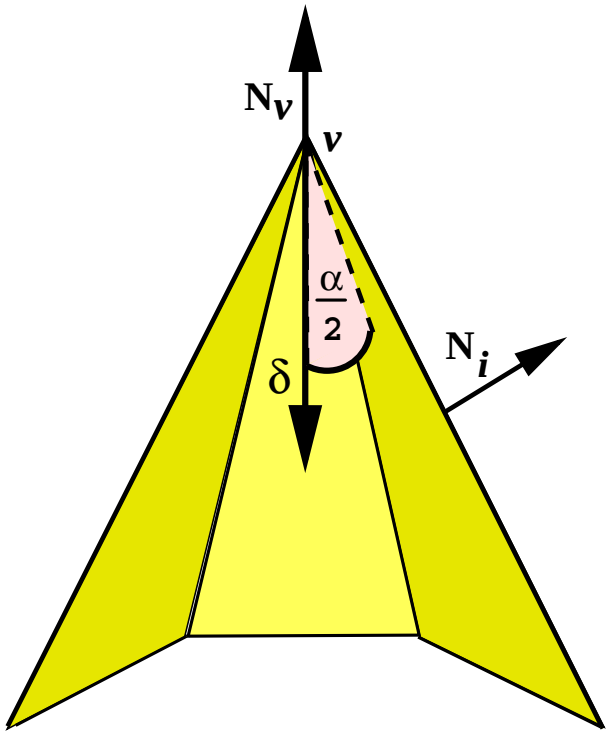


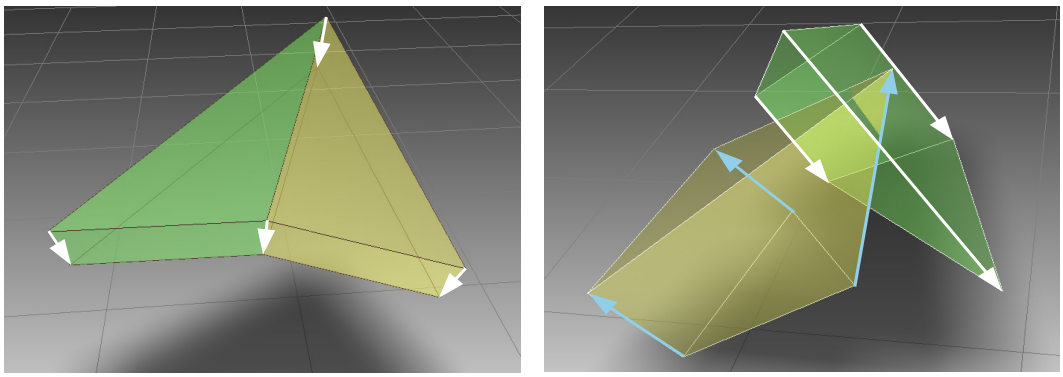
Figure 4.13: The object is shrunk by moving the vertex in the opposite direction of the vertex normal. The shrinking distance δ is calculated based on the view cell parameter ε and the angle $\frac{\alpha}{2}$.

4.2.1.4 Shrinking Occluders

The calculation of a correct shrinking distance is not enough to create conservative shrunk versions of the occluders; some faces may go inside one another and invalidate conservativeness (see Figure 4.14). To prevent such cases, we check the intersection of the volumes formed by the movement of the faces in the occluder and the corresponding faces in the shrunk version and remove the intersected ones. In order to accomplish this;

- we first check the intersection of their axis-aligned bounding boxes,
- if the axis-aligned bounding boxes intersect, we try to find a supporting plane between the volumes,
- if there is a supporting plane between them, the volumes do not intersect,
- otherwise, we remove the faces that cause intersection from the shrunk object (see Figure 4.15).

We also remove other bad cases such as triangles with no area and overlapping triangles. Shrinking examples are given in Figures 4.15 and 4.16.



(a) (b)

Figure 4.14: (a) Shrinking without any intersection: the neighboring triangles do not intersect because of common vertices. (b) The movement volumes of the two triangles intersect and this case results in removal of the triangles.

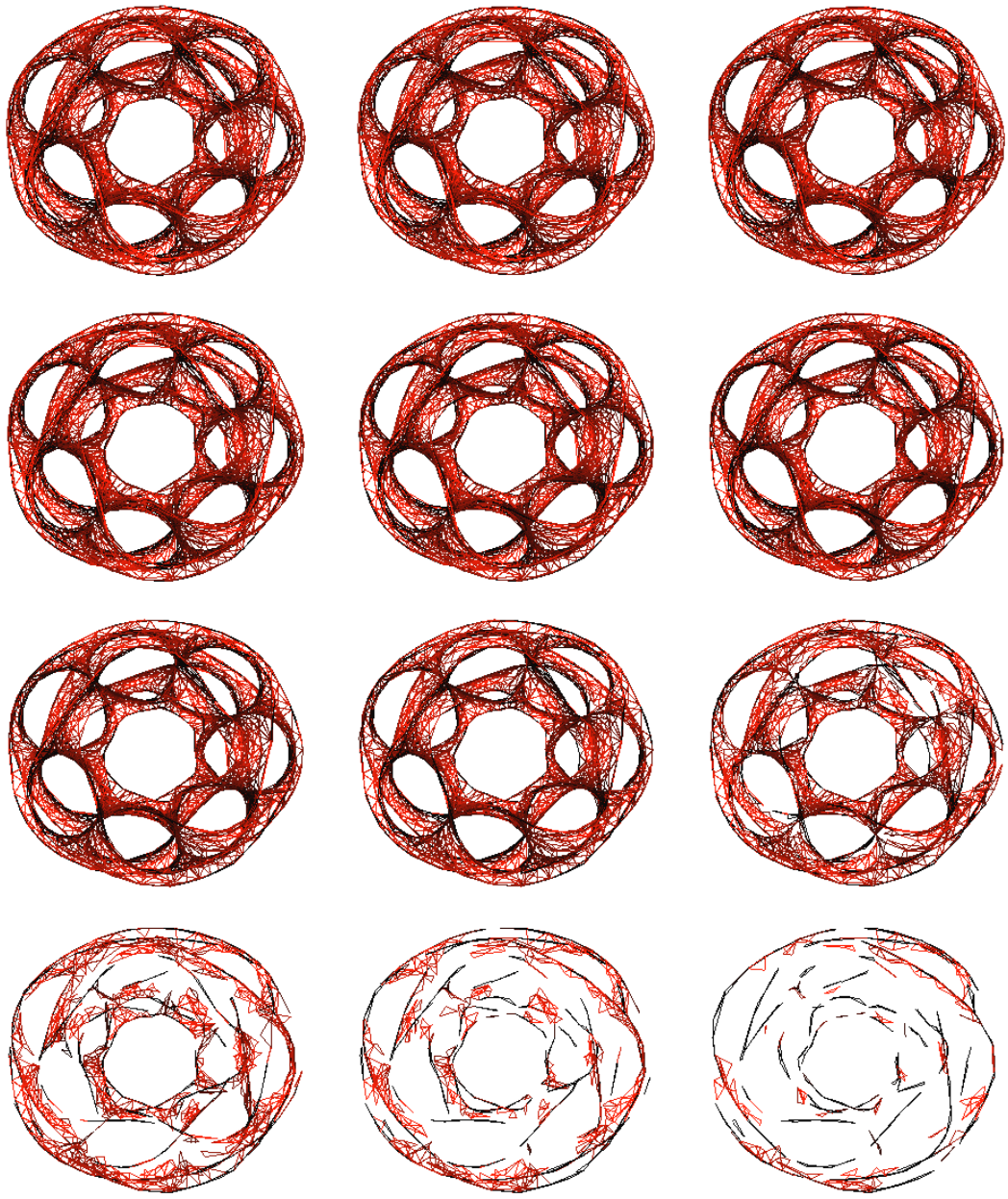


Figure 4.15: Shrinking applied to a heptoroid: The rims around holes shrink to null when the triangles from opposite sides start to intersect with each other (δ values are increasing in rowwise order).

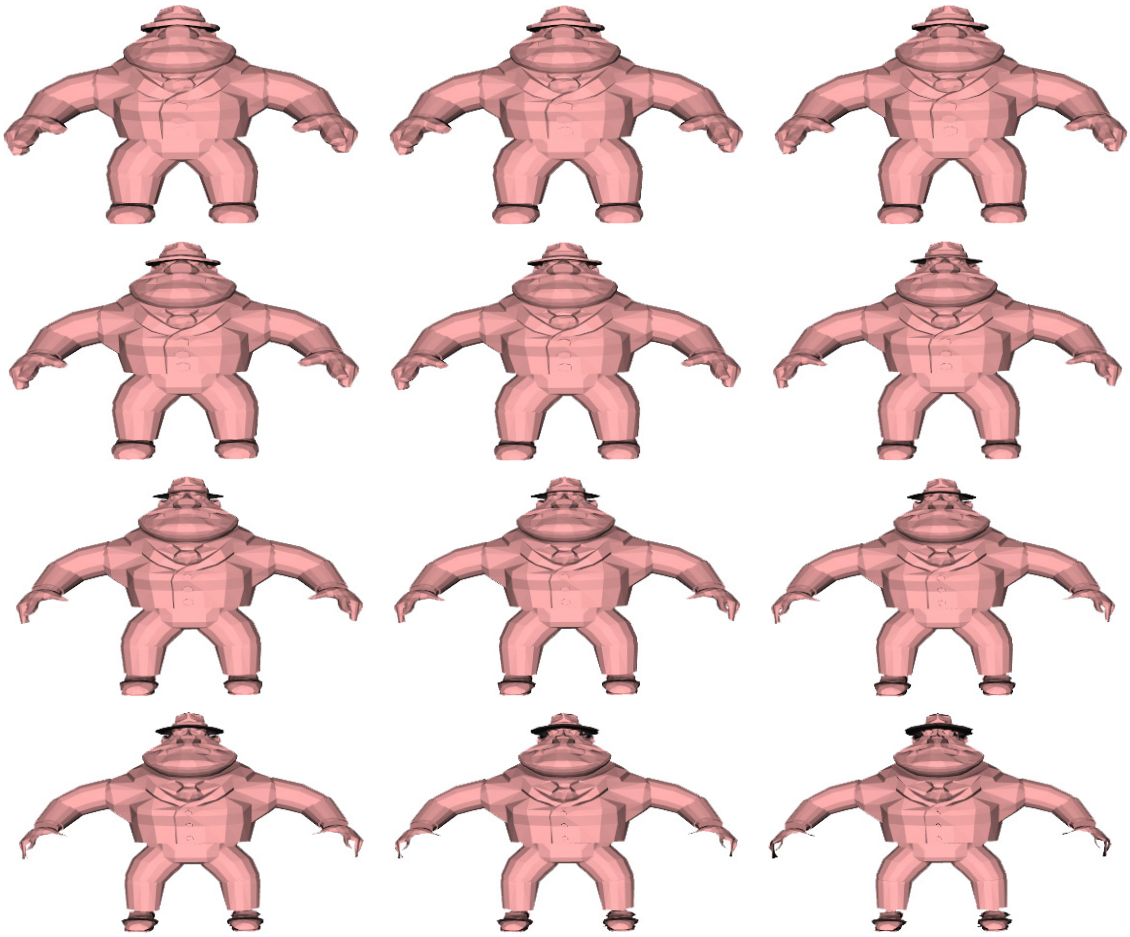


Figure 4.16: Shrinking applied to a general 3D object: the vertices touching the ground are not moved vertically (δ values are increasing in rowwise order).

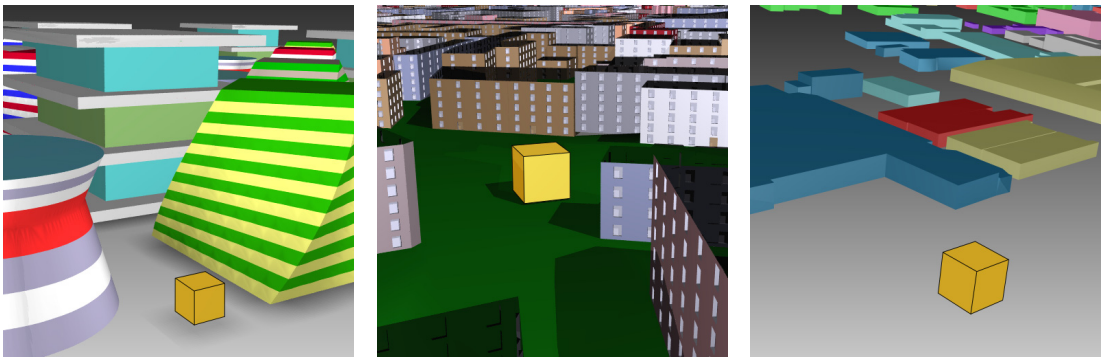
Our shrinking algorithm has some similarities with simplification envelopes [23]. Simplification envelopes are used to create simplified versions of 3D models. The simplified models are obtained by moving the vertices at most δ distance from their original position. When many triangles come close to each other, they are removed and smaller number of triangles are inserted. In simplification envelopes, it is guaranteed that the movement distance of the vertices are *at most* δ, whereas in our shrinking algorithm it is guaranteed to be *at least* δ. In simplification envelopes, the vertices are moved with small steps and the triangles are checked for intersection at each step. In our case, since the movement distance has to be at least δ, we calculate the necessary distance once and check for intersections later. Since our aim is to shrink the objects, we do not create new triangle

patches as opposed to simplification envelopes since the triangle patch creation may invalidate conservativeness.

The shrinking approach used in [20] is also applicable to general 3D scenes. They accept triangles or groups of connected triangles as occluders. They use the supporting planes of the triangles or a combination of supporting planes to construct an occluder umbra with respect to a selected projection point. The shrinking is performed in this umbra by just calculating the inner offset of the supporting planes towards the center of the occluder umbra which intersect at the projection point. Since they use planes of the triangles, the view-cell sizes are not the same and change with respect to the amount of the geometry. This is an intelligent approach which facilitates the load balancing of the geometry for each view-cell. However, in large environments, the view-cell partitioning may go deeper and a large number of view-cells can be faced with as reported by the authors, i.e. they have 500K view-cells ranging from several inches to a few feet wide. This is natural because it is very hard to use anything other than individual triangles or a few triangles for the case of occlusion culling in general environments. All of these issues result in smaller view-cell sizes and the need for a determination of the shrunk versions of the occluders for each view-cell. Instead we use the objects as a whole and calculate their shrunk versions once using the algorithms described above, which are valid for all the view-cells using the Minkowski difference concept. In our approach, we use the objects as a whole and calculate their shrunk versions once.

As a result of the occluder shrinking algorithm, some triangles may be removed from the model should there be any intersection during shrinking. Conservative property of the calculated visibility may be invalidated if we let these intersected triangles in the occluder model. This triangle removal may be overly conservative especially for the models with triangle sizes less than the shrinking distance. Exact shrinking is not computed where as we can find exact dilation using Minkowski sums. Creating a dilated model does not require intersection calculation –triangles may intersect without invalidating the dilation. However, exact calculation of the Minkowski difference requires creating new geometry instead of the intersected or eliminated ones, while checking the conservativity. For now

our intention is only to find a version of the model so as to provide a conservative visibility of the scene. Currently, conservativeness degree is achieved by adjusting the view-cell size with respect to the average size of the buildings. A visual comparison of a view-cell and the model is shown in Figure 4.17.



(a) (b) (c)

Figure 4.17: View-cells (outlined boxes) for different city models: (a) The view-cell in the procedurally-generated 40M triangle test model. (b) The view-cell in the Vienna2000 model. (c) The view-cell in the Glasgow Model.

4.2.2 Occlusion Culling

The occlusion-culling algorithm works in the preprocessing phase. It regards each scene object as a candidate occludee and performs an occlusion test with respect to all other objects in the scene. At each step, slices of the horizontal axis are checked for complete occlusion. The other two axis slices are checked for partial occlusion. An object is tested against a combination of the shrunk versions of all other objects; this creates occluder fusion and determines the occlusion amounts. This process is repeated for each navigable view-cell. The culling scheme is applied from a coarse-grained to fine-grained occlusion culling tests (see Algorithm 4.1).

```
foreach grid location in 3D do
    Draw the shrunk versions of the visible objects in the frustum as occluder;
    Cull portions of the scene using a quadtree;
    foreach candidate occludee belonging to visible quadtree blocks do
        Construct frustum towards the center of the occludee;
        Test the bounding box of the occludee using NV_OCCLUSION_QUERY;
        if the bounding box of the occludee is visible then
            Mark the object as VISIBLE
        else
            Mark it as INVISIBLE
    foreach VISIBLE object do
        Test slices using Algorithm 4.2;
    foreach PARTIALLY_VISIBLE object in the scene do
        Optimize visible slice counts using Algorithm 4.3;
```

Algorithm 4.1: The occlusion-culling algorithm: this algorithm differentiates between visible and invisible occludees. Then the visible objects are sent to the slice-wise occlusion culling algorithm. Next the number of visible slices are optimized and the PVS for the view-cell is determined.

4.2.2.1 Coarse-Grained Culling

The visible buildings are classified as either partially visible or completely visible. Determining these two forms require more occlusion queries to be performed. Algorithm 4.1 performs a coarse-grained occlusion culling to eliminate large invisible portions of the scene. It performs the following operations:

- Draw shrunk versions of all objects as occluders and disable color and depth buffer updates.
- We perform projection towards all 90-degree sections of the viewpoint and send quadtree blocks constructed from the ground locations of the buildings for being culled. We use hardware occlusion queries for this purpose. This step eliminates most of the invisible buildings quickly without testing them one by one.
- We calculate the projection of the ocludees belonging to visible quadtree blocks. In practice, although the calculated shrink versions are conservative, due to the use of graphics hardware to detect occlusion and hence the

rasterization errors can be encountered, the conservativeness can be violated. For example, a far away object may project to less than a pixel but an occluder right in front of the object may still cover the entire pixel due to the rasterization errors. These errors may be caused by projection, image sampling, and depth-buffer precision errors [45]. We overcome these problems by adjusting the viewing parameters for the projection so that the occludee is zoomed to the maximum extent on the occlusion test screen, which is 1024x768-pixels. This results in a very large view of the redrawn object. In other words, the outer contour of an occluder in the current view becomes tested with a very high precision. Besides, the bounding box of the candidate occludee is tested while generating two fragments for each pixel as in [45] and using antialiasing. In this way, rasterization errors that can be faced due to hardware occlusion queries are prevented.

- We test the bounding box of the candidate occludee. If the bounding box test returns visible pixels, we mark the object as *visible*, otherwise as *invisible*.

Most occlusion-culling algorithms stop after this step and accept an object as visible if the occludee becomes partially visible. We go through further steps and determine a tighter visibility set for the object. If the bounding box of the occludee is visible, we submit occlusion queries for the slices; we then determine the maximum occlusion height for each slice of the occludee using Algorithm 4.2. Thus, we classify the buildings as *completely visible*, *partially visible* and *invisible*. The visibility information for the slices is sent to Algorithm 4.3 to decrease the number of slices and determine the visibility indices to be used during navigation.

4.2.2.2 Fine-Grained Culling

Algorithm 4.2 performs fine-grained occlusion-culling. It checks the slices of a candidate occludee. To find the exact occlusion, we first submit occlusion queries and test the vertical slices with blocks of size Δ (see Figure 4.18). Next, we collect the query results. Finding the last invisible Δ allows us to determine the occluded height of the slice. Horizontal slices are checked for complete occlusion.

```
Generate and submit NV_occlusion_queries:
begin
    forall vertical slices do
        slice_increment⟵ 1;
        while slice_increment*Δ ≤slice_height do
            Query the slice box with height (slice_increment*Δ);
            slice_increment++;
    forall horizontal slices do
        Query the horizontal slice box;
end
Collect the results of the occlusion queries:
begin
    forall vertical slices do
        slice_increment⟵ 1;
        while slice_increment*Δ ≤slice_height do
            if The query returns any visible pixels then
                slice_occlusion_height ⟵ (slice_increment-1)*Δ);
                break;
            else
                slice_occlusion_height ⟵ slice_increment*Δ;
            slice_increment++;
        if slice_occlusion_height≡slice_height then
            Mark the slice as INVISIBLE;
        else
            if slice_occlusion_height≡0 then
                Mark the slice as COMPLETELY_VISIBLE;
            else
                Mark the slice as PARTIALLY_VISIBLE;
    forall horizontal slices do
        if The query returns any visible pixels then
            Mark the slice as COMPLETELY_VISIBLE;
        else
            Mark the slice as INVISIBLE;
end
if all slices are COMPLETELY_VISIBLE then
    Mark the object as COMPLETELY_VISIBLE;
else
    Mark the object as PARTIALLY_VISIBLE;
```

Algorithm 4.2: Testing the slices for occlusion: each slice is tested against the shrunk occluder. The vertical slice bounding boxes are drawn from the bottom to the top incrementing them gradually as in Figure 4.18 (b).

4.2.2.3 Optimizing the Visible Slice Counts

An object occluded by several occluders may have an irregular appearance that cannot be easily represented (see Figure 4.19). However, our aim is to decrease the amount of information needed to represent visibility, and therefore reduce the time to access the visible parts of the objects. In particular, the purpose of this optimization is to represent the visible area by using a small number of slices and determine a single index for each axis. We have to sacrifice tightness of visibility somewhat to reduce the access time and memory requirement (see Figure 4.19).

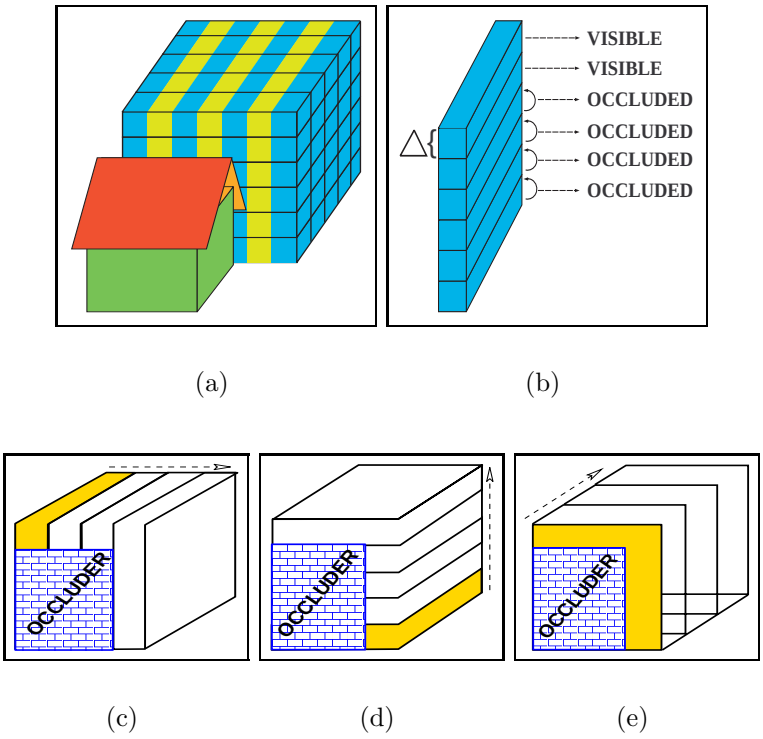


Figure 4.18: In order to determine the correct occlusion height in (a), the slices are tested beginning from the lowest unoccluded height and the point of occlusion is found as in (b) by iteratively eliminating blocks of size Δ from the vertical slice; this creates occluder fusion (the viewpoint is in front of the occluder). The test order for slices along the x, y, and z-axes are depicted in (c-e), respectively. While the slices in the x and z-axes are tested for exact heights, the y-axis slices are tested for complete occlusion (d). Testing y-axis slices for complete occlusion may result in unnecessarily accepting the slice as visible. However, this case is handled by optimizing the slice counts.

The algorithm for optimizing the slice counts is given in Algorithm 4.3. This algorithm is used to decrease the number of slices representing the visible portion of an occludee. In this algorithm:

- Any triangle of the object is represented by slices from three axes. We first find the maximally-occluded axis by calculating the occluded regions and the percentages of occlusion with respect to each axis.
- The rectangle that represents the occluded area is constructed.
- For all the slices of the maximally occluded axis, we discard the vertical ones up to the vertical edge of the rectangle and the horizontal ones up to the upper horizontal edge.
- The region above the upper edge of the rectangle is represented using horizontal slices and the region on the right- or left-hand side of the rectangle is represented using vertical slices.
- We discard the slices of the minimally occluded axis.
- Finally, the indices are calculated for each axis (see Figure 4.5).

$X\_occlusion \longleftarrow$ Percentage of occlusion for X-axis slices;
$Z\_occlusion \longleftarrow$ Percentage of occlusion for Z-axis slices;
$work\_axis \longleftarrow$ **max** $(X\_occlusion, Z\_occlusion)$;
Construct maximum sized rectangle of occlusion in the $work\_axis$;
forall *Slices of the* $work\_axis$ **do**
    Discard the vertical slices within the horizontal range of the rectangle;
    Discard the horizontal slices within the vertical range of the rectangle;
Discard the slices of the vertical axis other than $work\_axis$;
Assign the visibility indices to each axis for being stored;

Algorithm 4.3: The algorithm for optimizing the slice numbers for a partially occluded object: The algorithm reduces the number of slices used to represent the visible portion for an occludee (see Figure 4.19).

It should be noted that the visibility information for each partially visible object is represented using only three bytes, one byte for the visibility along each axis. The created PVS size is a function of the number of view-cells and the number of the objects. The size of the PVS does not change with respect to the model complexity(see Figure 4.7).

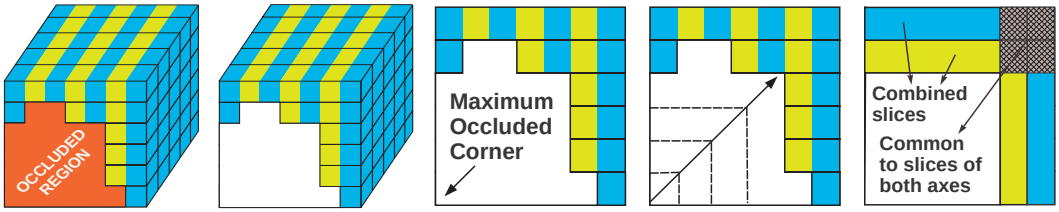


Figure 4.19: The resultant shape of the occlusion may have a jaggy appearance and we need to smooth it to represent with the slice-wise structure (a and b). This is handled as described in Algorithm 4.3. After selecting the maximally occluded axis, the starting corner of the occlusion is determined (c). The rectangle to represent the occlusion is determined (d). The vertical slices up to vertical edge of the rectangle and horizontal ones up to the horizontal edge are discarded (e).

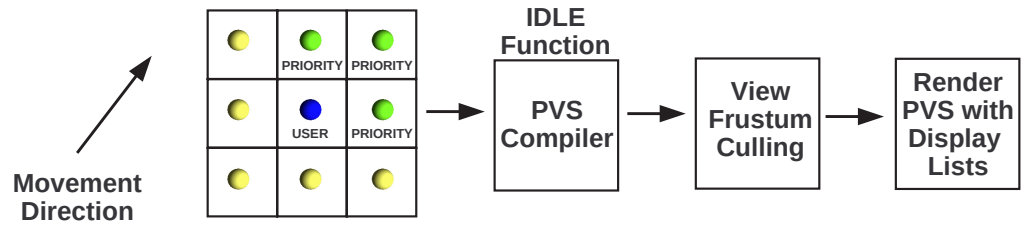


Figure 4.20: The rendering process: The user is in the blue view-cell. The display list for the view-cell is compiled before entering to the navigation stage, along with the neighboring view-cells. During navigation, the compilations of the display lists for the neighboring view-cells in the movement direction are given high priority. The display list compiler is attached to the idle function of the OpenGL so that the neighboring view-cell compilation does not cause bottlenecks. In this way, frame dips caused by the compilation of the display lists are prevented. After view frustum culling is performed on the quadtree blocks of the ground locations of the buildings, the constructed display lists for each building in the view frustum are rendered.

4.2.3 Rendering

Vertex arrays [20] and vertex buffer objects [107] are two popular techniques used for rendering the complex environments. We perform rendering by creating OpenGL display lists for the view-cells on-the-fly. OpenGL display lists are more flexible for run time purposes than using vertex buffer objects. We also create display lists for the neighboring view-cells of the user location to prevent performance degradation. This degradation might occur because of the display list compilation of the new PVS for the view-cell the user moves into. The neighboring view-cells are updated on-the-fly gradually without decreasing the frame rate below 25 fps. It should be noted that this operation is also highly parallelizable. The rendering process is shown in Figure 4.20. We avoid multiple renderings of the triangles at the intersections of the horizontal and vertical axes of the partially visible objects (see Figure 4.19 (e)).

Chapter 5

Stereoscopic Urban Visualization Using GPU

Urban environments provide the opportunity to detect a lot of occlusion during a walkthrough, which can be eliminated from the graphics pipeline as it does not contribute to the final view. Therefore, previous work has mostly concentrated on determining these occluded parts. The quality of a visibility algorithm depends on how fast it determines the visible parts of the model for different views, which are called *potentially visible sets (PVSs)*, and the degree of tightness of the potentially visible sets.

The advances in graphics hardware allow detection of occluded regions of urban geometry, even with complex 3D buildings. Visual simulations, urban combat simulations and city engineering applications require highly detailed models and realistic views of an urban scene. Occlusion detection using preprocessing is a very common approach, because of its high polygon reduction and its ability to handle general 3D buildings.

Virtual reality applications require special treatment because the geometry is rendered twice, once for each eye. Generally, performance-enhancing techniques such as view-frustum culling (VFC) are applied twice for both eyes; this increases the overhead. The first contribution presented in this chapter is a simple VFC approach, which requires only one application from a culling location well placed for

both eye coordinates rather than two locations for stereoscopic visualization. The view calculated from this location has the same coverage as both eyes together.

The second contribution presented in this chapter is the use of the graphics processing unit (GPU) for occlusion culling. The PVSs are determined in preprocessing time and the resultant visibility list is stored using a slice-wise building representation. We use the GPU to create buffer objects and store the model information in the high-speed memory thereby improve the rendering speed. In particular, we demonstrate how the GPU can achieve high frame rates during stereoscopic visualization.

We first explain how the GPU is utilized using the slice-wise representation for the monoscopic case. The GPU utilization is based on the memory configuration for the vertices of the buildings. During run-time, we use only the indices of the vertices in the GPU, which are bound by a single index denoting the locations of the vertices of the slices for partial visibility and for the completely visible buildings. Next, we explain our contribution for the stereoscopic visualization, namely the single application of the VFC.

5.1 Using Slice-wise Representation on GPU

A significant feature of this representation is that it facilitates the storage of partial visibility in case a building is partially visible for a view point. The slice-wise representation is constructed by applying a regular subdivision to a building, and then combining these subdivided blocks into slices for each axis. In this way, a triangle can be accessed by at least three slices. During the occlusion culling process, the slices –not individual triangles– are tested for occlusion. A building is tested for occlusion using the shrunk versions of other objects as occluders, and the slices of buildings parallel to each axis as occludees. The vertical slices are tested by gradually increasing their height and the first visible heights are recorded for each. The horizontal slices are checked for complete occlusion. After determining which slices and portions of each are visible, the resultant list is optimized and partial visibility is represented with only three bytes, one for each

axis. As a result, visibility is encoded by the first visible slice numbers of vertical and horizontal axes (see Figure 5.1). For the sake of simplicity, three bytes are stored for each building including the unused axis.

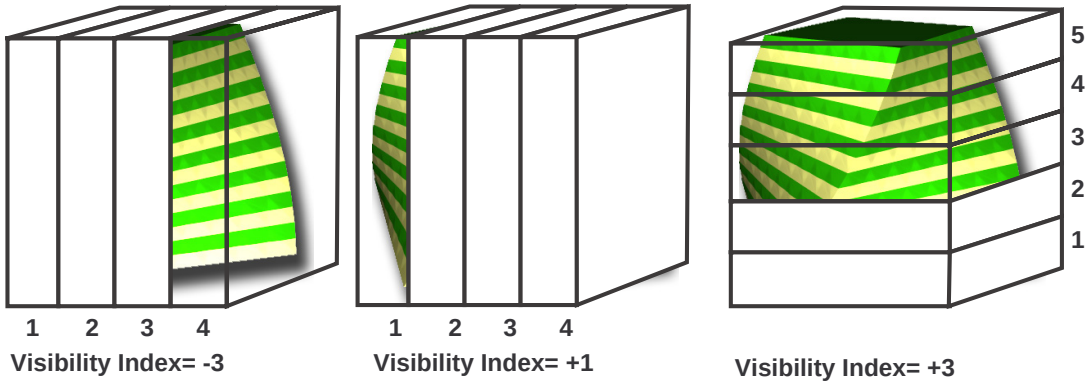


Figure 5.1: Visibility index determination using the slice-wise building representation: The index number to be stored depends upon the occluded section of the object. "+" or "−" signs are used to define the occlusion side.

However, the rendering method employed uses dynamic display-list compilation in OpenGL, which can cause bottlenecks if there is a large amount of visible geometry. To reduce display-list compilation bottleneck, we created display lists online for nine view-cells including the neighbors of the user's view-cell. This approach provides a suitable environment for visualization and eliminates frame dips that may arise because of the compilation. In the worst cases, it has the disadvantage of replicating display lists for all neighboring view-cells; this may lead to memory overflows.

5.1.1 OpenGL:Vertex Buffer Objects (VBO)

A VBO provides a mechanism for handling the data that might be pointed to by a function. Typically these functions are *glVertexPointer()*, *glColorPointer()*, *glNormalPointer()* and drawing commands such as *glDraw[Range]Elements()* [74]. This mechanism provides some chunks of memory (buffers) that are available through identifiers. These identifiers are used to refer to the memory chunks. As with the display list mechanism of OpenGL, the user is able to bind such a buffer on the client side. This binding operation turns every

pointer in a client function into offsets to be used on the server side. In other words, it turns a client function into a server function.

With VBOs, it is possible to keep vertices in high speed memory and access them quickly. Actually, the vertices are stored in a memory-efficient fashion and the memory usage becomes less than keeping them in separate linked list of pointers (see Figure 4.4). VBOs provide accessibility by using a binding pointer; they do not require that triangles be kept in the main memory.

5.1.2 VBO Creation for the Buildings

Our VBO configuration is shown in Figure 5.2. The vertex buffer is filled with the x, y, and z vertex coordinates for each building. A second buffer for each building is created, which stores the indices of the vertices for each triangle in *unsigned short int* data type. This limits the number of the vertices to be used to 65,535 per object, which is suitable for a single building representation. There is no limit on the number of triangles that can be formed with these vertices. This index buffer is used to represent completely visible buildings during navigation. Next, for each slice, other index buffers are created so as to represent partial visibility. It should be noted that the index buffers required for each slice can be created during walkthrough by storing the indices in main memory. The triangles are not needed after the VBOs for a building are created because by then, they are in the GPU.

Algorithm 5.1 gives the VBO creation algorithm. In the first part of the algorithm, vertex coordinates, normals and color data are sent to the GPU. These data will be used once with the rendering commands for the buildings, regardless of their visibility class. In the second part, the vertex index data for the triangles of a completely visible object are sent. Next, the same kind of data is sent for the slices. In the last part, the vertices, triangles and other related data are deleted from the main memory through linked lists. To implement this algorithm, the data structure shown in Figure 4.4 must be modified slightly to include binding values for complete visibility, and for the slices for partial visibility (see Figure 5.3).

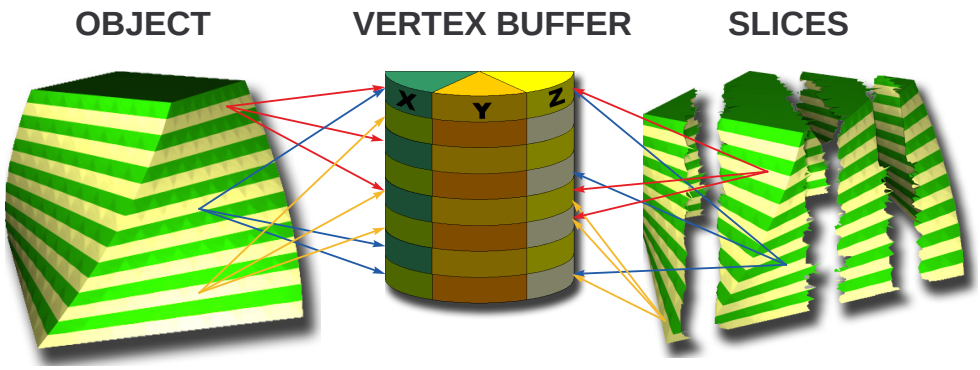


Figure 5.2: The VBO data structure used in GPU-based visualization. The object triangles are constructed using pointers to the vertex list and stored separately for each building.

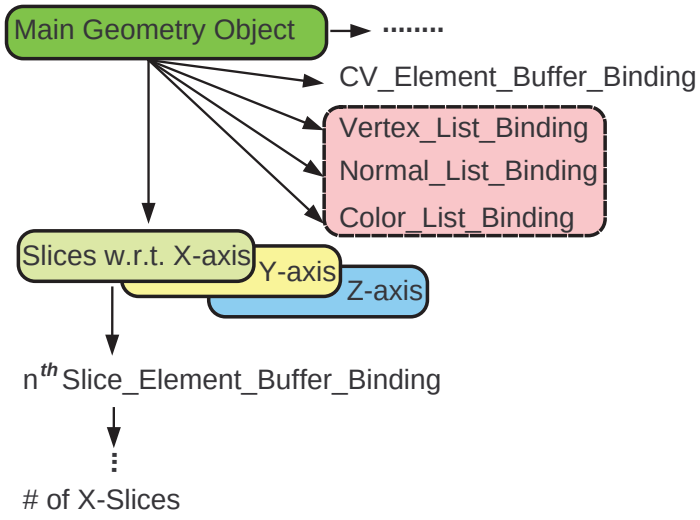


Figure 5.3: The modified data structure for slice-wise representation to facilitate GPU implementation: the vertex, normal and color list bindings point to their memory locations in the GPU. These data are referenced by the element buffer bindings (*CV_Element_Buffer_Binding* and n^{th} *Slice_Element_Buffer_Binding*) depending on visibility status during run-time.

foreach *Object in the model* **do**
- /* 1^{st}*part* */
- Generate a VBO Array Buffer for the vertices
- *Vertex_List_Binding*⟵VBO_vertex_buffer
- Send vertices in the main memory to GPU through *Vertex_List_Binding*
- Generate a VBO Array Buffer for the normals of the triangles
- *Normal_List_Binding*⟵ VBO_normal_buffer
- Send normals in the main memory to GPU through *Normal_List_Binding*
- Generate a VBO Array Buffer for the colors of the triangles
- *Color_List_Binding*⟵ VBO_color_buffer
- Send colors in the main memory to GPU through *Color_List_Binding*
- /* 2^{nd}*part* */
- Generate a VBO Element Array Buffer for indice list of triangles
- *CV_Element_Buffer_Binding*⟵ VBO_index_buffer
- Send array of indices to GPU as Element Array for the whole building
- **forall** *Slices of the building in X, Y and Z axes* **do**
  - Generate a VBO Element Array Buffer for indice list of triangles
  - n^{th}*Slice_Element_Buffer_Binding*⟵ VBO_index_buffer
  - Send array of indices to GPU as Element Array for the n^{th}Slice
- /* 3^{rd}*part* */
- Delete vertices, triangles, normals and color data from the main memory

Algorithm 5.1: The VBO creation algorithm: this algorithm is used to send the vertex coordinates, normals and color data along with the vertex indices of the triangles to GPU. In the first part, the necessary information for the vertices is sent. In the second part, we send the indices of the vertices for the triangles of a completely visible object and its slices. In the last part, these data are deleted from the main memory after they are transferred to the GPU.

5.1.3 Implications of Using VBOs for the Slices

The slice-wise representation coupled with VBO provides a suitable environment for visualization, because the only memory overhead of this representation is the index buffers that are needed. It has several benefits: it supports partial visibility; it provides the lowest potentially visible set storage cost; and it facilitates a fast visualization environment.

As a result, the storage and accessibility representation of each slice is fully utilized although the GPU memory amount may cause slight limitation on this issue. However, VBOs have the advantage of being able to swap with the main memory, if the GPU memory becomes full. We have performed tests even with 32 MB of GPU memory; there were no overflows.

The representation of each slice does not need to be changed. However, instead of keeping display lists and triangles in the main memory, they are kept in the high speed memory of the graphics hardware. This produces a huge decrease in the amount of main memory used because of the driver optimization of the OpenGL. Figures 5.2 and 5.3 show the resultant configuration and the memory resident structures for GPU-based visualization using the slice-wise representation.

5.1.4 VBO Referencing During Run-time

Run-time VBO access is depicted in Algorithm 5.2. In this algorithm, the slice-wise representation of buildings is exploited. This algorithm uses the visibility information, which is produced using the occlusion culling algorithm and the slice-wise representation. In this algorithm, the following operations are performed:

- First, the active view-cell (or view-cells since two eyes may be in two different cells) are determined by looking at the user location in the navigable space (see Chapter 3). Visible objects are determined and stored as a linked list for each view-cell.
- Next, this list is traversed and any completely visible objects are rendered using the *CV_Element_Buffer_Binding* index of the object. If the object is

partially visible, then we traverse the slices of the object. The occlusion can be either on the left or right of the vertical axes or in the lower part of the object (see Figure 4.5).

- Depending on the occlusion status, a variable is kept and if the object is occluded from the left and the right part is visible, which is denoted by a negative visibility index, we increment the variable and do not render the slice. Until the incremented variable becomes greater than the absolute value of the visibility index, we just pass the slice and then send the n^{th} *Slice_Element_Buffer_Binding* number and others for rendering.
- If the object is occluded from the right and the left part is visible, which is denoted by a positive visibility index, we render the slices until the incremented variable becomes greater than the visibility index.

5.2 Stereoscopic Rendering

The following conditions are required to achieve the best performance in stereoscopic visualization:

- The rendering rate should be sufficient to achieve interactive visualization; i.e., it should be at least 17 frames per second.
- The ghosting effect (crosstalk), which is caused by drawing a geometry for one eye and not drawing it for the other eye, should be reduced or eliminated.
- The strongest stereo effect with the lowest values of parallax should be provided. Parallax values should not exceed 1.6° [93].

5.2.1 Stereoscopic Projection Method

We applied off-axis projection with parallel frustums (Figure 5.4) for stereoscopic visualization, i.e., two projections are performed for each viewing direction and for each eye and converge at infinity. Since an urban scene contains many buildings at

```
Determine the active view-cells where the user eyes are located;
Apply Single Location VFC;
forall the objects attached to the active view-cells that are in the frustum do
    Get X, Y and Z indices;
    BindObject();
    if Y≡1 then
        /* object is completely visible                              */
        draw_CV_object(object);
    else
        foreach axes X, Y and Z do
            if slice_index≡0 then
                loop;
            pass_the_slice←—0;
            if slice_index<0 then
                /* Right part is visible                             */
                while slices are not finished do
                    pass_the_slice←—pass_the_slice+1;
                    if pass_the_slice>abs(slice_index) then
                        draw_slice();
            else
                /* Left part is visible                              */
                while slices are not finished do
                    pass_the_slice←—pass_the_slice+1;
                    if pass_the_slice≤slice_index then
                        draw_slice();
```

Algorithm 5.2: The algorithm for selecting the slices to be rendered. The selection is performed based on the visibility index assigned to the slice as described in [106], (see Chapter 4). The *BindObject()* function is used to inform the GPU that the object is to be accessed for rendering.

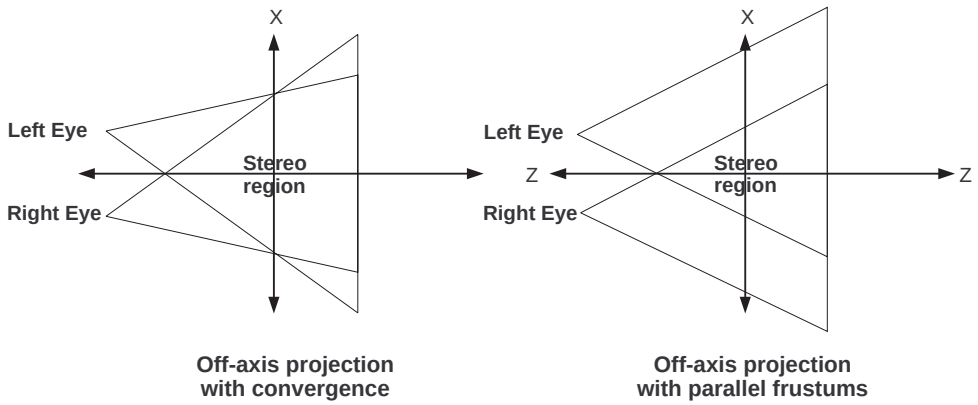


Figure 5.4: Off-axis projection using convergence is shown on the left. If the user converges to the assumed location in the scene, then perfect stereo is achieved. However, for urban scenes where there are lots of buildings, assuming a single convergence point is not realistic. On the right, off-axis projection with parallel view frustums is shown. Converging viewing directions at infinity decreases the ghosting effect if the viewing parameters are kept within reasonable limits.

a distance, we found that using off-axis projection with a single convergence point (*toe in projection*) causes lots of ghosting effects on the screen (see Figure 5.4). Because of the convergence angle and varying scene depth, locations other than the convergence point can have noticeable ghosting effect, even when the viewing parameters are kept within reasonable limits. In real life, the human eyes can converge easily to any point the viewer wants. In computer-generated stereo, it is not easy to determine the point where the user's eyes are converging; there has been some work in this area but these are not easily applicable [96, 109]. Using a convergence point works better for observing a single object. Therefore, we choose to use off-axis projection with parallel view frustums converging at infinity. If the stereo parameters, such as interocular distance and user-screen distance are kept within reasonable limits, the ghosting effect on the inner parts of the screen becomes unnoticeable. We do not use on-axis projection because it causes image distortions at the peripheries of the screen due to projection transformations.

5.2.2 View-Frustum Culling in Stereoscopic Visualization

View-frustum culling is one of the most important methods of eliminating primitives that do not contribute to the final image during navigation. It is generally performed twice for stereoscopic visualization. We made a simple change to decrease the number of VFC operations for stereoscopic visualization from two to one. Instead of performing VFC according to the locations of the eyes, we move backwards a calculated distance and put the culling location at the spot indicated in Figure 5.5. This location is determined by using the midpoint of both eyes, frustum angle, and the interocular distance. The viewing frustum becomes enlarged by moving the user position virtually backwards, until the new frustum edges coincide with the right edge of the frustum with respect to the right eye and the left edge of the frustum with respect to the left eye. Thus, we are able to cover the whole region that can be observed during stereoscopic visualization. Although this single-location VFC increases the number of polygons to be processed for rendering, it is much less costly than performing VFC twice.

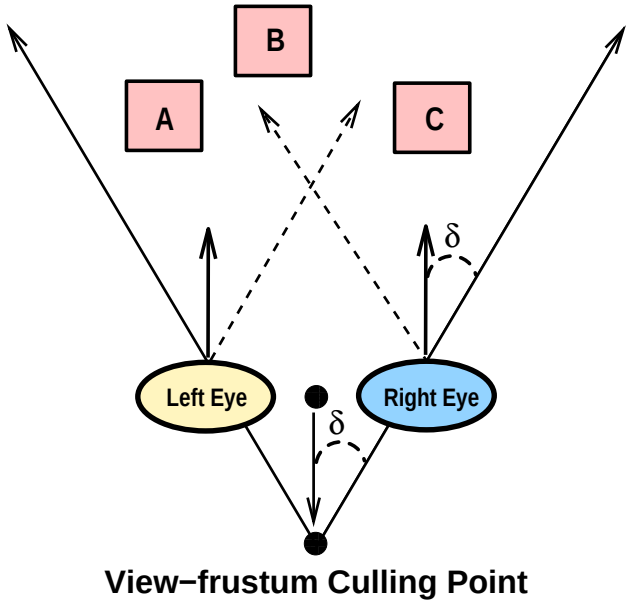


Figure 5.5: Changing the VFC location: since we know the projection angle, the exact distance to move backward becomes a simple function of half of the eye separation distance and half of the projection angle (*backward_distance=half_interocular_distance/tan(δ)*). By moving the VFC location, a single test can cover all the volume that can be viewed in stereo.

VFC can be performed on the unoccluded objects by making an inorder traversal of the scene quadtree. Another solution is to test the bounding boxes of each unoccluded object one by one. Our experiences show that when the scene quadtree subdivision depth is too high, it may take longer to cull the objects from the frustum than testing unoccluded objects one by one. Since the scene is large and the number of visible objects is much smaller than the number of quadtree nodes, for ground-based navigation it is faster to test only the bounding boxes of individual buildings in urban scenes.

VFC can be done using stencil tests on the quadtree blocks of the unoccluded geometry. It can also be carried out by applying hardware occlusion queries for the quadtree blocks. If the scene hierarchy is to be used for the VFC operation, then the in-frustum information for each node of the hierarchy is needed, in order to determine the tests for deeper level nodes. However, this requires a hardware occlusion query setup and retrieval operation for each quadtree block and the setup time for hardware occlusion culling is longer than it is for the stencil buffer mechanism. This is not the case for testing the bounding boxes of each object individually; all of the bounding boxes can be sent to the GPU in a single batch using hardware occlusion query, and the ones returning visible pixels can be quickly rendered. These options are scene dependent and we have chosen to test the bounding boxes of the objects using hardware occlusion queries; we use an empty screen as an occluder buffer and test the bounding boxes of each object individually.

Chapter 6

Results

6.1 Navigable Space Extraction

During the development of navigation systems for urban sceneries, the navigation determination becomes one of the most vital parts of the work. The navigation space determination is simple for the scene databases where the building footprints are used and the navigation is bounded to the ground. However, for the systems that need 3D navigation and the scene database is composed of complex objects where footprints do not define the navigable area, the navigation space determination becomes one of the most daunting tasks.

At this point, our approach becomes a solution to the definition of navigable space determination. It also constructs the hierarchical scene database as an additional feature. One important feature of our approach is that it is independent from the architecture of the scene objects. The building models may have pillars or holes where seeing through them is also possible. The method can be applied to any type of unstructured scene files composed of objects such as buildings. The application of the method produces two octrees; one containing the definition of the navigation area and another one containing the scene hierarchy, both in the form of forest of octrees. A sample from the created octree is shown in Figure 6.1. The created cells are then used as view-cells, from where the occlusion culling is performed using shrinking with the Minkowski difference. Besides, using the

created navigable space octree, it is possible to perform urban navigation in a fly-through type application.

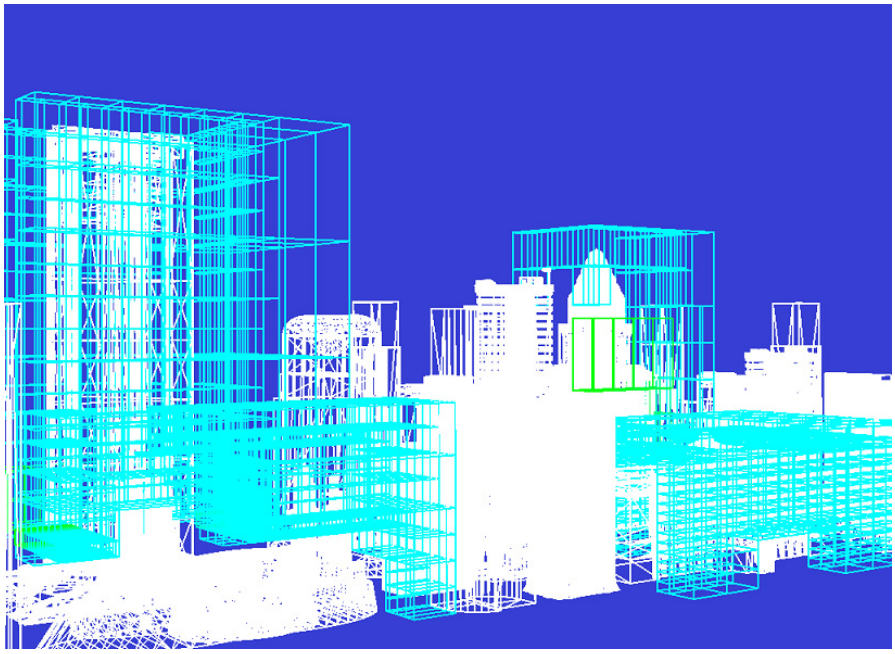


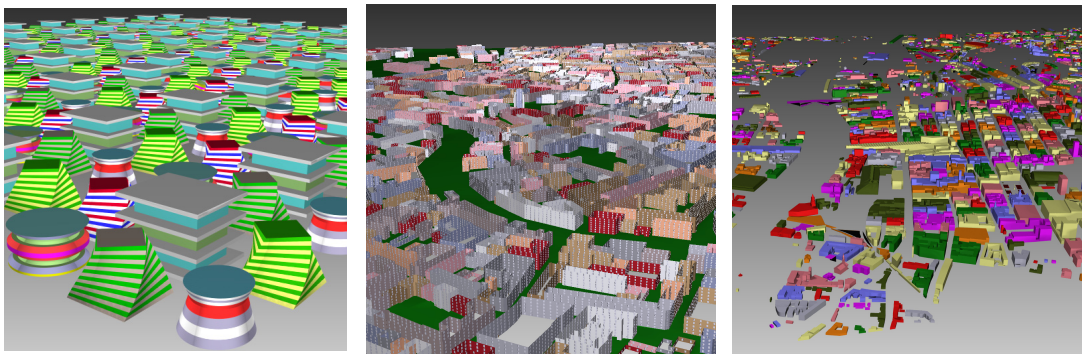
Figure 6.1: Created octree structure for a small urban model.

6.2 Occlusion Culling using Slice-wise Representation

In this section, we will present the results of the tests performed using the display list method for rendering. In the next section, we will present the results of using the GPU-based rendering of the stereoscopic visualization with comparisons and improvements.

6.2.1 Test Environment

The proposed algorithms were implemented using *C language* with *OpenGL* libraries; they were tested on an Intel Pentium IV- 3.4 GHz. computer with 4 GB



(a) (b) (c)

Figure 6.2: The models used in the empirical study: (a) the procedurally-generated 40M-polygon test model is composed of 6144 buildings ranging from 5K to 8K polygons each; (b) the Vienna model is a 7.8M-polygon model that has 2078 blocks of buildings ranging from 60 to 30768 faces. In this model, contrary to previous approaches by other authors, each surrounding block of buildings is accepted as a single object during the tests. (c) Glasgow model has originally 290K polygons. However, the mesh structure is not well-defined and has intersecting, long and thin triangles. Therefore, the mesh structure has been refined and a total of 500K polygons are used during the tests.

of RAM and NVidia Quadro Pro FX 4400 graphics card with 512 MB of memory. We use three different urban models for the tests (Figure 6.2). The first one is a procedurally-generated model using a few detailed building models, which is composed of 40M triangles. The second one is the model of the city of Vienna composed of 7.8M triangles (Vienna2000 Model with detailed buildings). The last one is the Glasgow model, which is a relatively small (500K) model. Table 6.1 shows various statistics about these models. The resource consumption and parameters for the models are summarized in Table 6.2. We discuss the results for the largest one, 40M-polygon procedurally-generated model. The interpretation of the test results for two real city models are similar. The tests were performed in 1024x768 screen resolution. The navigation algorithm is supported by a view-frustum-culling algorithm, which eliminates the objects that are completely out of the view frustum.

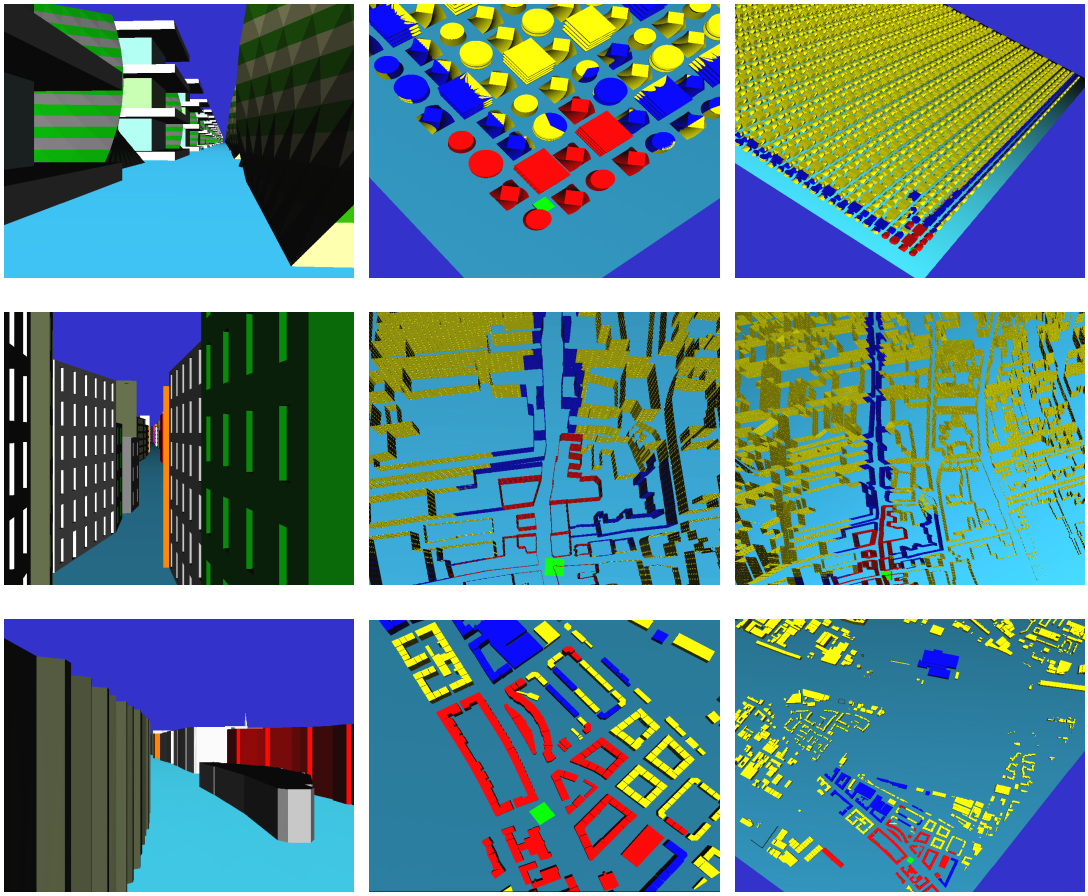


Figure 6.3: Still frames from the navigations through the scenes used in the experiments. On the left column, still frames from the current viewpoint are shown. In the middle, the views above the view-points are shown. The view-cell is the green box. On the right, larger areas showing the results of the slice-wise occlusion culling are shown. Partially visible buildings are in blue, completely visible buildings are in red, and invisible buildings are in yellow color. The rows belong to procedurally-generated, Vienna, and Glasgow models, respectively.

Table 6.1: Statistics of the models used in the tests.

Model	Procedurally generated	Vienna 2000	Glasgow
No. polygons	40M	7.8M	500K
No. buildings	6,144	2,086	1,461
Model size	100K x 63K	2,385 x 2,900	4,246 x 3,520
View-cell size	200 x 200	10 x 10	15 x 15
No. navigable cells	45.5K	72K	66K

Table 6.2: Summary of the test results using the slice-wise structure. *Since the procedurally-generated model contains 6 different types of buildings repetitively, total shrinking time is low.

Model	Procedurally generated	Vienna 2000	Glasgow
Total PVS size on disk (MB)	52	18	65
No. slices	377,920	30,392	11,948
No. triangle pointers	136.2M	27.3M	1.6M
Slice-wise memory usage (MB)	1,094	218.7	12.4
PVS calculation time / cell (msec)	323	292	436
Shrinking time / building (sec)	30.0	13.8	3.7
Total PVS calculation (hrs)	4.08	5.6	8.0
Total shrinking time (hrs)	0.05*	8	1.5

For the procedurally-generated model, the navigation area is divided into 200-pixel grids using the navigable space extraction algorithm described in Chapter 3. The area of the city is 100Kx63K pixels. There are about 45.5 K navigable grid points in the scene, from where the visibility culling is performed. The test city model used in the experiments consists of 6,144 complex buildings with six different architectures, each having from 5K to 8K polygons with a total of 40M polygons. The slices are 200 pixels wide, the same width as the grid cells, although they can be different to adapt to the dimensions of the buildings. On the average, there are 15 slices on the x and z axes. The number of slices of the y axis depends on the heights of the buildings, which in our case is around 25 slices. As a result each object has about 55 slices. Preprocessing takes approximately 323 milliseconds for each view-cell. We perform a navigation containing 12,835 frames (Figure 6.3). The navigation is performed on the ground to make the occlusion results comparable with other works. It should be noted that flythrough-type

navigation is also possible without any modification.

The first aim of the empirical study is to test whether our slice-wise structure and the shrinking algorithm provide an advantage in occlusion culling over one, where an object is sent to the graphics pipeline completely even if it is only partially visible. This is performed to test, if there are any overheads that will prevent its usage for fine grained visibility testing. *Slice-wise occlusion culling* refers to occlusion culling where the granularity is individual slices whereas the *building-level occlusion culling* refers to the occlusion culling where the granularity is buildings. The second aim is to compare the PVS storage requirements of an occlusion culling approach using a slice-wise representation and other subdivision schemes, such as octree and triangle level occlusion culling.

6.2.2 Rendering Performance

Figures 6.4, 6.5 and 6.6 shows the frame rates obtained using the slice-wise and building-level occlusion culling. The graphs are smoothed for easy interpretation using a regression function. The average frame rate of the building-level occlusion culling is 61.36 frames per second (fps). We achieve average frame rates of 111.06 fps by using the slice-wise granularity, 81 % faster than using building-level granularity. In our tests, 99.26 % of the geometry is culled on the average. The culling percentages strongly depends on the size of the view-cell used. As the size of the view-cell increases, the number of preprocessed occlusion culling operations decreases, whereas the number of the triangles unnecessarily accepted as visible increases. As examples of geometry culling performance: in [52] from 72 % to 99.4 %; in [66] from 99.86 % to 99.95 %; in [100] 99.34 % culling ratios are reported.

In [107], it is reported that 25-55 % speed up is achieved, when from-point occlusion culling is used. We obtain double speed up from occlusion culling with respect to their approach and achieve this just by using slice-wise occlusion culling instead of building-level occlusion culling.

Figures 6.4, 6.5 and 6.6 also gives the number of polygons rendered for the slice-wise and building-level occlusion culling. Using the building-level granularity, 93

buildings and 592K polygons are drawn on the average for each frame. Using the slice-wise occlusion culling, 89 of these buildings are accepted as partially visible. This decreases the number of rendered polygons to 290K on the average, which is approximately 49 % of the number of polygons rendered with building-level occlusion culling. Table 6.3 gives the average frame rate and rendered polygon count comparisons for slice-wise and building-level occlusion culling methods for the three test models.

Table 6.3: Comparison of the average frame rates and rendered polygon counts for slice-wise occlusion culling and building-level occlusion culling. Frame rate is in frames per second (fps); polygon counts are the average number of polygons rendered per frame.

	Model	Procedurally generated	Vienna 2000	Glasgow
Frame rate	Slice-wise	111.06	135.1	152.2
	Building-level	61.36	77.8	120.1
Polygon count	Slice-wise	290K	122.1K	26.2K
	Building-level	592K	227.6K	40.1K

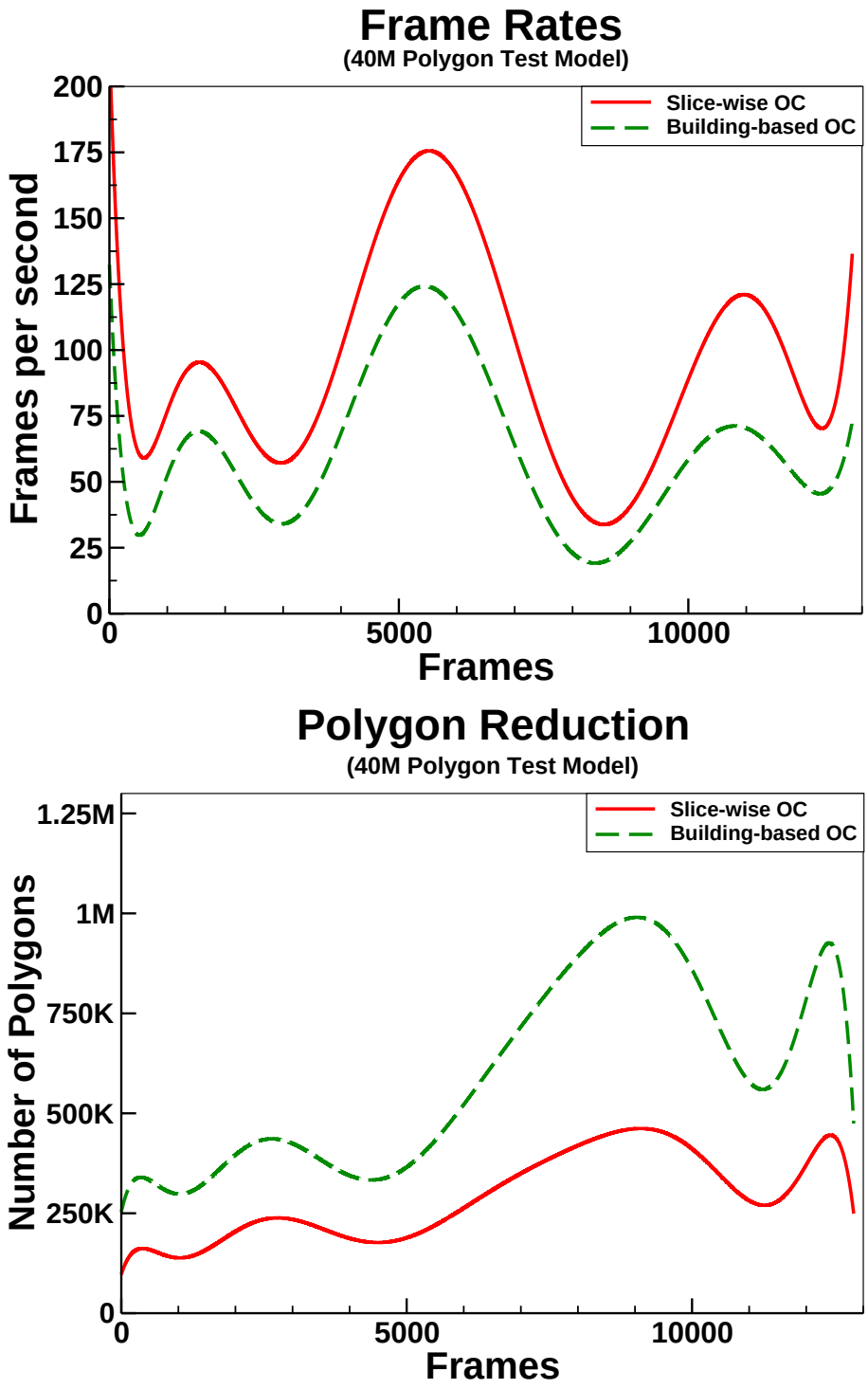


Figure 6.4: Frame rate speedups and average number of polygons rendered of the proposed approach as compared to the building-level approach for the procedurally-generated model. The frame rate speedup is 81 %; the polygon reduction is 51 %; and average culling rate is 99.26 %.

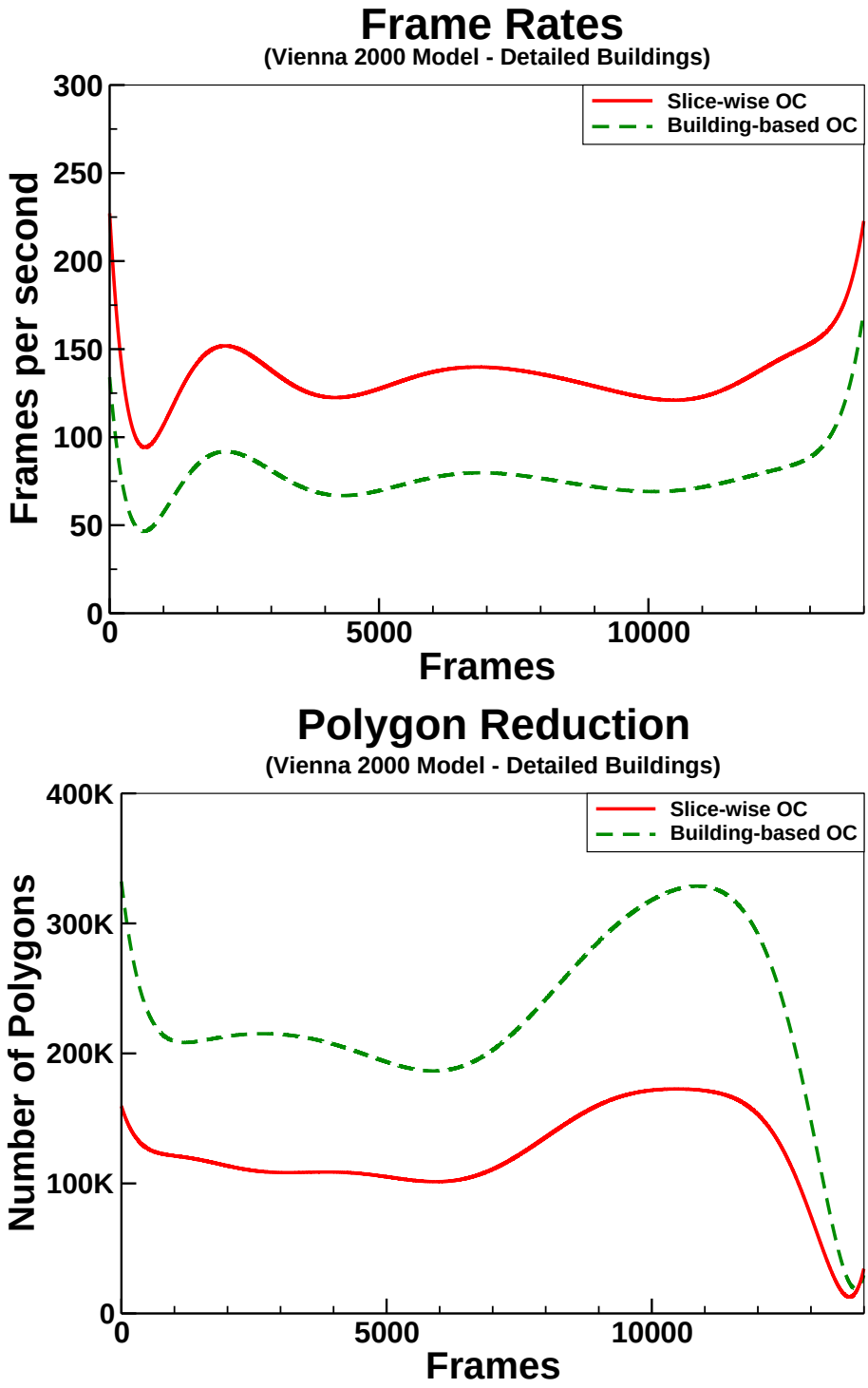


Figure 6.5: Frame rate speedups and average number of polygons rendered of the proposed approach as compared to the building-level approach for the Vienna2000 model. The frame rate speedup is 73.7 %; the polygon reduction is 46.3 %; and average culling rate is 98.42 %.

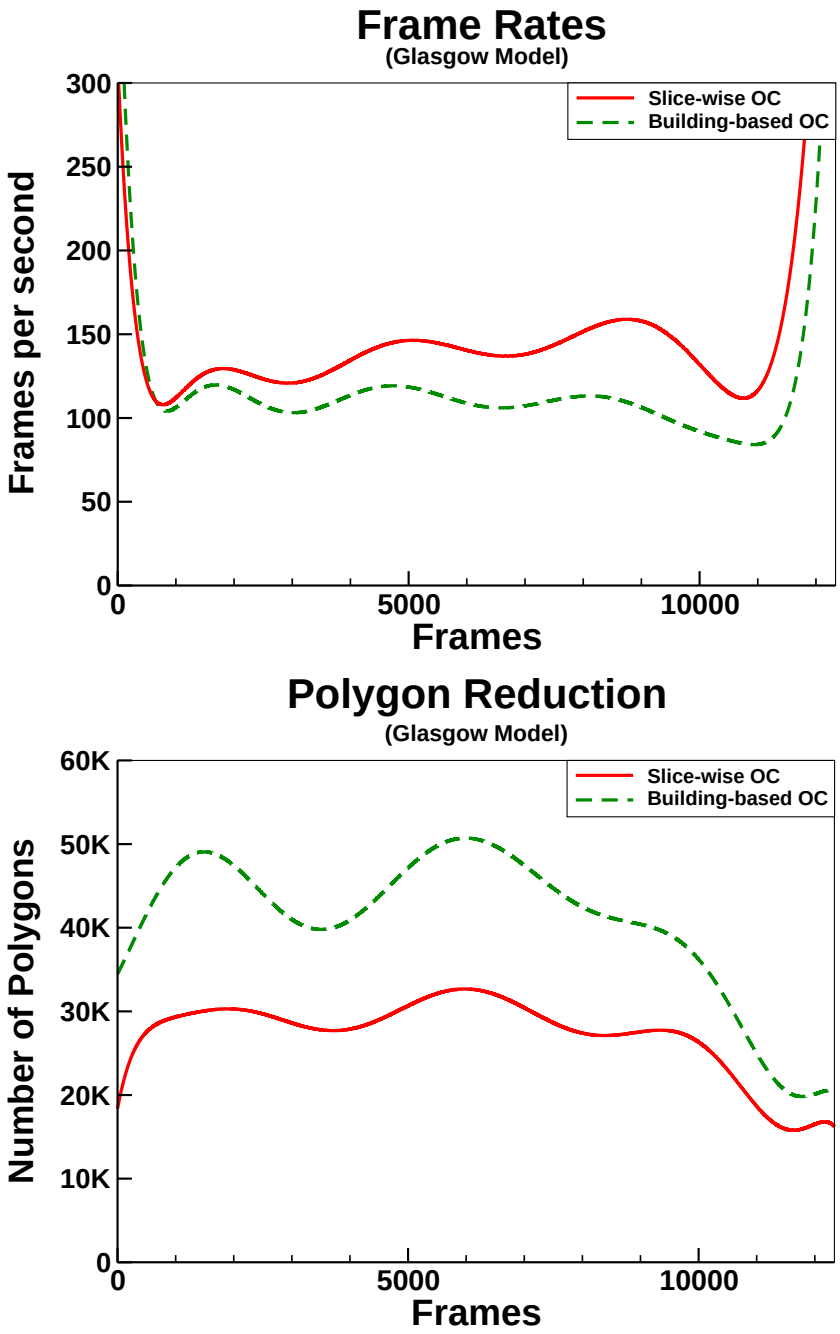


Figure 6.6: Frame rate speedups and average number of polygons rendered of the proposed approach as compared to the building-level approach for the Glasgow model. The frame rate speedup is 26.7 %; the polygon reduction is 34.6 %; and average culling rate is 94.8 %. One reason for the lower performance increase in the Glasgow model is that the average number of the polygons rendered are very small for both granularities and the GPU is not fully utilized. The other reason is that the mesh structure has long and thin triangles belonging to several slices, thereby decreasing the exploitation of the representation.

6.2.3 PVS Storage

The proposed slice-wise occlusion culling algorithms are optimized to exploit their benefits. Applying each subdivision scheme and performing tests on their performances would require different optimizations. This would make the comparisons unbalanced. Therefore, we only give informal results of using octrees and polygon level occlusion culling processes for their effects in terms of the resultant PVSs.

We compare PVS storage requirements of the slice-wise structure and other subdivision schemes, namely the octree-based and triangle-based PVS storage (see Figure 4.7). In 45.50K navigable view-cells, there are 4 completely visible, 89 partially visible buildings and about 290K visible polygons on the average. The PVS created using the slice-wise structure is 52 Mbytes, where each partially visible buildings is represented with 55 slices on the average.

For the octree structure, a building should be represented with 4680 nodes to obtain the same granularity with the slice-wise structure used here, which requires a subdivision depth of 4. We assume 75 % of this amount for the adaptive octree case, which is 3510 bytes. We further assume that each node of octree for the buildings is in the memory and the visible nodes are represented in bits, hence decreasing down to around 438 bytes/building. Totally, the octree-based PVS storage requirement is about 1.95 Gbytes.

For the triangle level PVS storage, there are about 13.20 billion triangles (290 K visible polygons for 45.5K view-cells), which should be encoded into bits resulting in about 1.65 Gbytes. Thus, the storage requirement of the slice-wise structure is about 3.15 % of the triangle-level PVS storage and 2.67 % of the octree-based PVS storage. Since, the PVS storage requirement for the slice-wise representation is the same for all subdivision levels, as the subdivision goes deeper, it becomes more advantageous to use it.

A rough comparison of the PVS storage requirement of the proposed approach with some well-known approaches is as follows. In [100], the model used consists of 82K view-cells with 7.8M triangles and a PVS with a size of 55MB (building-level occlusion culling is used). In [20], the model used consists of 90K view-cells

with 34M triangles. The authors employ a very powerful compression and decompression scheme for the PVS and they have 1.1GB-size PVS for their environment (polygon-level occlusion culling is used). In our test environment, we have 45.5K view-cells and 40M triangles. The PVS created by our scheme is 52MB without any compression.

In [107], there is no preprocessed occlusion information –the occlusion culling is done on the fly–, however, their approach requires 88MB per million vertices for the Clustered Hierarchy of Progressive Mesh representation. This accounts to 3.5GB storage requirement for our model.

For the scenes that have a lot of connectivity, it may be necessary to subdivide the scene into clusters as in [11, 20, 107]. The clustering approach is suitable for the cases where there is no natural object definition. However, the buildings are mostly disconnected for urban models. Thus, the cluster formation process is not very useful since the quadtree or k-d tree-based hierarchy for the ground locations of the buildings serves the same purpose, as shown in [66].

Our approach can also capture occlusion in birds-eye view. However, since the occlusion becomes less and the amount of the visible geometry increases, it may be more suitable to combine the approach with LOD (Level-of-Detail) rendering approach, especially for the completely visible buildings.

6.3 Stereoscopic Urban Visualization Using GPU

In this section, we will present the advantages of using GPU-based rendering in the context of stereoscopic visualization and its superiority to display list usage of OpenGL, as presented in the previous section.

The same test environment is used as in the previous section: Intel Pentium IV-3.4 GHz. computer with 4 GB of RAM and a NVidia Quadro Pro FX 4400 graphics card with 512 MB of memory supporting the quad buffering needed for stereoscopic visualization. The Crystal Eyes LCS glasses for viewing in stereo

were used. The purpose of the empirical study is to test:

- if single-location VFC brings an advantage over multiple VFC, given that since the enlarged frustum may decrease performance because of containing more polygons,
- GPU performance with the slice-wise building representation.

We performed tests using Vienna2000 Model, which is 7.8M polygons in 2,086 buildings, and a procedurally-generated city model composed of 23M polygons in 1,536 buildings with six different architectures. Still frames from navigations through these models are shown in Figure 6.7 and 6.8.

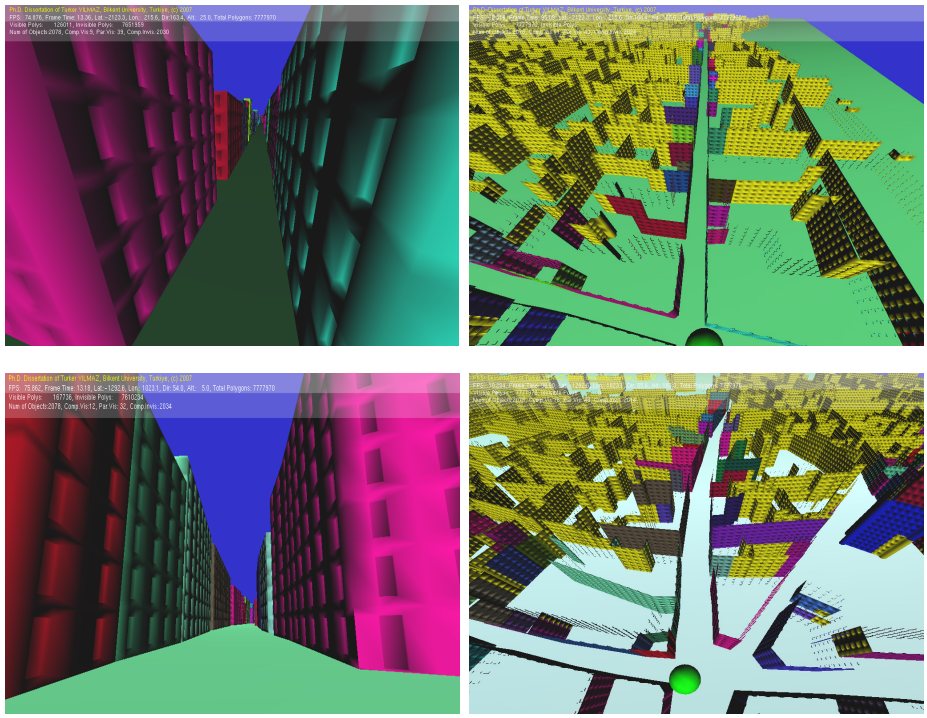


Figure 6.7: Still frames from navigations through the Vienna2000 model in monoscopic view using the GPU-based algorithm. On the left, still frames from a given viewpoint are shown. To the right of each frame, the view from above the user position represented by the green sphere, shows the rendered buildings using occlusion culling based on the slice-wise representation. Invisible buildings are shown in yellow.

In Figure 6.9, we compare the frame rates obtained by using different VFC schemes. Our aim is not to test the advantage of VFC but to test the gain

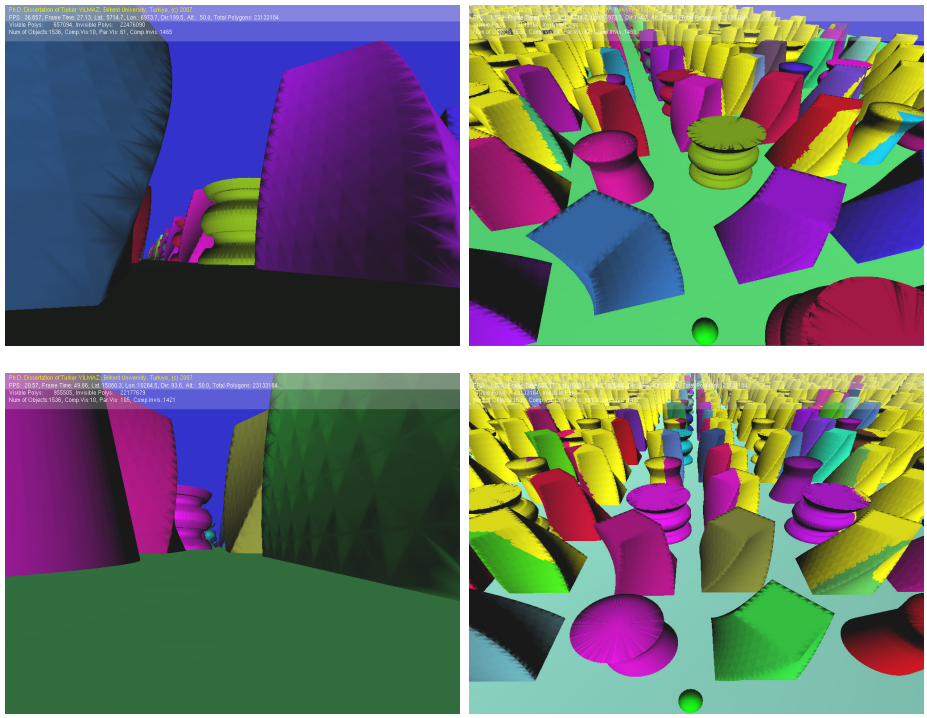


Figure 6.8: Still frames from navigations through the procedurally-generated model in monoscopic view using the GPU-based algorithm. On the left, still frames from a given viewpoint are shown. To the right of each frame, the view from above the user position represented by the green sphere, shows the rendered buildings using occlusion culling based on the slice-wise representation. Invisible buildings are shown in yellow.

in performance from using single-location VFC instead of multiple-location VFC. However for the sake of completeness we also give performances when VFC is not applied.

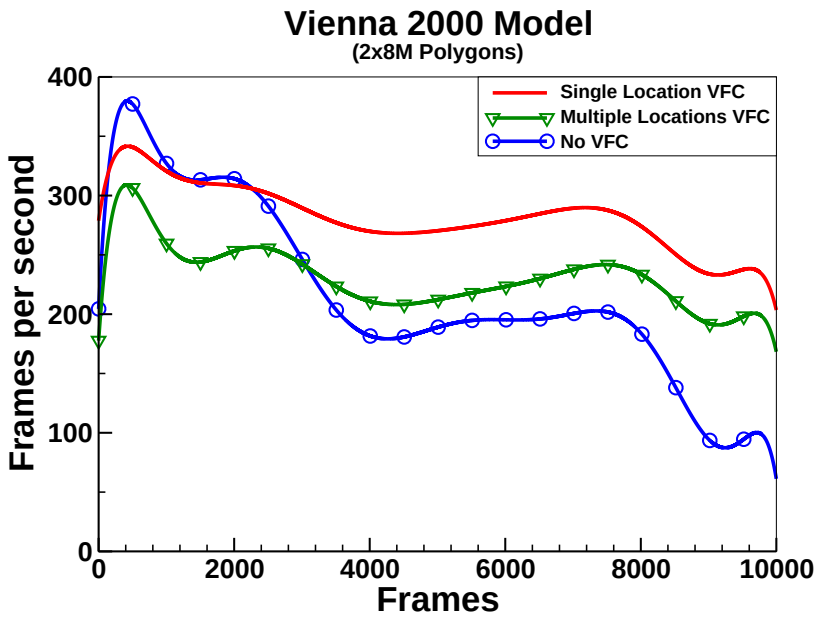
The average frame rates for the Vienna2000 Model are 281.8, 231.0 and 215.8 frames per second (fps) for the single-location, multiple-location and no frustum culling schemes, respectively. The average frame rates for the procedurally-generated model are 34.24, 30.5 and 10.2 frames per second (fps) for the single-location, multiple-location and no frustum culling schemes, respectively. The procedurally-generated model has long streets, which means a lot of geometry is instantly visible in each frame. The culling ratios are 98.53 %, 98.53 %, 96.43 % for the Vienna2000 Model and 97.00 %, 97.00 %, 91.82 % for the procedurally-generated model for the single-location, multiple-location, and no frustum culling

schemes, respectively including the occlusion culling ratios. Using single-location VFC with the Vienna2000 model produces a 22.0 % gain in frame rates when compared to using multiple location VFC; for the procedurally-generated model, the gain is 12.3 %.

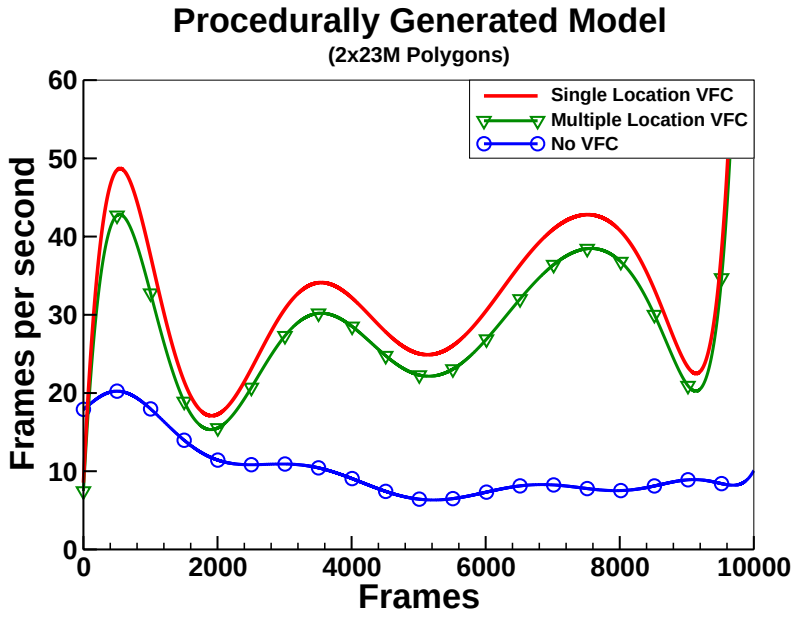
The advantage of using a GPU based rendering approach with the slice-wise building representation can be examined in two aspects: rendering speed-up and memory usage. The average frame rate for the monoscopic rendering of the Vienna2000 Model using OpenGL display lists is 135.1 fps (see Table 6.3). The frame rate for GPU-based stereoscopic rendering is 281 fps on the average. Since we render two images for each frame, this corresponds to a 315 % speed-up when compared to using OpenGL display lists. For the main memory, the usage for the slice-wise representation is 218.7MB. For the GPU-based approach, the memory usage is only 1.3MB (14 bytes per each of 94,480 slices). Thus, GPU-based rendering confers significant advantages both in terms of rendering speed and memory usage. Test results are summarized in Table 6.4.

Table 6.4: Summary of test results using the stereoscopic framework.

Model	Vienna 2000	Procedurally-Generated
No. polygons	7.8M	23M
No. buildings	2,086	1,536
No. slices	94,480	30,392
Main memory usage	1.3MB	425.5K
Single Location VFC	281.8 fps	34.24 fps
Multiple Location VFC	231.0 fps	30.5 fps



(a)



(b)

Figure 6.9: Frame rate comparison of the VFC schemes in stereoscopic visualization: (a) frame rates for the Vienna2000 model with 7.8M polygons. (b) frame rates for the procedurally-generated model with 23M polygons. These graphs show the advantage of using single-location VFC with respect to multiple location VFC and not performing VFC. Note that we render two images for each frame.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation, we presented a framework for stereoscopic visualization of large and complex urban environments. The framework consists of a navigable space extraction algorithm, which determines and cellulizes the navigable space in complex urban models; a powerful occlusion culling and an intelligent VFC algorithm, which eliminate most geometry that do not contribute to the user's view during navigation; and a GPU-based stereoscopic visualization approach, which provides smooth and real-time visualization of large urban models capable of rendering up to 46M polygons.

The occlusion culling algorithm makes use of the graphics hardware and incorporates a novel storage scheme, which exploits the visibility characteristics of buildings that can typically be experienced in a navigation through urban scenery. The proposed approach avoids sending a building entirely to the graphics pipeline if only a small portion of it becomes visible, thereby solving the partial occlusion problem. The objects are divided into axis-aligned slices and the slices rather than the whole objects are checked for occlusion.

We also showed how to shrink objects in a scene, including nonconvex ones, in order to use them as occluders for from-region conservative occlusion culling. Our shrinking algorithm can be used for any kind of objects, not just 2.5D buildings in

an urban scene. Our experiments showed that the proposed slice-wise occlusion culling provides significant increase in frame rates and decrease in the number of processed polygons as compared to a visualization using building-level occlusion culling. In addition, the slice-wise structure drastically reduces the PVS storage requirement.

The slice-wise structuring of objects can also be used to visualize scenes other than urban scenery, although we did not test this. Another application of our method would be scenes, where buildings are touching (as in some European cities). In this case, a subdivision at the object level could be done to create smaller objects as in [11, 107, 20]. Using visibility forms and the slice-wise representation is more useful for the static objects and decreasing the PVS storage costs. Occlusion culling done at the preprocessing stage cannot be easily applied to scenes containing dynamic objects.

The proposed approach works for flythrough-type navigations, where the user can be above buildings. Since, the occluded parts in the urban model become less as the flying altitude increases, it would be helpful for real time rendering to integrate our method with other approaches, such as view-dependent refinement.

The stereoscopic visualization consists of a rendering, which is based on the GPU architecture and makes use of the slice-wise representation. The framework also consists of a modified VFC approach, in which only one culling, instead of two, covers the necessary region for the two eye locations in the stereoscopic visualization. The resultant visibility list is rendered using a GPU-based algorithm, which perfectly fits into the proposed slice-wise representation.

The proposed algorithms were implemented on personal computers. The visualization was done using off-axis stereoscopic projection. Liquid crystal shutter (LCS) glasses for stereoscopic visualization were used. The framework was tested on several urban models ranging from 500K to 46M polygons. This study showed that;

- the visibility characteristics experienced in a typical navigation through urban models can be exploited for creating the tightest visibility list possible,
- the proposed hardware-based occlusion culling method and the usage of the

slice-wise representation increase the performance by 81 % as compared to occlusion culling using building-level granularity,

- the GPU-based method increase it by an additional 315 % in frame rates over the one using OpenGL display lists,
- the single-location VFC brings 22 % performance gain over multiple-location VFC,
- large urban models can be rendered and visualized at real-time by eliminating the invisible parts of the model at great ratios,
- scalability of using a preprocessed algorithm for large urban models can be achieved due to storage requirements,
- fast production of city models obeying the real locations of the buildings can be done automatically.

This study showed that a real time stereoscopic visualization of urban scenes can be achieved using the proposed framework.

7.2 Future Work

The navigable space algorithm is robust enough that can handle any type of buildings in a city model. However, the creation algorithm is slow in terms that it checks each triangle and the seed one by one, in a brute force manner. The creation speed can be increased by using a knowledge about the structure, such as neighboring information of the triangles.

The slice-wise representation perfectly fits into the GPU-architecture. During the occlusion culling process, the slices are checked for occlusion. The resultant list can further be tightened, if real triangles belonging to the slices are checked in a way, so that the slice portions without triangles are not accepted as visible during the occlusion culling process. The common triangles belonging to many slices, which are accepted as visible, are drawn multiple times. If we try to eliminate those parts, which we did so in Chapter 4, then we need to use OpenGL display list mechanism. However, using GPU-based rendering is much faster as shown

in Chapter 5, even if some triangles are rendered multiple times. If there were a way to determine if a primitive has been rendered just one and fast check, i.e. a tag bit in the vertex shader of the GPU, then the rendering speed may increase further.

The calculated PVS and the urban model itself currently must be loaded into the memory completely. A spatial database access would be very suitable, provided that no assumptions are made on the visibility, but only necessary models are loaded.

For the stereoscopic rendering, we only made improvements on the VFC operations. Another problem is the ghosting effect, which was described in Section 2.4.2.7. We have made experiments especially for the peripheral ghosting effect removal and succeeded up to a degree. We removed respective positions and masked them by writing a few fragment shaders in the GPU and also tried the same process by using additional clipping planes. The use of the fragments shaders did not give satisfactory results because of the perspective projection. The use of the clipping planes removed the peripheral ghosting effect. However, since the objects behind the removed parts became visible due to conservative occlusion culling, this process introduced additional ghost parts. These parts can be removed by developing additional techniques, which hide them from the viewer. For the usage of the GPU in rendering, more detailed and complex shaders can be integrated to achieve more realistic rendering. Their usage is vitally important for large scale rendering, especially if the buildings have transparency and reflection properties and the scene consists of many light sources.

For the automatic city modeling approach, we plan to generate class libraries that are capable of modeling different styles of architectural constructions. Then, it will be possible to model cities in a more realistic way. We should emphasize that our aim is not to model the cities by the way that remote sensing techniques do. Our approach enables the production of buildings that are more realistic as compared to the ones in city models obtained by remote sensing techniques. It is also possible to incorporate different data by using an adaptor for GML-the Geography Markup Language [29], which is capable of representing more information than a single DXF file can hold.

Appendix A

City Modeling

In this appendix, we present a method for the automatic generation of building models to be used in virtual city models. The models produced by this approach can easily be used in our implementation of stereoscopic urban visualization framework. Since the City Modeling is not a part but a feature of the stereoscopic urban visualization framework and not fully completed, we preferred to present it in this appendix.

The building generation process incorporates randomness and it can be steered by the help of derivation rules and assigned attributes. The derivation method is inspired by the shape grammars. During the derivation process, floor plans of the actual cities are used to generate 3D city models. Given the city plans, derivation rules and definitions of some basic objects, the system generates 3D building models, which are used to populate city models.

A.1 Introduction

In order to create virtual cities each building should be modeled separately. Modeling each and every building in detail manually is a tedious process. Even the use of aerial images or airborne laser scan data requires a great deal of manual post-processing.

Most countries have ground plans of the actual cities in digital format. There has been significant amount of research in the last years about generating 3D models using 2D ground plans and visualizing these models. For instance, Google released a geographical visualization system named Google Earth that combines the satellite photos with the 3D models obtained using city plans to generate 3D city models. Currently generated city models consist of only a few cities, mostly in the USA.

The original motivation for this work is to generate 3D city models using 2D city plans consistent with the real shapes of the buildings as much as possible and to visualize the city models in real time, [76, 75]. City ground plans are used to produce city models. This is accomplished by generating every building using its 2D ground plan. Buildings can be produced either deterministically or stochastically by using building footprints and shape grammars. The objects are first separated into subparts according to the pre-defined rules and the building model is produced from the final objects, such as walls, windows, balconies, etc. The shapes of the buildings are defined by specifying the rule set and the initial building models on which the rule set is to be applied. Deterministic building production applies fixed rules that split an object recursively to a fixed number of rows and columns whose sizes are defined in the rule. Stochastic building production uses random rules to split an object into a 2D grid that is composed of randomly placed rows and columns.

The produced building models are then represented using a slice-wise representation to facilitate the visualization process. The cells that belong to a building in a uniform subdivision of space are determined and these cells are then clustered into axis aligned slices to provide better granularity for the visibility calculations as compared to the visibility calculations where the granularity is individual buildings. The city models are visualized in real time by using the visualization system.

In the next section, we describe the proposed shape grammar-based approach to building model production.

A.2 Building Model Production

Our work is inspired by the split grammars and assumes that the buildings are consists of a number of facades that are vertical to the ground plane. Same with the split grammars, model generation process incorporates randomness to achieve higher variety of building models. Whereas split grammars splits and transforms 3D shapes to generate building models, the proposed system works with 2D planar shapes. This comes with a limitation such that buildings other that the ones with only vertical facades can not be generated. The fact that temporary shapes are simply planar surfaces in 3D and major details of the generated building model such as windows, balconies and doors are completely defined by the terminal shapes, limits the variety of the building models that can be generated.

On the other hand, with the proposed approach it is easier to implement the generation system and work with the rules. Since there are only two types of split rules in our work, namely random split and fixed split, which are both very simple, and one type of transformation rule, which simply replaces a temporary shape with a terminal shape, it easier to understand the generation process and so design the rule sets.

The system uses "data exchange format" (DXF) of AutoCAD as the input and output file format. DXF format is a very popular 3D model format since it is very simple, standardized and accepted by the community. DXF format does not include texture information. However, it is possible to add layers to store texture information. Building plans specify the number of floors and the placement of cells and portals for each floor. The city plans that are composed of building plans are given as input to the system and the system extracts individual building floor plans to generate buildings (Figure A.1).

Produced building models are composed of several facades, one for each edge of the floor plan. Each edge of the floor plan is handled at a time. A facade that corresponds to an edge of the floor plan is composed of a number of floors. A facade could be composed of multiple copies of the same type of floor or different type of floors.

In the facade derivation process, a facade is split into floors that are actually two

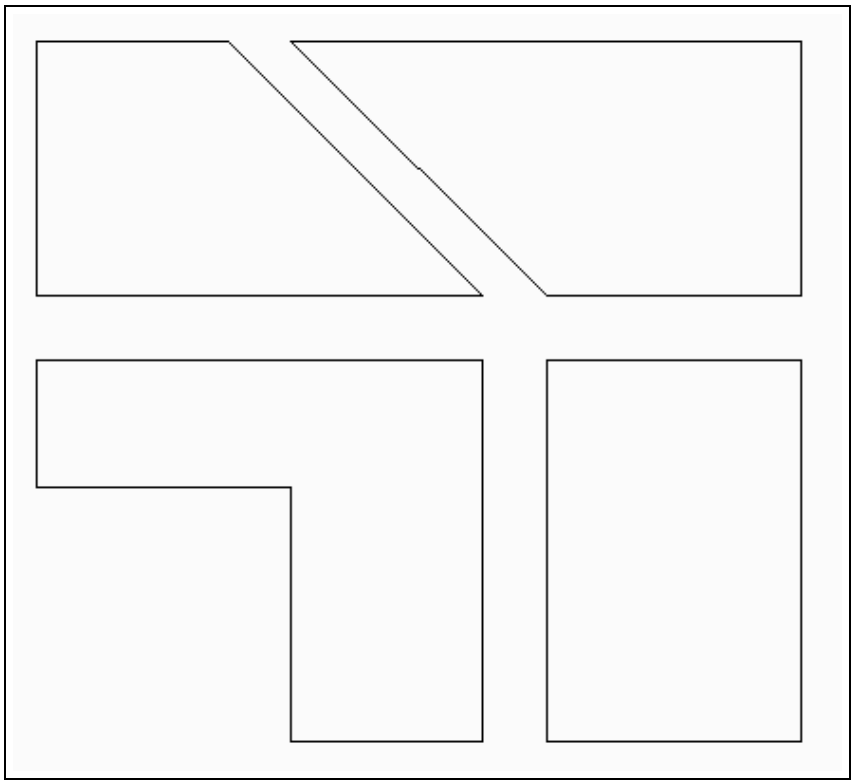


Figure A.1: A simple portion of a city plan. Since the system generates each facade of a building at a time by handling each edge of the building floor plan at a time, floor plans could be any kind of polygon.

dimensional floor face objects. Then, predefined split rules are applied to these floor objects to generate new two dimensional temporary objects (Figure A.2). This process continues until all paths end to a terminal object. All terminal objects, such as windows, walls or balconies, are predefined in DXF format. During the facade generation process, theseterminal objects are scaled, oriented, translated and output to the model file in DXF format. A building is generated when all the edges of the floor plan are transformed to a corresponding facade.

A.3 Shapes

Currently, there are two types of objects defined in the system, which can be increased with the help of the derivation rule designs:

- *Terminal shapes:* these are the basic shapes designed by a 3D design program and stored in DXF format. Terminal shapes are composed of any number of planar surfaces and have unit dimensions to be easily scaled (Figure A.3). Texture information for the terminal shapes are stored as the layers in DXF format.
- *Temporary shapes:* these are the shapes that are split into other temporary shapes or terminal shapes by the rules defined in the configuration file in XML format. The derivation process is initiated by a floor object. Temporary shapes contain various attributes that are used to control the splitting process. These attributes eliminate the rules that do not apply to the temporary object. Temporary shapes are simply represented as rectangles in 3D.

A.4 Rules

Temporary objects are split by the rules until all the objects become terminal objects. The derivation process for a building model is steered by the set of rules that applies only to it. The system deduces its behavior based on the rule properties. An arbitrary number of attributes are attached to each rule. These attributes play an important role in the split rule selection for a temporary object. Within the set of the rules that applies to a temporary object, only the rules that have attributes of appropriate values could be applied to the object and one rule is selected randomly among them. Furthermore, we could give higher probabilities to some rules if they are to be selected more frequently. Alternatively, we can use weights based on previous rule usage statistics to favor the rules that are frequently used. There are two types of split rules: *random split* and *fixed split*.

A.4.1 Random Split

Given a temporary shape, a random rule splits the object into a 2D grid that is composed of randomly placed rows and columns. Minimum width of a column and minimum height of a row are given by the rule. All rows are of $[minHeight, 2*$

$minHeight$], whereas all columns are of $[minWidht, 2 * minWidth]$ at the end. In a random split, all of the created objects can be of the same shape. Each new object can be a terminal shape or a temporary shape. An instance of random split is shown in Figure A.4.

A.4.2 Fixed Split

Fixed rules split the given object to a fixed number of rows and columns whose sizes are defined in the rule. When a terminal object is split by a fixed split, the sizes of the rows and columns are defined as directly proportional to the width and height of the terminal shape. When a fixed rule is applied to a temporary shape, the number of newly created shapes is fixed. The type and attributes of every newly created shape are defined in the rule. An instance of fixed split is shown in Figure A.5.

With the help of the proposed building production, it becomes very easy to generate a whole city provided that the ground plans are given. Since, we are not able to get a complete city plan, we made use of several other models which range from 500K to 46M polygons. These models are Vienna2000, Glasgow and two of our procedurally generated model. Especially Vienna2000 and Glasgow models are publicly available and we are able to compare our work with the previous state of the art by the help of these models.

A.5 Results and Discussion

Building models produced by using only height information and floor plans neither have enough details nor reflect the architectural style of the actual building. The method presented in this appendix allows fast generation of building models that reflect the intended architectural style. Figure A.6 shows a portion of Istanbul Historical Peninsula rendered by using only height information and floor plans. Although the shading and coloring is used, the view is not realistic enough since the basic building blocks (windows, balconies, etc.) could not be modeled.

Building models generated by using our approach are shown in Figures A.7 and A.8. In our approach, the floors of a building are not necessarily of the same type. A building model covered with textures is shown in Figure A.9. Generated models could then be used to populate virtual cities that can be navigated in real time by the visualization system.

We present a system that is capable of producing building models to be used for populating virtual cities. Buildings are produced both deterministically and stochastically by means of footprints and shape grammars. In the production process, the objects are first separated into subparts according to the pre-defined rules and the building model is created from the final objects, which are in DXF format. The architecture of a virtual building depends on the architecture of the predefined (initial) objects and the rule set that will be applied during production.

All the rules are stored in a configuration file in XML format. The generation of a building model with ten floors and 200 windows takes less than ten seconds on an Intel Centrino with 1.6 GHz. In the case of larger rule sets, storing the rules in a database would improve the generation time.

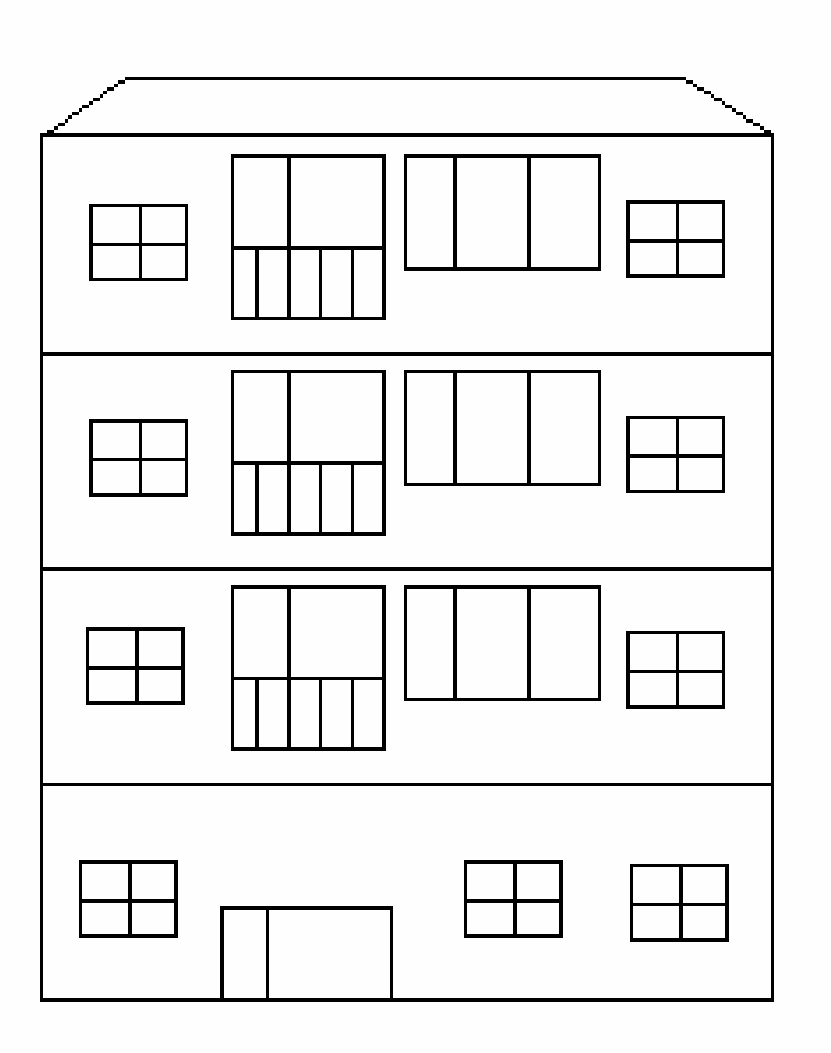


Figure A.2: A building facade that composed of same type of floors. It should be noted that this is not necessarily always the case.

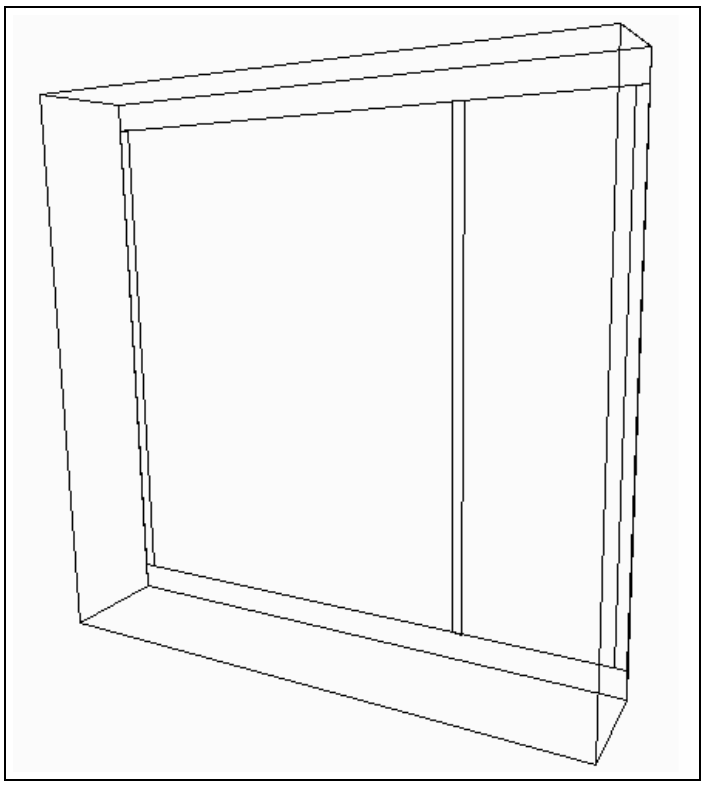


Figure A.3: A very basic terminal shape that could stand for a window.

```
<Floor>
     <Split Balcony="+" Window="+">
     <Random minWidth="2" minHeight="3">
     <Face Balcony="+" Window="+"></Face>
     </Random>
     </Split>
     ...
</Floor>
```

Figure A.4: A random split rule named as “Split”. It is defined to split the temporary object named “Floor”. “Face” is the object that is formed when this rule is applied; it could be a terminal shape or another temporary shape. It should be noted that there could be other rules defined between the tags <Floor> and </Floor>, which are the rules applied to the object named “Floor”.

```
<Face>
<Window Balcony="+" Window="+">
    <Fixed>
    <xProportions x1="2" x2="4" x3="2"></xProportions>
    <yProportions y1="3" y2="4" y3="3"></yProportions>
        <Elements>
            <Wall></Wall>
            <Wall></Wall>
            <Wall></Wall>
            <Wall></Wall>
            <Window></Window>
            <Wall></Wall>
            <Wall></Wall>
            <Wall></Wall>
            <Wall></Wall>
        </Elements>
    </Fixed>
</Window>
<Balcony>
    <Fixed>
    <xProportions x1="1" x2="4" x3="1"></xProportions>
    <yProportions y1="4" y2="1"></yProportions>
        <Elements>
            <Wall></Wall>
            <Wall></Wall>
            <Balcony></Balcony>
            <Wall></Wall>
            <Wall></Wall>
            <Wall></Wall>
        </Elements>
    </Fixed>
</Balcony>
...
</Face>
```

Figure A.5: Two simple fixed split rules defined for the shape “Face”, named “Window” and “Balcony”. The proportions of the size of the rows and columns that are to be formed are defined as attributes. The children of the <Element> tag are the shapes created when these split rules are applied, which could be terminal shapes or temporary shapes again.

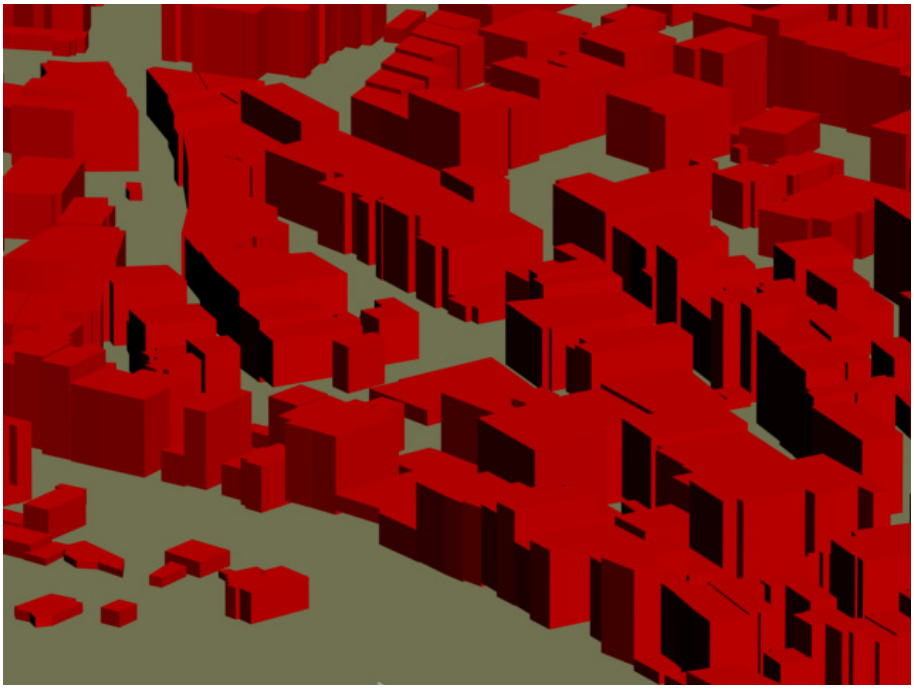


Figure A.6: A portion of İstanbul Historical Peninsula produced by using height information and floor plans.

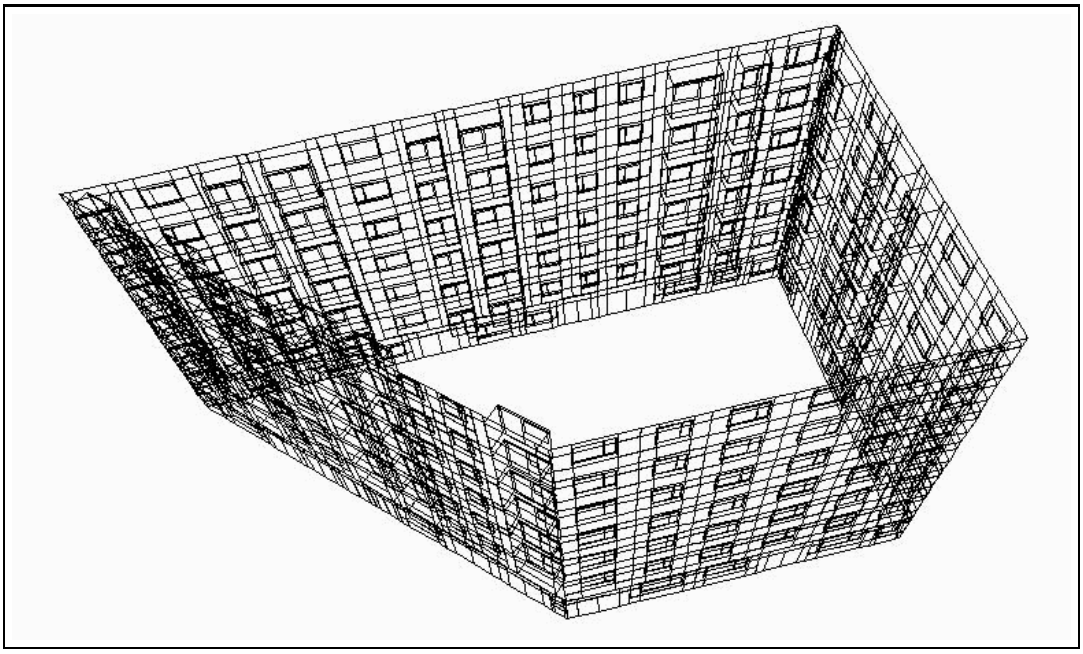


Figure A.7: A building model generated using the proposed automatic building modeling method.

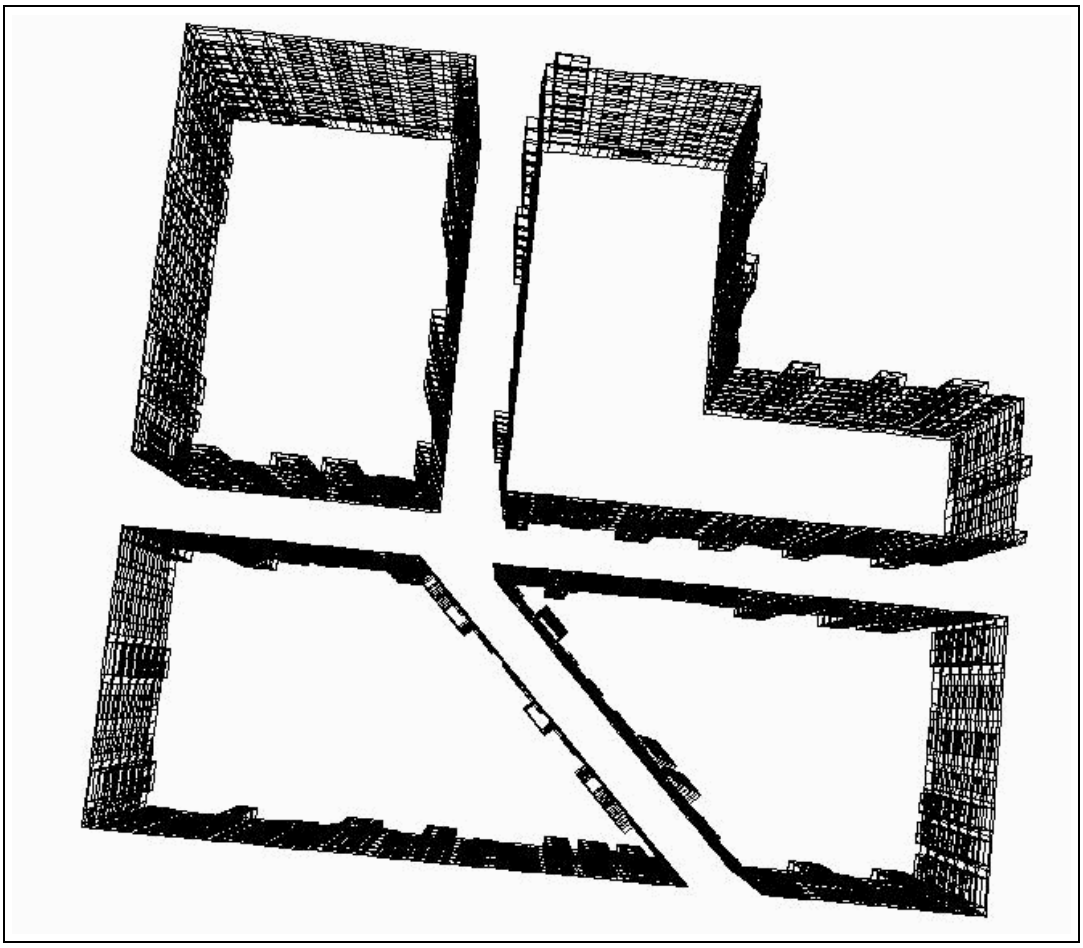


Figure A.8: A block of four buildings.

Figure A.9: Two views of a building model covered with textures.

Bibliography

[1] S. J. Adelson, J. B. Bentley, I. S. Chong, L. F. Hodges, and J. Winograd. Simultaneous generation of stereoscopic views. *Computer Graphics Forum*, 10(1):3–10, 1991.

[2] S. J. Adelson and C. D. Hansen. Fast stereoscopic images with ray traced volume rendering. In *Proceedings of Symposium on Volume Visualization*, pages 3–9, 1994.

[3] S. J. Adelson and L. F. Hodges. Stereoscopic ray-tracing. *The Visual Computer*, 10(3):127–144, 1993.

[4] P. K. Agarwal and M. Sharir. Arrangements. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier Science Publishers B.V., North-Holland, Amsterdam, 1999.

[5] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: an interactive massive model rendering system using geometric and image-based acceleration. In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 199–206, 1999.

[6] C. Andújar, C. Saona-Vázquez, and I. Navazo. LOD visibility culling and occluder synthesis. *Computer Aided Design*, 32(3):773–783, 2000.

[7] C. Andújar, C. Saona-Vázquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of detail through hardly-visible sets. *Computer Graphics Forum*, 19(3):499–506, 2000.

[8] B. Aronov, H. Brönnimann, A. Y. Chang, and Y. Chiang. Cost-driven octree construction schemes: An experimental study. In *Proceedings of 19th Annual ACM Symposium on Computational Geometry*, pages 227–236, 2003.

[9] C. L. Bajaj, V. Pascucci, and G. Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *Proceedings of IEEE Visualization*, pages 307–316, 1999.

[10] D. Bartz, M. Meißner, and T. Hüttner. OpenGL-assisted occlusion culling for large polygonal models. *Computers & Graphics*, 23(5):667–679, 1999.

[11] W. V. Baxter III, A. Sud, N. K.Govindaraju, and D. Manocha. Gigawalk: Interactive walkthrough of complex environments. In *Proceedings of 13th Eurographics Workshop on Rendering*, pages 203–214, 2002.

[12] P. N. Belhumeur and D. Mumford. A bayesian treatment of the stereo correspondence problem using half-occluded regions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 506–512, 1992.

[13] J. Bittner, J. Prikryl, and P. Slavik. Exact regional visibility using line-space partitioning. *Computers & Graphics*, 27(4):569–580, 2003.

[14] J. Bittner, P. Wonka, and M. Wimmer. Visibility preprocessing for urban scenes using line space subdivision. In *Proceedings of Pacific Conference on Computer Graphics and Applications*, pages 276–284, 2001.

[15] J. Bittner, P. Wonka, and M. Wimmer. Fast exact from-region visibility in urban scenes. In *Proceedings of the 15th Eurographics Workshop on Rendering Techniques*, pages 223–230, 2005.

[16] A. F. Bobick and S. S. Intille. Large occlusion stereo. *International Journal of Computer Vision*, 33(3):181–200, 1999.

[17] P. Bos. Time sequential stereoscopic displays: The contribution of phosphor persistence to the 'ghost' image intensity. In *Proceedings of the International Training and Education Conference ITEC'91 (Three-Dimensional Image Technologies)*, pages 603–606, 1991.

[18] G. Boström, M. Fiocco, D. Puig, A. Rossini, J. Gonçalves, and V. Sequeira. Acquisition, modelling and rendering of very large urban environments. In *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'04)*, 2004.

[19] B. Chen, J. E. Swan, E. Kuo, and A. E. Kaufman. LOD-sprite technique for accelerated terrain rendering. In *Proceedings of IEEE Visualization*, pages 291–298, 1999.

[20] J. Chhugani, B. Purnomo, S. Krishnan, J. Cohen, S. Venkatasubramanian, and D. S. Johnson. vLOD: High-fidelity walkthrough of large virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 11(1):35–47, 2005.

[21] Y.-Y. Chuang and M. Ouhyoung. Clustering and visibility preprocessing of hierarchical radiosity for object-based environment. In *Proceedings of the Computer Graphics Workshop*, pages 102–111, 1995.

[22] S. Co. *Stereographics Developer's Handbook*. Stereographics Corporation, 1997.

[23] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. In *Proceedings of SIGGRAPH*, pages 119–128, 1996.

[24] D. Cohen-Or, Y. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[25] D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–254, 1998.

[26] S. Coorg and S. J. Teller. Temporally coherent conservative visibility. In *Proceedings of the ACM Symposium on Computational Geometry*, pages 78–87, 1996.

[27] S. Coorg and S. J. Teller. Real-time occlusion culling for models with large occluders. In *Proceedings of the ACM Symposium on Interactive 3D Graphics*, pages 83–90, 1997.

[28] I. J. Cox, S. Hingorani, S. Rao, and B. Maggs. A maximum-likelihood stereo algorithm. *Computer Vision and Image Understanding*, 63(3):542–567, 1996.

[29] S. Cox, P. Daisey, R. Lake, C. Portele, and A. Whiteside. OpenGIS Geography Markup Language (GML) Encoding Specification (3.0), 2003.

[30] L. Darsa, B. Costa, and A. Varshney. Walkthroughs of complex environments using image-based simplification. *Computers & Graphics*, 22(1):55–69, 1998.

[31] D. Davis, W. Ribarsky, T. Y. Jiang, N. Faust, and S. Ho. Real-time visualization of scalably large collections of heterogeneous objects. In *Proceedings of IEEE Visualization*, pages 437–440, 1999.

[32] X. Decoret, G. Debunne, and F. Sillion. Erosion based visibility preprocessing. In P. Christensen and D. Cohen-Or, editors, *Proceedings of the 14th Eurographics Workshop on Rendering*, pages 281–288, 2003.

[33] X. Decoret, F. Sillion, G. Schaufler, and J. Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3):61–73, 1999.

[34] X. Decoret and F. X. Sillion. Street generation for city modeling. In *Architectural and Urban Ambient Environments*, 2002.

[35] L. Downs, T. Möller, and C. H. Séquin. Occlusion horizons for driving through urban scenes. In *Proceedings of SIGGRAPH*, pages 121–124, 2001.

[36] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. In *Proceedings of SIGGRAPH*, pages 239–248, 2000.

[37] J. A. El-Sana, N. Sokolovsky, and C. T. Silva. Integrating occlusion culling with view-dependent rendering. In *Proceedings of IEEE Visualization*, pages 371–378, 2001.

[38] C. Erikson, D. Manocha, and W. V. Baxter. HLODs for faster display of large static and dynamic environments. In *Proceedings of SIGGRAPH*, pages 111–120, 2001.

[39] J. D. Ezell and L. F. Hodges. Some preliminary results on using spatial locality to speed-up raytracing of stereoscopic images. In *Proceedings of the International Society for Optical Engineering (SPIE) 1256, Stereoscopic Displays and Applications I*, pages 298–306, 1990.

[40] C. Früh and A. Zakhor. Constructing 3D city models by merging aerial and ground views. *IEEE Computer Graphics and Applications, Special Issue on 3D Reconstruction and Visualization*, pages 52–61, 2003.

[41] T. A. Funkhouser, C. H. Sequin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. *ACM Computer Graphics (Proceedings of ACM Symposium on Interactive 3D Graphics)*, 25(2):11–20, 1992.

[42] D. Geiger, B. Ladendorf, and A. L. Yuille. Occlusions and binocular stereo. *International Journal of Computer Vision*, 14(3):211–226, 1995.

[43] C. Gotsman, O. Sudarsky, and J. A. Fayman. Optimized occlusion culling using five-dimensional subdivision. *Computers & Graphics*, 23(5):645–654, 1999.

[44] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 325–336, New York, NY, USA, 2006. ACM Press.

[45] N. K. Govindaraju, M. C. Lin, and D. Manocha. Fast and reliable collision culling using graphics hardware. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):143–154, 2006.

[46] N. Greene. Hierarchical Z-buffer visibility. In *Proceedings of SIGGRAPH*, volume 27, pages 231–238, 1993.

[47] N. Greene. Efficient occlusion culling for Z-buffer systems. In *Proceedings of SIGGRAPH*, pages 78–79, 1999.

[48] N. Greene. Occlusion culling with optimized hierarchical buffering. In *Proceedings of SIGGRAPH (Sketches & Applications)*, pages 261–261, 1999.

[49] U. Güdükbay and T. Yılmaz. Stereoscopic view–dependent visualization of terrain height fields. *IEEE Transactions on Visualization and Computer Graphics*, 8(4):330–345, 2002.

[50] T. Haven. A liquid-crystal video stereoscope with high extinction ratios, a 28% transmission state, and 100 μs switching. In *Proceedings of the International Society for Optical Engineering (SPIE)*, volume 761, pages 23–26, 1987.

[51] T. He and A. Kaufman. Fast stereo volume rendering. In *Proceedings of IEEE Visualization*, pages 49–56, 1996.

[52] J. Heo, J. Kim, and K. Wohn. Conservative visibility preprocessing for walkthroughs of complex urban scenes. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 115–128, 2000.

[53] L. F. Hodges. Tutorial: Time-multiplexed stereoscopic computer graphics. *IEEE Computer Graphics & Applications*, 12(2):20–30, 1992.

[54] L. F. Hodges and D. McAllister. Stereo and alternating-pair techniques for display of computer-generated images. *IEEE Computer Graphics & Applications*, 5(9):38–45, 1985.

[55] J. Hu, S. You, and U. Neumann. Approaches to large-scale urban modeling. *IEEE Computer Graphics and Applications, Special Issue on 3D Reconstruction and Visualization of Large Scale Environments*, 23(6):62–69, 2003.

[56] R. Hubbold, D. Hancock, and C. Moore. Stereoscopic volume rendering. In *Proceedings of Visualization in Scientific Computing*, pages 105–115, 1998.

[57] T. Hüttner, M. Meißner, and D. Bartz. Opengl-assisted visibility queries of large polygonal models. Technical Report WSI-98-6, Dept. of Computer Science (WSI), University of Tübingen, 1998.

[58] S. S. Intille and A. F. Bobick. Disparity-space images and large occlusion stereo. In *Proceedings of the European Conference on Computer Vision*, volume B, pages 179–186, 1994.

[59] J. T. Klosowski and C. T. Silva. Rendering on a budget: A framework for time-critical rendering. In *Proceedings of IEEE Visualization*, pages 115–122, 1999.

[60] J. T. Klosowski and C. T. Silva. The prioritized–layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108–123, 2000.

[61] J. T. Klosowski and C. T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365–379, 2001.

[62] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Virtual occluders: An efficient intermediate PVS representation. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, pages 59–70. Springer-Verlag, 2000.

[63] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Hardware-accelerated from-region visibility using a dual ray space. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 205–216, London, UK, 2001. Springer-Verlag.

[64] S. Kumar, D. Manocha, W. Garrett, and M. Lin. Hierarchical back-face computation. *Computers & Graphics*, 23(5):681–692, 1999.

[65] A. Lefohn. Glift: an abstraction for generic, efficient GPU data structures. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 140, New York, NY, USA, 2005. ACM Press.

[66] T. Leyvand, O. Sorkine, and D. Cohen-Or. Ray space factorization for from-region visibility. *ACM Trans. Graph.*, 22(3):595–604, 2003.

[67] J. Lipscomb and W. Wooten. Reducing crosstalk between stereoscopic views. In *Proceedings of the International Society for Optical Engineering (SPIE)*, volume 2177, pages 92–96, 1994.

[68] L. Lipton. Binocular symmetries as criteria for the successful transmission of images. In *Proceedings of the International Society for Optical Engineering (SPIE) (Processing and Display of Three-Dimensional Data II)*, volume 507, 1984.

[69] L. Lipton. Factors affecting ghosting in a time-multiplexed plano-stereoscopic CRT display system. In *Proceedings of the International Society for Optical Engineering (SPIE)*, volume 761, 1987.

[70] L. Lipton, J. Halnon, J. Wuopio, and B. Dorworth. Eliminating π-cell artifacts. In *Proceedings of the International Society for Optical Engineering (SPIE)*, volume 3957, pages 264–270, 2000.

[71] W. J. MITCHELL. *The Logic of Architecture: Design, Computation, and Cognition.* MIT Press, 1990.

[72] S. Nirenstein and E. Blake. Hardware accelerated visibility preprocessing using adaptive sampling. In *Proceedings of the Eurographics Symposium on Rendering*, pages 207–216, 2004.

[73] S. Nirenstein, E. Blake, and J. Gain. Exact from-region visibility culling. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 191–201, 2002.

[74] nVidia White Paper. Using vertex buffer objects (VBOs). *nVidia Corperation*, pages 1–15, 2003.

[75] O. Oğuz, M. E. Aran, T. Yılmaz, and U. Güdükbay. Automatic production and visualization of urban models from building allocation plans. In *Proceedings of the Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'06)*, 2006.

[76] O. Oğuz, M. E. Aran, T. Yılmaz, and U. Güdükbay. Bina tahsis planlarından 3-boyutlu şehir modellerinin üretilmesi ve görüntülenmesi (in Turkish). In *IEEE Sinyal İşleme ve Uygulamaları Kurultayı (SIU'06)*, 2006.

[77] OpenSG Forum. *OpenSG – Open Source Scene Graph.* http://www.opensg.org, 2000.

[78] M. V. D. Panne and A. J. Stewart. Efficient compression techniques for precomputed visibility. In *Proceedings of Eurographics Symposium on Rendering*, pages 306–316, 1999.

[79] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of SIGGRAPH*, pages 301–308, 2001.

[80] J. Popović and H. Hoppe. Progressive simplicial complexes. In *Proceedings of SIGGRAPH*, pages 217–224, 1997.

[81] J. Rossignac. Geometric simplification and compression in multiresolution surface modeling. In *SIGGRAPH Course Notes #25*, 1997.

[82] F. Rottensteiner. Automatic generation of high-quality building models from LIDAR data. *IEEE Computer Graphics and Applications, Special Issue on 3D Reconstruction and Visualization*, 23(6):42–50, 2003.

[83] H. Samet. The quadtree and related data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[84] C. Saona-Vázquez, I. Navazo, and P. Brunet. The visibility octree: a data structure for 3D navigation. *Computers & Graphics*, 23(5):635–643, 1999.

[85] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of SIGGRAPH*, pages 229–238, 2000.

[86] D. Schmalstieg and R. F. Tobler. Exploiting coherence in 2.5-D visibility computation. *Computers & Graphics*, 21(1):121–123, 1997.

[87] M. Schmitt and J. Mattioli. *Morphologie Mathématique.* Masson, Paris, 1993.

[88] F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering)*, 16(3):207–218, 1997.

[89] D. Staneker, D. Bartz, and W. Straßer. Occlusion culling in OpenSG PLUS. *Computers & Graphics*, 28:87–92, 2004.

[90] G. Stiny. *Introduction to shape and shape grammars.* Environment and Planning, 1980.

[91] S. J. Teller. *Visibility Computations in Densely Occluded Environments.* PhD thesis, University of California, Berkeley, 1992.

[92] S. J. Teller and C. H. Sequin. Visibility preprocessing for interactive walkthroughs. In *Proceedings of SIGGRAPH*, volume 25, pages 61–69, 1991.

[93] N. A. Valyus. *Stereoscopy.* Focal Press, 1962.

[94] G. Varadhan and D. Manocha. Accurate minkowski sum approximation of polyhedral models. In *Proceedings of the Pacific Conference on Computer Graphics and Applications*, pages 392–401. IEEE Computer Society, 2004.

[95] M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Straßer. The randomized z–buffer algorithm: Interactive rendering of highly complex scenes. In *Proceedings of SIGGRAPH*, pages 361–370, 2001.

[96] J.-G. Wang, E. Sung, and R. Venkateswarlu. Estimating the eye gaze from one eye. *Computer Vision and Image Understanding*, 98(1):83–103, 2005.

[97] C. Wheatstone. On some remarkable, and hitherto unobserved, phenomena of binocular vision. *Philosophical Transactions of the Royal Society of London*, 128:371–394, 1838.

[98] M. Wimmer, M. Giegl, and D. Schmalstieg. Fast walkthroughs with image caches and ray casting. *Computers & Graphics*, 23(6):831–838, 1999.

[99] P. Wonka and D. Schmalstieg. Occluder shadows for fast walkthroughs of urban environments. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering)*, 18(3):51–60, 1999.

[100] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Proceedings of Rendering Techniques*, pages 71–82, 2000.

[101] P. Wonka, M. Wimmer, and F. X. Sillion. Instant visibility. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering)*, 20(3):411–421, 2001.

[102] P. Wonka, M. Wimmer, F. X. Sillion, and W. Ribarsky. Instant architecture. In *Proceedings of SIGGRAPH*, 2003.

[103] A. J. Woods and S. S. L. Tan. Characterising sources of ghosting in time-sequential stereoscopic video displays. In *Proceedings of the International Society for Optical Engineering (SPIE)*, volume 4660, pages 66–77, 2003.

[104] T. Yılmaz. Fast stereoscopic view–dependent visualization of terrain height fields. Master's thesis, Bilkent University, Department of Computer Engineering, Ankara, Turkey, 2001.

[105] T. Yılmaz and U. Güdükbay. Extraction of 3D navigation space in virtual urban environments. In *Proceedings of the 13th European Signal Processing Conference*, 2005.

[106] T. Yılmaz and U. Güdükbay. Conservative occlusion culling for urban visualization using a slice-wise data structure. *Graphical Models (to appear)*, 2007.

[107] S.-E. Yoon, B. Salomon, and R. Gayle. Quick-VDR: Out-of-core view-dependent rendering of gigantic models. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):369–382, 2005.

[108] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of SIGGRAPH*, pages 77–88, 1997.

[109] Z. Zhu and Q. Ji. Robust real-time eye detection and tracking under variable lighting conditions and various face orientations. *Computer Vision and Image Understanding*, 98(1):124–154, 2005.