Analysis of Urban Morphology for Real Time Visualization of Urban Scenes

by

Sami Mohammed Shalabi

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 28, 1998

Author .....
Department of Electrical Engineering and Computer Science
May 18, 1998

Certified by .....
Julie Dorsey
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Analysis of Urban Morphology for Real Time Visualization of Urban Scenes

by

Sami Mohammed Shalabi

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 1998, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Much research in real-time visualization has been devoted to general techniques that either simplify geometry or effectively extract the currently visible geometric dataset. However, little work has been devoted to understanding the environments being visualized and using the information about their structure. We describe the design and re-implementation of a system used to visualize urban environments based on the notion of using the structure embedded in these environments. This methodology is based on a hybrid approach that uses traditional 3D techniques to render geometry near the viewer and pseudo-geometry generated from images to render the far geometry[SDB97].

Designing a system for the visualization of urban scenery is challenging because of the great complexity of these environments. Typical views of urban scenes contain rich visual details at a fairly small scale, while the extent of the model is often large. To address the complexity, we have introduced a new set of data structures, motivated by urban morphology, to deal with and manage it. An expandable system was developed to interactively analyze, build and modify the data structures associated with the visualization.

Urban scenes are heavily structured [Lyn60]. To take advantage of the structure, we perform an analysis of the urban environment, develop algorithms to extrapolate urban features, include an implementation that deals with a fixed set of urban conditions, and finally propose a treatment that uses the structure more concretely. We have developed a system to generate synthetic urban environments and successfully used our system to visualize these synthetic environments, as well as a modeled portion of Paris.

Thesis Supervisor: Julie Dorsey
Title: Associate Professor

Analysis of Urban Morphology for Real Time Visualization of Urban Scenes

by

Sami Mohammed Shalabi

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 1998, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Much research in real-time visualization has been devoted to general techniques that either simplify geometry or effectively extract the currently visible geometric dataset. However, little work has been devoted to understanding the environments being visualized and using the information about their structure. We describe the design and re-implementation of a system used to visualize urban environments based on the notion of using the structure embedded in these environments. This methodology is based on a hybrid approach that uses traditional 3D techniques to render geometry near the viewer and pseudo-geometry generated from images to render the far geometry[SDB97].

Designing a system for the visualization of urban scenery is challenging because of the great complexity of these environments. Typical views of urban scenes contain rich visual details at a fairly small scale, while the extent of the model is often large. To address the complexity, we have introduced a new set of data structures, motivated by urban morphology, to deal with and manage it. An expandable system was developed to interactively analyze, build and modify the data structures associated with the visualization.

Urban scenes are heavily structured [Lyn60]. To take advantage of the structure, we perform an analysis of the urban environment, develop algorithms to extrapolate urban features, include an implementation that deals with a fixed set of urban conditions, and finally propose a treatment that uses the structure more concretely. We have developed a system to generate synthetic urban environments and successfully used our system to visualize these synthetic environments, as well as a modeled portion of Paris.

Thesis Supervisor: Julie Dorsey
Title: Associate Professor

Acknowledgments

This document is a product of over a year and a half of research. I have to thank many people for their constant feedback and support. In particular, I would like to thank Julie Dorsey and François Sillion for their constant advice. Thanks also to Hans Pedersen for many useful discussions during the development of the project and Noshirwan Petigara for his help with the Paris model. Finally, I would like to thank all my family and friends, whose support helped me stay focused and determined. It is finally over, everyone.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Visualization of urban environments

Urban scenes present themselves as good examples of some of the very complex environments that need to be visualized and simulated interactively. They contain a wealth of information, both visually and structurally. We believe that they form an interesting subset that warrants specialized research, because they contain a strong underlying structure and challenging visual conditions.

A particularly interesting factor in this quest for real-time performance is that the user expectations in terms of image detail and quality grow as more graphics processing power becomes available. Thus an increase in raw graphics processing power does not directly translate into a greater maximal size of the database that can be visualized. Instead, some of this power is typically used to render more geometric or lighting detail in "interesting" areas of the scene. In this respect, we anticipate that high-performance visualization will not meet the user's expectations simply by relying on hardware performance increases alone.

Much research in real-time visualization has been devoted to general techniques that either simplify geometry or effectively extract the currently visible geometric dataset. However, little work has been devoted to understanding the environments being visualized and using the information about their structure. Understanding the structure and coherence of organized geometric environments opens up large avenues

of exploitation, which have been inaccessible due to the limitations of existing techniques and representations.

This thesis will consider the visual qualities of urban environments by studying their morphology. It will concentrate on these qualities and their applications to real-time visualization. Kevin Lynch, a well known scholar in urban planning, has emphasized the importance of *legibility* of the cityscape [Lyn60, Lyn96]. By this he means the ease with which its parts can be recognized and can be organized into coherent components. A legible city would be one whose districts, landmarks or pathways are easily identifiable and are easily grouped into an overall pattern.

Indeed, a distinctive and legible environment heightens the potential for applications in visualization. Potentially, the city is in itself a powerful symbol of organized complexity. If well set forth, the identification and use of structure during visualization will improve the user's visual experience by providing better interactivity. The simulation and visualization of urban environments are necessary for a number of applications and pose a number of significant challenges for the design and implementation of high-performance graphics tools. Typical applications include:

- city planning (A good example of this application is the model of South Central Los Angeles built and visualized by the UCLA School of Architecture for city planning purposes [JLF95]),
- construction and renovation in urban areas: visual impact studies,
- climate and environmental studies (plant growth in urban areas, detailed visualization of a number of simulations such as the diffusion of pollutants, et cetera),
- virtual tourism and education,
- civil and military simulators (flight, drive, combat), and
- navigation helpers for automobiles.

The structure of a city can be understood as the superposition of spatial, social, and historical relationships: any city has an obvious spatial structure consisting of streets, parks, and built areas, but the history of the city development and its economic/social organization also constitute key elements for the structure of urban environments. There is a growing consciousness that we need to actively study and monitor the way cities and urban environments develop. The recent U.N. conference on human settlements (HABITAT-II), held in Istanbul in June 1996, identified some of the challenges faced by governments in organizing urban growth. These issues can only become more pressing as by 2006 more than half the world's population is estimated to live in cities.

All these factors could be used to evaluate the importance of various components of the model for any particular visualization scenario. We exploit the notion of legibility in the way we identify and construct a number of representations, and allow users to manipulate the scene.

1.2 Previous work

The focus of most previous related work in computer graphics has been on the development of algorithms for visualization of large scenes, notably based on spatial partitioning and culling, level-of-detail approaches, image-based rendering, and hybrid approaches. Little work has attempted to use the structure inherent in the models to increase visualization performance. The most thorough work that analyzes the structure of urban environments is in the field of urban planning. We will briefly explore some of this work.

1.2.1 Walkthrough systems, spatial partitioning

In order to achieve interactive walkthroughs of large building models, a system must store in memory and render only a small portion of the model in each frame; that is, the portion seen by the observer. Research on increasing frame rates during interactive visualization of large architectural models has been underway for over twenty

years [Cla76, Jon71]. Pioneering work in spatial subdivision and visibility precomputation was done by Airey *et al.*, Teller and Séquin, and Luebke and Georges *et al.*[ARB90, TS91, LG95]. These methods for interactive walkthroughs of complex buildings compute the potentially visible set of polygons for each room in a building, use real-time memory management algorithms to predict observer motion and pre-fetch from disk objects that may become visible during upcoming frames. These techniques determine a small portion of the model to store in memory and render during each frame of a building walkthrough.

Other algorithms have been described for culling occluded polygons during interactive visualization. The hierarchical Z-buffer algorithm [GK93], uses a pyramid of Z-buffers to determine the cells of an octree (and the enclosed polygons) that are potentially visible for a particular viewpoint.

The techniques described above gain much of their effectiveness from the significant occlusion present in a building model. Outdoor urban environments are much less occluded, especially for points of view high above ground level. Therefore, these building walkthrough techniques do not scale to outdoor urban environments.

1.2.2 Level of detail modeling

A fruitful approach to high-performance visualization is to use the concept of "levels of detail" (LOD): several descriptions of the objects in the scene are provided or automatically computed, with different levels of complexity [RB93, Kaj85, FS93]. One of these representations is dynamically selected for rendering based on the viewing conditions and other factors. For example, at large scales, geometric models are necessary. At intermediate scales, texture mapping and similar techniques may suffice.

Early flight simulators were the first systems to exploit the LOD concept for interactive visualization [Bla87]. More recently, Funkhouser and Séquin [FS93], Maciel and Shirley [MS95], Chamberlain *et al.*[CDL+96], Shade *et al.* [SLS+96] and of Schaufler *et al.*[SS96] have incorporated new LOD approaches into walkthrough systems for complex environments. The work of Shade *et al.* and Schaufler *et al.* is the most relevant. They dynamically and automatically create view-dependent, image-based LOD

models. They use a spatial hierarchy to divide the scene so that the LOD models represent regions of the scene. This allows them to properly depth-sort the LOD models for rendering.

1.2.3 Image-based rendering

Recently, *image-based rendering* has emerged as a new approach to rendering and interacting with a scene. In this strategy, a 3D scene is supplanted by a set of images. Interactive scene display is achieved through the process of *view interpolation*, in which different views of a scene are rendered as a pre-processing step, and intermediate views are generated by morphing between the precomputed images in real time. This approach has been employed in flight simulators [Bla87], and has been applied to more general graphics applications by Chen and Williams [CW93] and McMillan and Bishop [MB95]. A major advantage of image-based rendering is that storing and traversing the scene are only weakly dependent on object space complexity, which makes it possible to tour complex scenes on machines that lack graphics hardware. A common drawback of these approaches is the difficulty in synthesizing views from arbitrary viewpoints. Recent results from Computer Vision can be adapted to help in the creation of new views from points distinct from the original viewpoints [DTM96, SD96].

Another new approach involves generating new views from arbitrary camera positions without depth information or feature matching, simply by combining and resampling a set of images [GGSC96, LH96]. These techniques interpret input images as 2D slices of a 4D function called the *light field* and allow significantly more freedom of movement in the range of possible views that can be generated. The light field represents the complete flow of light in a region of the environment and does not make assumptions about reflectance properties. Because of its high dimensionality, the light field requires a lot of memory and its effectiveness will be largely determined by the availability of efficient compression methods.

Regan and Pose [RP94] presented another image-based approach in which they render a scene onto the faces of a cube centered around a viewer location. The work

of Shade *et al.*, and Schaufler *et al.*[SLS+96, SS96] mentioned above is a hierarchical extension of the Regan and Pose approach.

The work is related to the hierarchical image-cache concept outlined by Shade *et al.* and Schaufler *et al.*Their methods build a BSP-tree that hierarchically partitions the geometric objects in a 3D scene, with geometry stored only at the leaves of the hierarchy. During a fly-through, images at various levels in the hierarchy are cached for reuse in subsequent frames. A simple error metric provides automatic quality control.

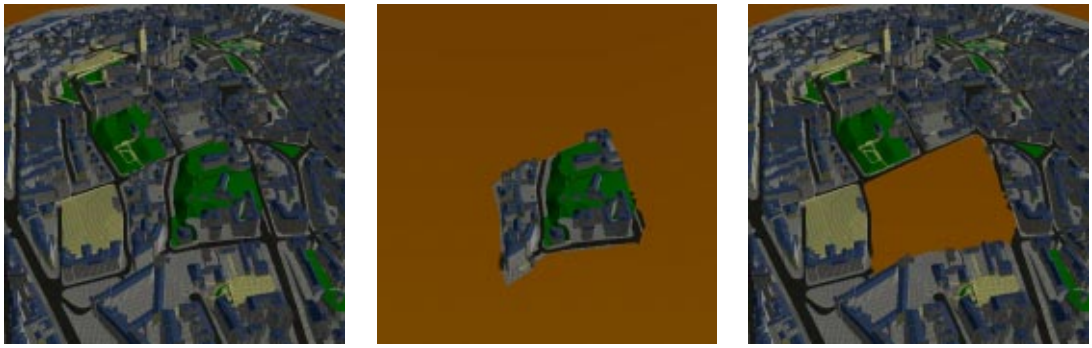
1.2.4 Hybrid techniques

Sillion *et al.*[SDB97] introduced a framework to visualize urban scenes using a hybrid system. The framework combines both traditional 3D techniques and image based rendering. The central concept proposed by Sillion *et al.* is that of a dynamic segmentation of the dataset, into a local 3D model and a set of "impostors" used to represent distant scenery (distant model). These impostors combine 3D geometry to correctly model large depth discontinuities and parallax, and textures to rapidly display visual detail. Thus, the impostors can be thought of as a 3D image, with subparts appropriately placed in 3D.

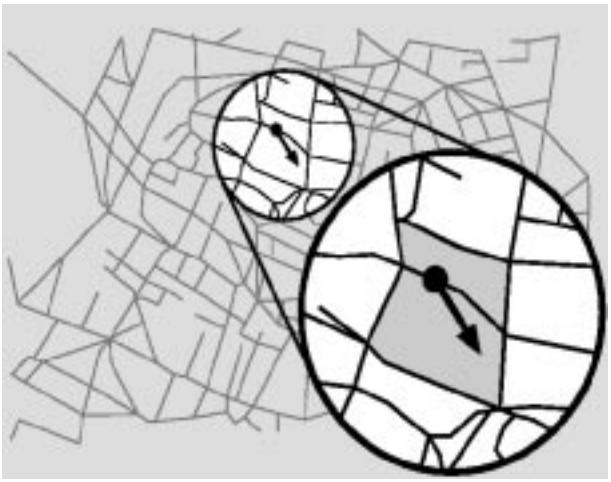
Sillion *et al.* used a simple segmentation based on urban subdivisions ("blocks" divided by streets). These blocks define the units with which the complete model (Figure 1-1a) is further divided into a "local model" (Figure 1-1b) and a "distant model" (Figure 1-1c). The local model is the fraction of the 3D scene extracted from the complete model using their segmentation technique. The distant model is the remainder of the 3D scene, and an impostor is used to render this distant model.

Users walking or driving on the ground are constrained to a network of streets. When users are at a given point on the network, typological information is used to extract the local 3D model. The simplest definition of the local model is all the blocks that touch the current street segment the user is on (Figure 1-1d).

The impostors used to represent the distant model are either created off-line, as a pre-process, or on demand when the user enters a new area in the model. Impostors



(a) (b) (c)



(d)

Figure 1-1: Blocks represented in the model

are associated with each street segment in the network of streets. Because visibility of distant objects is in practice mostly limited to the directions of the streets ends, associated to each street are two impostors. This implicitly assumes that the streets are in a densely built environment, where visibility is blocked by buildings along each side of the street.

The main stages of the impostor construction algorithm are listed below, and illustrated by Figure 1-2. The steps of the creation algorithm are:

1. Create an image of the distant scenery (to be used as the impostor texture). (Figure 1-2a).

2. Save the corresponding depth image (contents of the z-buffer). (Figure 1-2b)

3. Extract the external contour of the image. (Figure 1-2c)

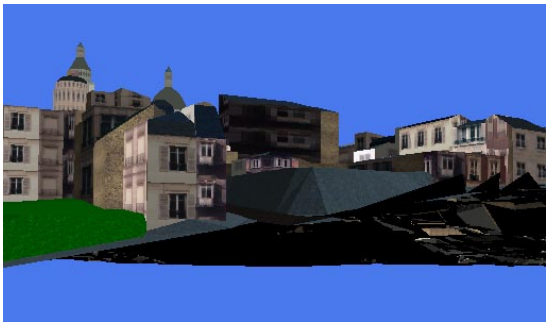
4. Identify the significant depth disparity contours. (Figure 1-2d)

5. Perform a constrained triangulation of the impostor. (Figure 1-2e)

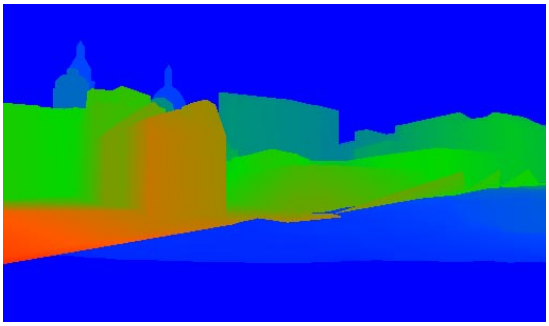
6. Store the list of 3D triangles along with the texture image.

The impostor used here produces some good results. Parallax effects, such as those when the most distant buildings become obscured by nearby ones, are effectively captured (Figure 1-2f). This approach enables the interactive visualization of these urban environments on low end graphics hardware.

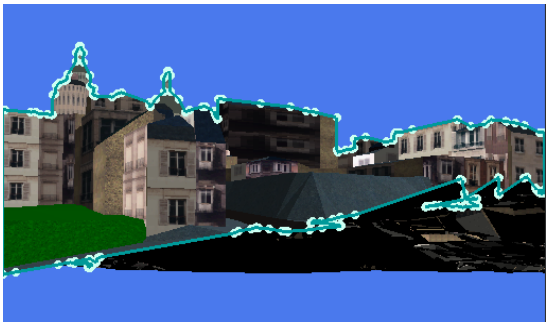
In this thesis, we describe an initial study that addresses these shortcomings. The system, however, has several drawbacks. The segmentation is rather simplistic and does not take full advantage of the structure existent in the urban environment. Their system divides the model into blocks and streets. The definitions of where impostors are used is only successful in one case, where where the viewer is on a densely occluded street where the far geometry needs to be represented by one image. It fails in situations such as squares and intersections where a large field of view needs to be represented. Also, if the viewer strays from the field of view of where the imposter was ordinarily created, the far field appears blank.



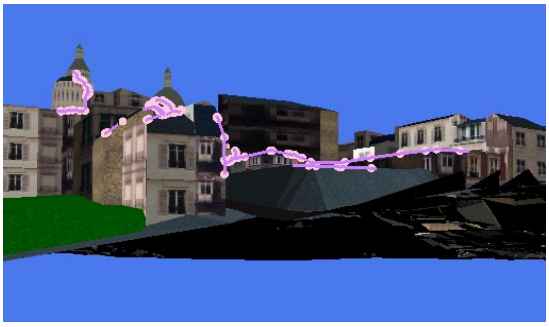
(a) - Impostor texture



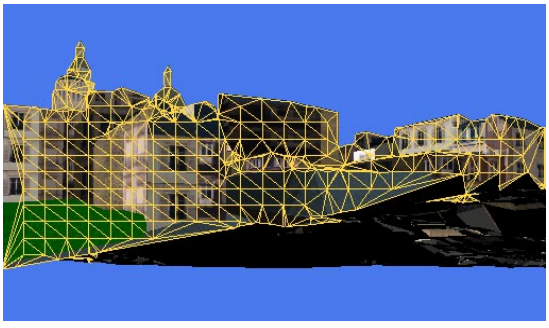
(b) - Depth image



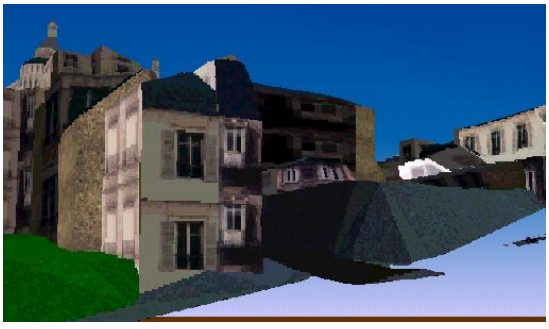
(c) - External contour



(d) - Depth disparity lines



(e) - Impostor Triangulation



(f) - Other view

Figure 1-2: Impostor creation steps

1.2.5 Urban planning analysis

The urban planning discipline has provided many of the tools needed to analyze urban environments. Understanding the methodology of urban planners enables us to identify which urban characteristics (e.g. squares, landmarks) are relevant, how they are categorized, and how they are defined.

Lynch [Lyn60, Lyn96], in his discussions of "legibility," classifies cities to contain the following primitives: paths, edges, districts, nodes and landmarks. Kostof [Kos91] in his definition of a city identified that it contains a "framework of monuments." A framework of monuments means that reference points exist in the city and are used by individuals as part of their urban experience. The reference points, discussed by Kostof, are composed of the primitives defined by Lynch. However, Kostof stresses the importance of landmarks as reference points, particularly those present in the sky line. "Sky lines are an urban signature. They are the short hand of urban identity...Cities raise distinctive landmarks to celebrate faith, power, and special achievement."

The crudest categorization of a city can be as: *planned* or *spontaneous* [Kos91]. Spontaneous cities have a very organic structure with random open spaces and curved streets. Planned cities, however, have a more methodological structure such as grid-like street patterns. City cores usually have an organic character and new additions are usually planned [Kos91]. Cities can be further categorized into more specific categories [Bur71]: concentric cities, cluster cities, linear cities, and grid cities. Concentric or peripheral cities are composed of one center, with radial routes, surrounded by rings of development. Cluster cities can exist in three forms: The first appears as more then one concentric city merged together. The second is a cluster of smaller municipal cities with separate downtowns for each municipality (constellation). The third is as as series of cities connected to each by radial transportation lines (satellite city). Linear cities are built around lines of communication. Finally, grid cities are constructed by having a grid-like road patterns. Grid or gridiron street patterns are the most common pattern used for planned cities [Kos91].

Lynch [Lyn60, Lyn96], in addition to identifying urban primitives, defines each

primitive precisely. We use his definitions to construct algorithms to identify the primitives. The definitions of physical forms are used as the primitives for analyzing urban environments. We propose a treatment, geared to visualization, of the information identified by the analysis.

1.3 Contribution

This thesis is part of a larger system developed by Sillion *et al.*[SDB97]. The central concept introduced by Sillion *et al.* is that of dynamically segmenting the dataset into a local 3D geometric model and a set of image-based "impostors" used to represent distant scenery (distant model). The impostor structure was derived from the level-of-detail approach, and combines 3D geometry (to correctly model large depth discontinuities and parallax) and textures (to rapidly display visual detail).

Our main contribution involves the development of a new implementation of the Sillion *et al.* system. This implementation introduces enhancements to the framework necessary for the introduction of new and more sophisticated data structures. A study of urban morphology is performed, which facilitates an additional set of associated data structures. We provide algorithms for the automatic extraction of the morphological components and finally propose a treatment of impostors, based on this morphology.

The new implementation is centered around the idea of interactivity because the data structures are motivated by urban morphology. The morphology is subjective and influenced by spatial, social, and historical relations and, thus, human interaction is necessary to insure the correctness of the data structures.

1.4 Outline

The remainder of the thesis is organized as follows. Chapter 2 presents an overview of the system pipeline. In Chapter 3, we outline the most relevant data structures used. Chapter 4 discusses the input data and its generation. Chapter 5 talks about the

sub-system responsible for data structure construction and visualization. Chapter 6 presents the visualization stage of the pipeline. In Chapter 7, we present an analysis of urban morphology and its use for visualization. Chapter 8 discusses the performance of the system with a series of results. Finally, Chapter 9 discusses some conclusions and future directions.

Chapter 2

An Overview of the System

This chapter introduces our system for visualizing urban environments. We describe the system pipeline as a whole and then define the different operations available to the user at each stage. This chapter provides an overview of the system; implementation details will be explained in later chapters.

2.1 System pipeline

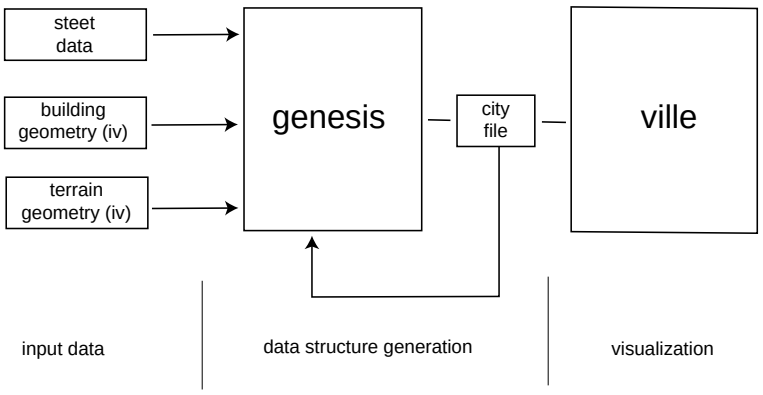


Figure 2-1: System pipeline.

Figure 2-1 represents the system data pipeline for urban visualization. The pipeline is composed of three stages: Input data and its construction, data structures generation, and visualization. The first part deals with the construction of the

generic data files needed as input to the pipeline. The second stage, which we call Genesis, is where the generic data files are used to create the data structures needed for the visualization. Genesis creates a CIT file (defined in Appendix B), which can also be read by Genesis for incremental data structure modifications. Finally visualization, called Ville, reads in all the data in the CIT file and uses it to visualize the urban environment using a hybrid model composed of both traditional 3D and image-based techniques.

2.2 Input data and generation

The first stage of the system deals with input data definitions and their generation. The input data is composed of a set of generic files. The files represent the geometry of the urban environment as well as a street plan of the road network. These files are divided into three parts. The street data is a network of line segments and uses the street file format (defined in Appendix A). The next two files use a well-known file format called the inventor file format[Wer94]. One contains the terrain geometry, while the other contains street geometry. The 3 files are segmented because each file can come from a different source. The street file can be manually generated from the terrain geometry, or alternatively from a map. The terrain file can be acquired from Geographic Information System (GIS)[GR91b]. The building geometry file can either be modeled, as was the case with the Paris model we are using, or synthesized using our city generator, which will be discussed in chapter 4 (Figure 2-2 shows a screen capture of the City Generator).

2.3 Genesis

The second phase of the pipeline is called Genesis (Figure 2-3 shows a screen capture of Genesis). This subsystem reads in all the data, dividing and organizing it in the appropriate urban data structures (Section 3.1). It then creates all the associated data structures necessary to visualize the urban environment (to be described in Chapter 3)

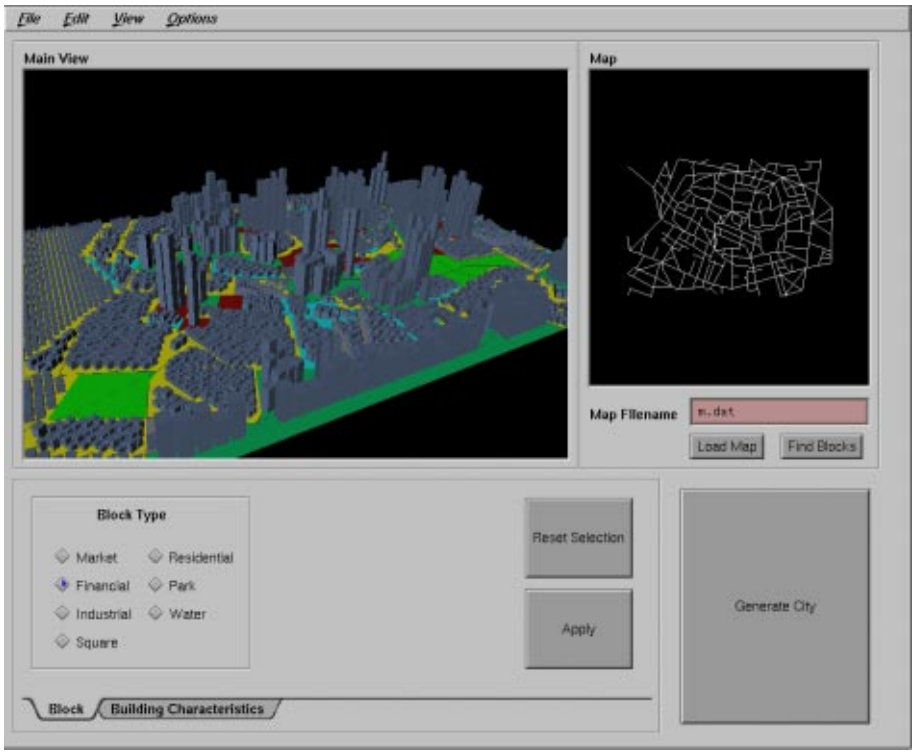


Figure 2-2: Screen capture of city generator

and attempts to identify specific urban characteristics. Genesis is an interactive tool. The data structures can be modified, recomputed and visualized as needed. All the data is then written to a CIT file (described in Appendix B). Genesis is also capable of reading CIT files; thus, incremental updates of the data structures are possible.

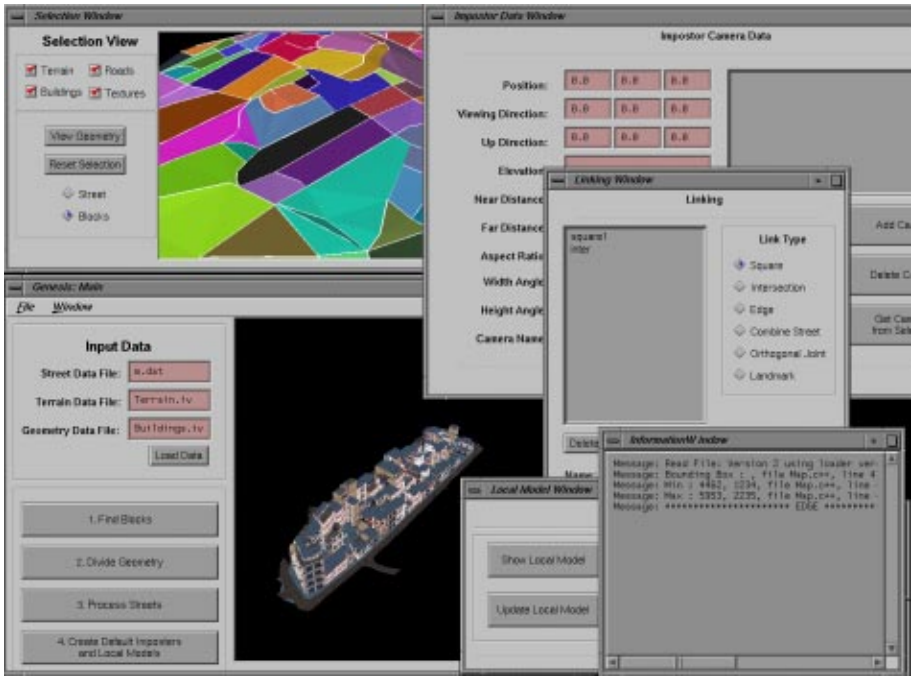


Figure 2-3: Screen capture of Genesis

2.4 Ville

The last phase is "Ville," or visualization (Figure 2-4 shows a screen capture of Ville). This module reads in the CIT file and uses the information to reconstruct the data structures. Ville uses these data structures to determine how to approach the visualization of the different parts of the model.

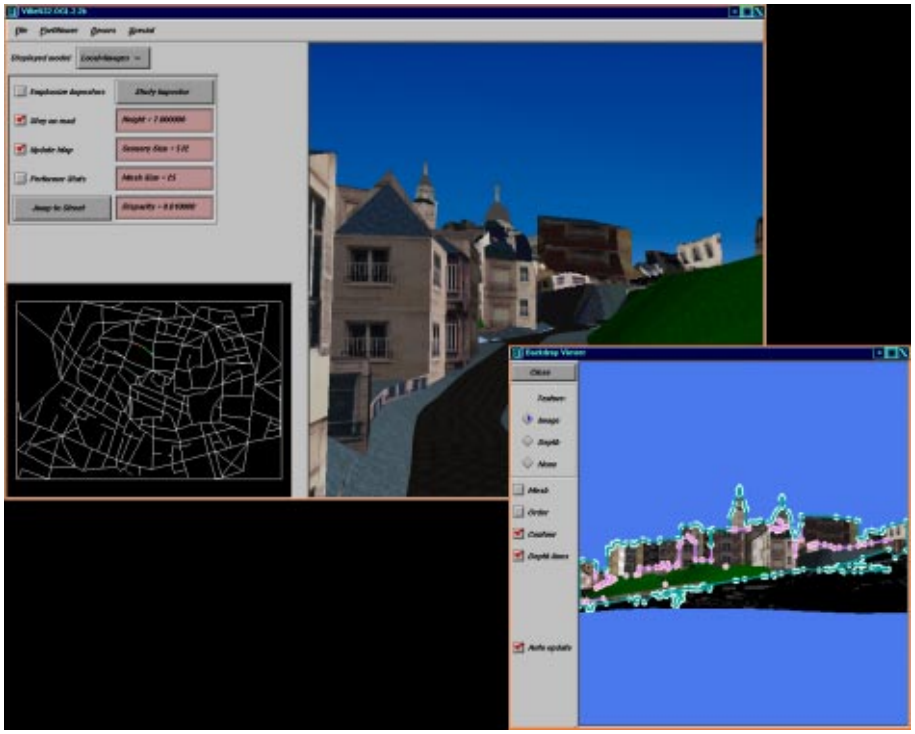


Figure 2-4: Screen capture of Ville

Chapter 3

Data Structures

This chapter presents the data structures used throughout the system. There are four sets of data structures that are worthy of in-depth discussion. The first set is the urban data structure, which defines the hierarchy of a city. The second is the winged edge data structure, which is used to navigate the urban environment. Third, the linking data structure is the framework used to define urban conditions (e.g., squares, city edges, et cetera). Lastly, the impostor data structure, which is used to define all the data associated with creating an impostor.

3.1 Urban data structures

Figure 3-1 shows the internal urban hierarchy used to define an urban environment. From a high level a city is composed of a series of blocks, a map, and a linked object manager. The map contains all the edges and vertices needed for the winged edge data structure. Blocks are the units of urban organization used in the system. The city also contains a set of linked objects (discussed in section 3.3). The linked objects are managed by a linked object manager.

Each block defines a subset of buildings that contain no paths. Each block has pointers to the edges that surround it and to its geometry. The geometry is divided into terrain or building geometry. Building geometry is further divided into building objects, where each building object is used to represent the usual notion of a building.

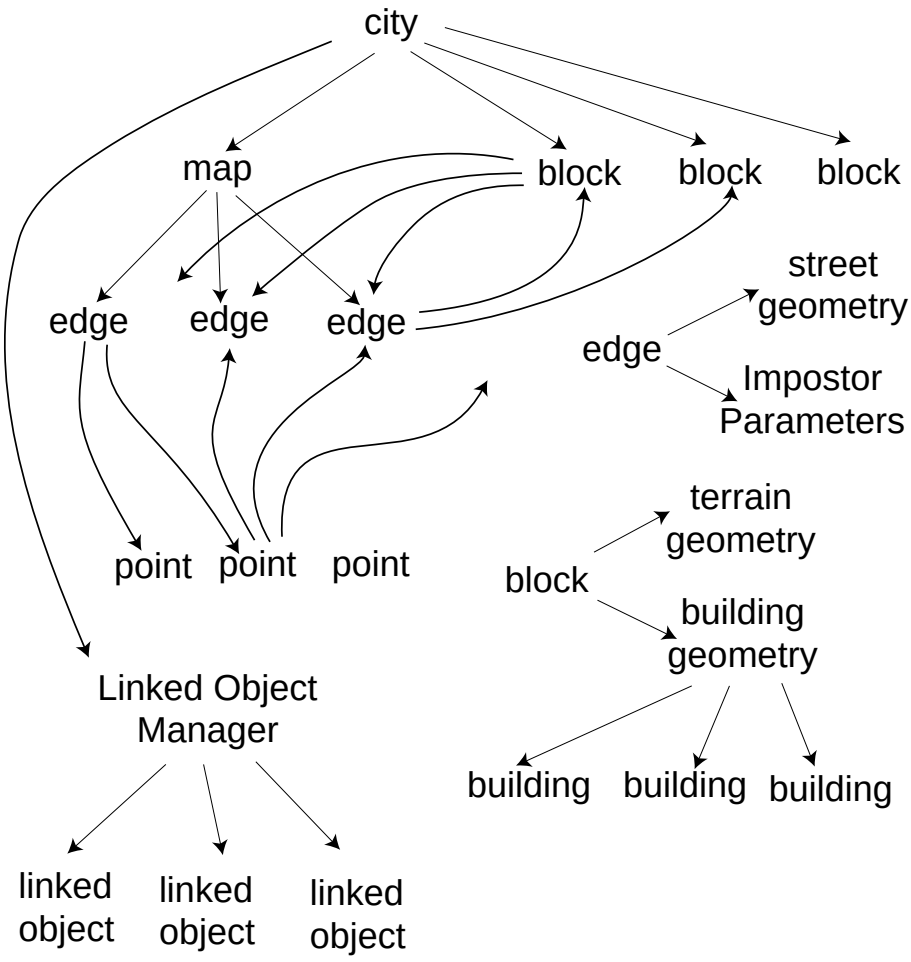


Figure 3-1: Urban data structures

Terrain geometry is a series of triangles that are not streets. Figure 3-2 and figure 3-3 show a number of blocks represented in the Paris model and map.

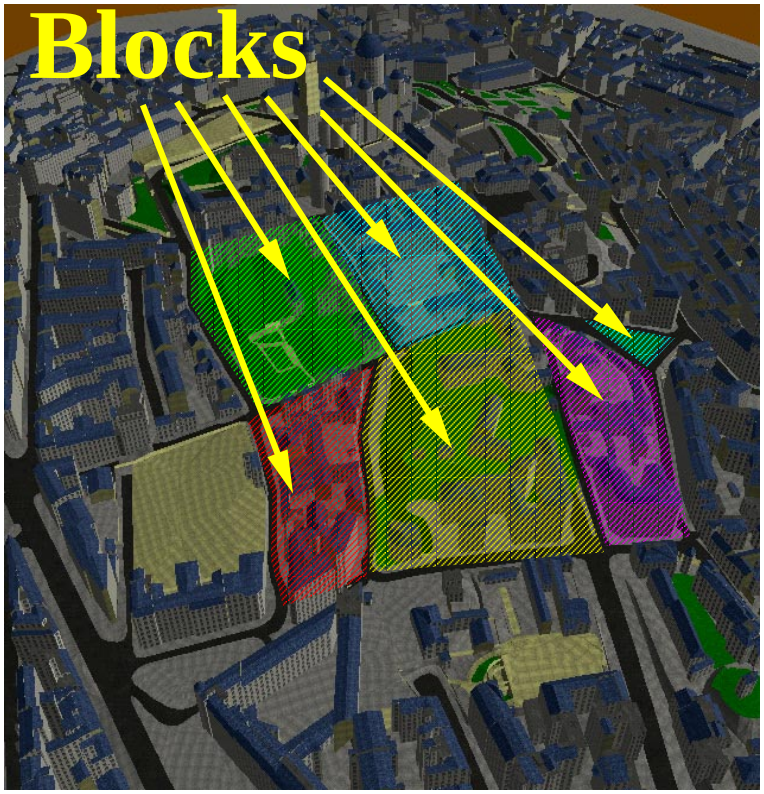


Figure 3-2: Blocks represented in the model

The map is composed of edges. An edge is a straight street segment and several edges may compose a street. Each point is a vertex, each block is a face, and each edge is an edge in the winged edge data structure (discussed in section 3.2). In addition to holding all the associated information of the winged edge data structure, edges hold the geometry they represent as well as all the impostor related data (discussed in section 3.4). Figure 3-3 shows a number of blocks and edges represented in the Paris map.

The linked object manager is the access point to the linked objects (discussed in Section 3.3). Briefly, linked objects are used to define urban characteristics, such as squares, intersections, et cetera.

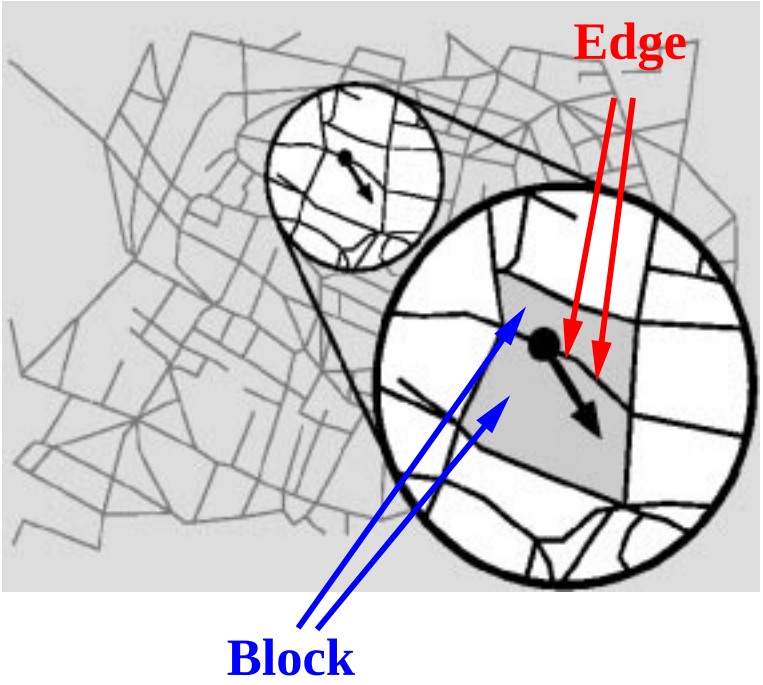


Figure 3-3: Blocks and edges represented on the Paris map

3.2 Winged edge data structure

Figure 3-4 shows the winged edge data structure used to navigate the city map. The map is composed of edges and vertices. Each edge is composed of two directed edges. For example, edge e is composed of the directed edges $e_1^{'}$ and $e_1^{''}$. The edge $e_1^{'}$ is directed from vertices v_2 to v_1 while $e_1^{''}$ is directed from v_1 to v_2. Each directed edge also knows its "twin" which is the same edge directed in the opposite direction. For example $e_1^{'}$ has the twin $e_1^{''}$, and $e_1^{''}$ has the twin $e_1^{'}$. The faces on the left and right of an edge are stored with the edges; for example, $e_1^{'}$ has F_1 to its left and F_2 to its right, while $e_1^{''}$ has F_2 to its left and F_1 to its right.

Vertices, on the other hand, store some geometric information, i.e. their position. In addition they also store a list of edges that point out from the vertex. For example vertex v_2 has the list $e_1^{'}$, $e_2^{'}$, and $e_3^{'}$ in its list edges. The data structure makes it easy to navigate the urban map and identify adjacent objects. The faces in this structure are the urban notion of blocks. Thus when on edge e_1, one can easily identify the

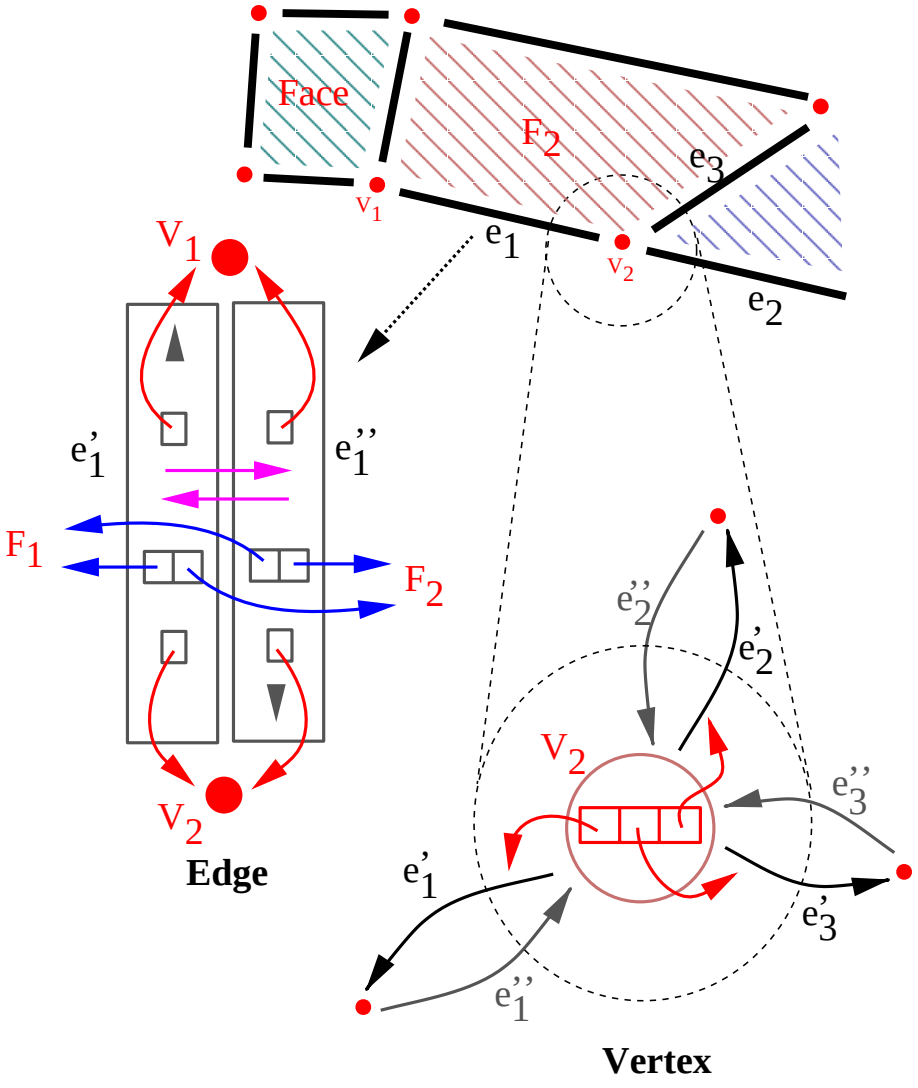


Figure 3-4: Winged edge data structure

blocks that surround the edge. Faces also have a list of the edges that surround it. The list of edges a face contains is only composed of one of the directed edges of each edge surrounding it.

This representation is rather rich. There are, however, more compact representations [Bau74, GS85, Wei86, Eas82], but we used this amalgamate form of the winged edge data structure because of its simplicity for debugging and development.

3.3 Linked objects

Linking is a concept used to group objects together. A linked object is composed of a series of edges, blocks, and buildings, a type and a name. It is the methodology we use to define one urban characteristic. The lists contain only one pointer to each object in question. The types are squares, intersections, edges, and landmarks. The details of how they are used are explained in the chapter 7. Linked object are independent units and contain objects that may be shared with other linked objects. A linked objects contains a list of other linked objects. This list contains pointers to other linked objects they share components with.

3.4 Impostor parameters

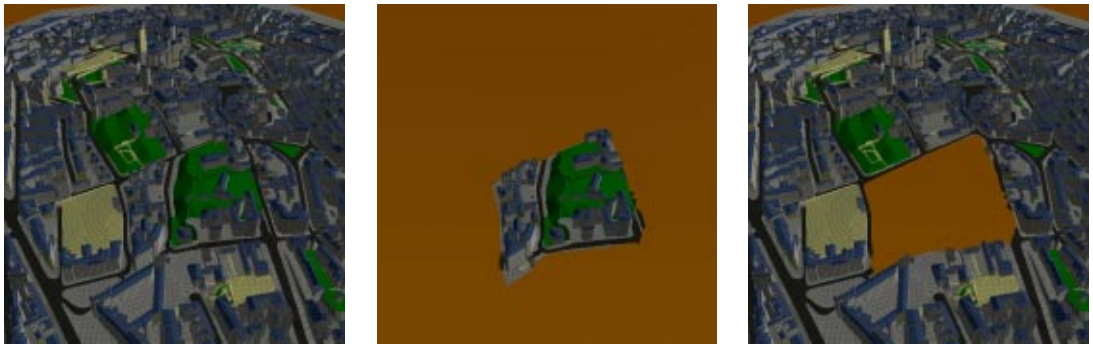
All the impostor parameter information is encapsulated in the edges. Imposter parameters are composed of two different objects. The first are the impostor camera objects. An impostor parameter object may contain one or more impostor cameras. Each impostor camera is used to create an image. Each image is then used to create an impostor. The second object is the impostor local model, this defines what blocks are used for the local model and consequently used to identify the distant model.

3.4.1 Impostor camera data structure

The impostor camera data structure contains all the information needed to define a camera. These parameters include position, viewing direction, up direction, near

distance, far distance, height angle, width angle, and elevation. The elevation parameter is the distance the camera is raised in the up direction. During visualization the system uses this object to know which impostor cameras to use and then creates the images needed for impostor creation.

3.4.2 Impostor local model data structure



(a) (b) (c)

Figure 3-5: Complete, local and distant models

The impostor local model data structure is a list of edges and blocks. It contains all the blocks that compose a local model and all the edges that share this local model. Figure 3-5 shows a simple visualization of how the complete (Figure 3-5a) model is segmented into a local (Figure 3-5b) and distant models (Figure 3-5c).

Chapter 4

Input Data and Generation

This chapter discusses the input data needed at the start of the pipeline. It describes the different input files, what they represent and how they are constructed. We, in addition, describe a system we had developed to generate synthetic urban environments.

4.1 Input files

Before a model is fed into Genesis, three files are needed to create the data structures. A street map file (file format defined in Appendix A), and two inventor [Wer94] files, one for terrain and one for buildings. The street file defines a series of 3D lines that compose the city map. The coordinates of the street segments have to match the original model since they define where the viewer can travel. The line segments that compose the streets have to connect, with the exception of intersections, which are detected automatically. The inventor file format [Wer94] is used to define the terrain geometry and building geometry. All the geometry in these files will be converted into triangles. An assumption is made that the geometry in these files is complete. This means that normals, materials, textures, et cetera are correctly assigned. The building geometry file requires further non-standard information. In the inventor file, objects need to be grouped together to form buildings and assigned an I.D. This is achieved by adding a name field to the inventor *Separator* tag.

4.1.1 Texture generation

Initially, the Paris model did not have any textures; to give the model more realism textures were generated. Before texture images were assigned to the model, texture coordinates were computed at the vertices of each face. Using the following process, texture coordinates were computed. Each face was projected onto a plane perpendicular to the normal of the face, where the up direction was predefined in the model. In our case positive z represented the up direction. This plane was, then, treated as a 2D axis. The bounding box of the projected points was computed. Each point in the projection was assigned a texture coordinate by interpolating the values defined by the bounding box and then was normalized by its limits. A texture image, from a database of scanned images, was randomly assigned to each building. The model was then saved to two files, one for terrain and another for building geometry. Figure 4-1 shows an image of the textured model.

4.2 City generator

The city generator is a tool used to construct specific urban conditions to test the pipeline and its associated algorithms. The steps taken by the city generator are detailed below:

1. The city generator reads in a street map. Figure 4-2a shows some sample street maps used.

2. The street segments are, then, fed into a constrained 2D Delaunay triangulation algorithm[DP92, Jun88, Sei88, GR91a, Slo91, KM92, WT92] where they are treated as constrained edges. The reason Delaunay triangulation is used is that intersections between street segments can then be identified and split up. The triangles created by the algorithm constitute the terrain geometry and, eventually, the blocks. Adjacency information is constructed from the resulting triangles. Figure 4-2b shows a triangulation of the map data.

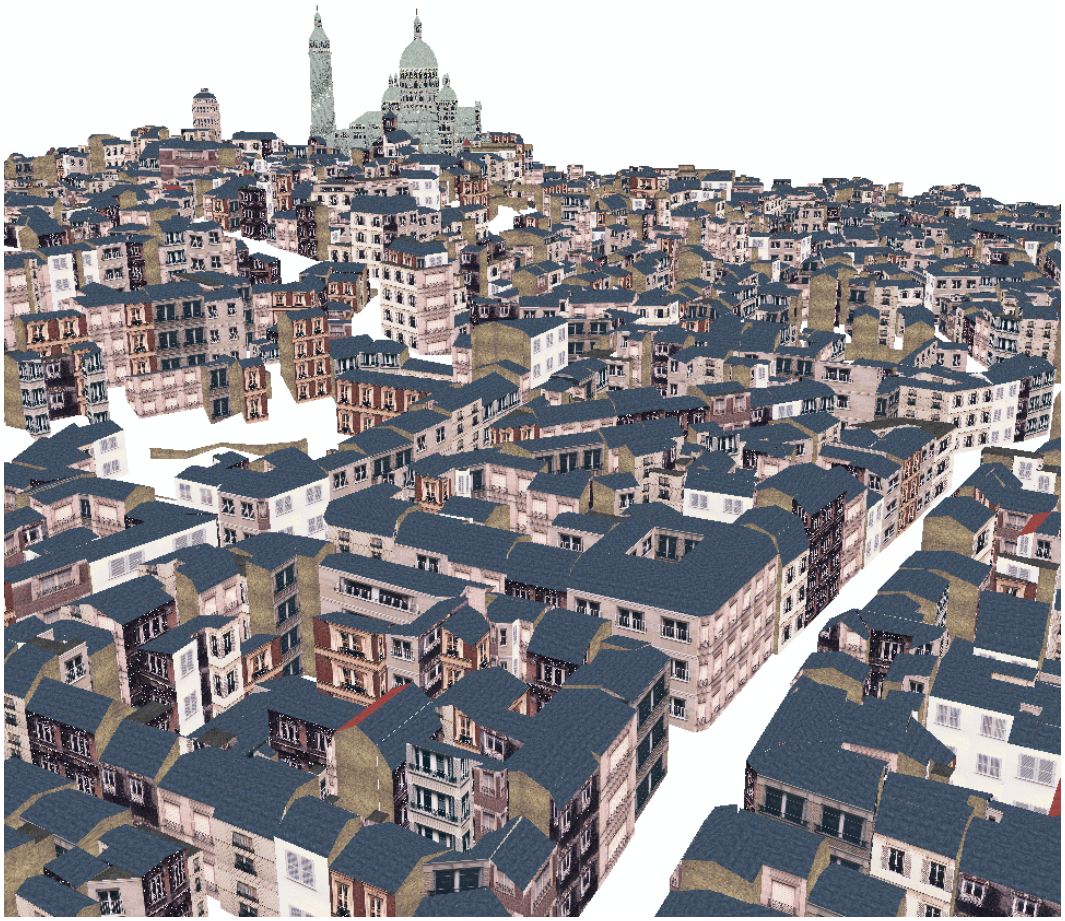
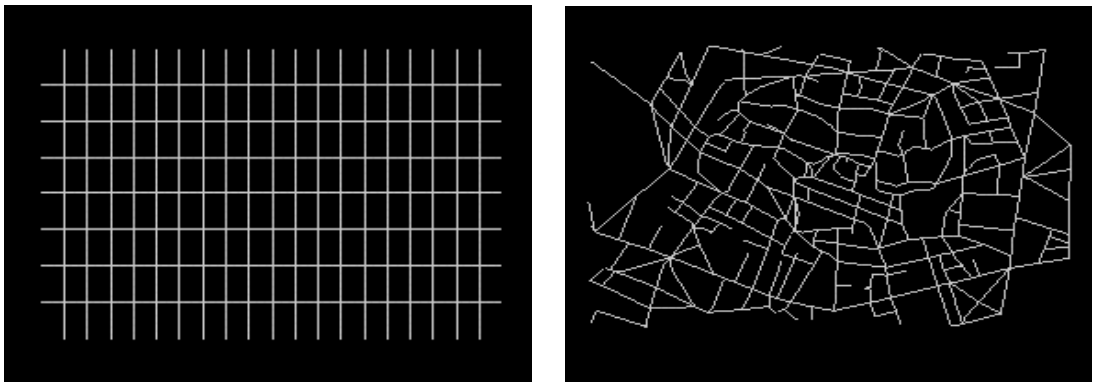


Figure 4-1: Sample view of the Paris model with textures.

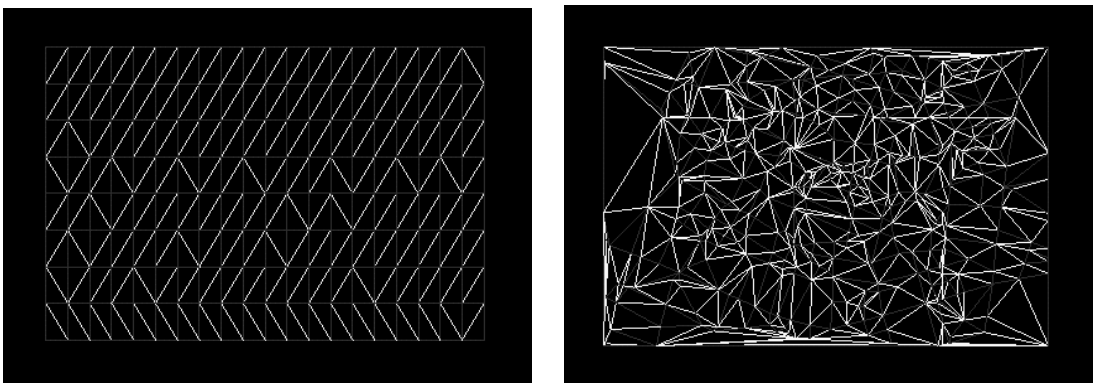
3. After the triangles and adjacency information are constructed, blocks are identified and extracted. The algorithm chosen for this step marches around the triangles and flags visited triangles until a block is constructed. The algorithm then moves on to the next unflagged triangle until all the triangles are flagged.

4. Each block is randomly assigned a property. Available properties include water, park, residential, square, industrial, financial, and market. Each property has associated with it a mean height, a mean width and a variance. Block properties can then be interactively modified.

5. The city is then generated. The city takes the form of scaled cubes that adhere to their respective means and variances. To create an element of randomness.



(a)



(b)

Figure 4-2: City generator map data and corresponding triangulation.

the orientation of the buildings is defined by the orientation of the longest edge in the block.

6. Users can then go back to step 4 or 5 if they do not like the results.

4.2.1 Results

Figure 4-3 and Figure 4-4 show four sample synthetic cities constructed from two different street maps.

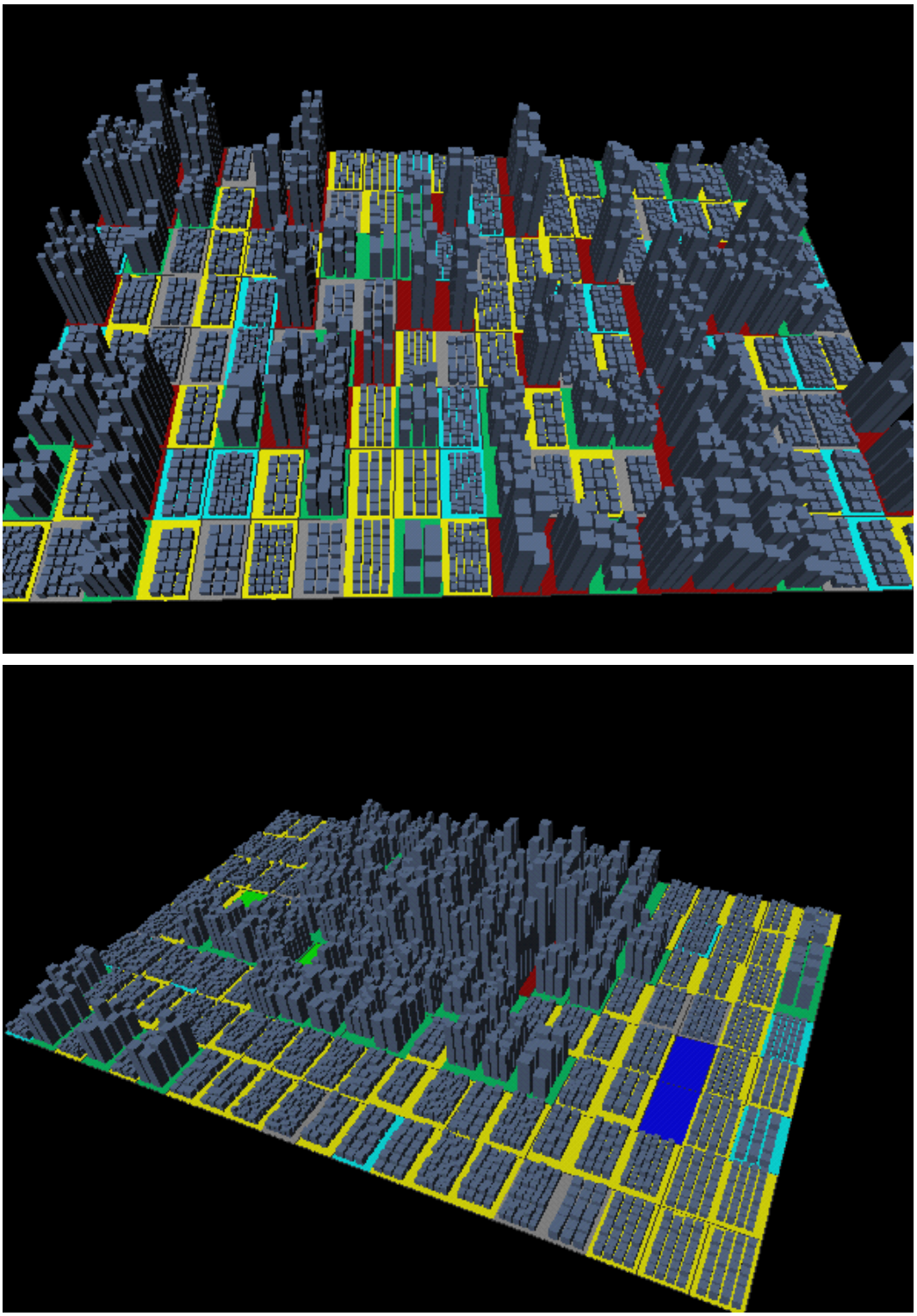


Figure 4-3: Sample generated cities using a grid street map

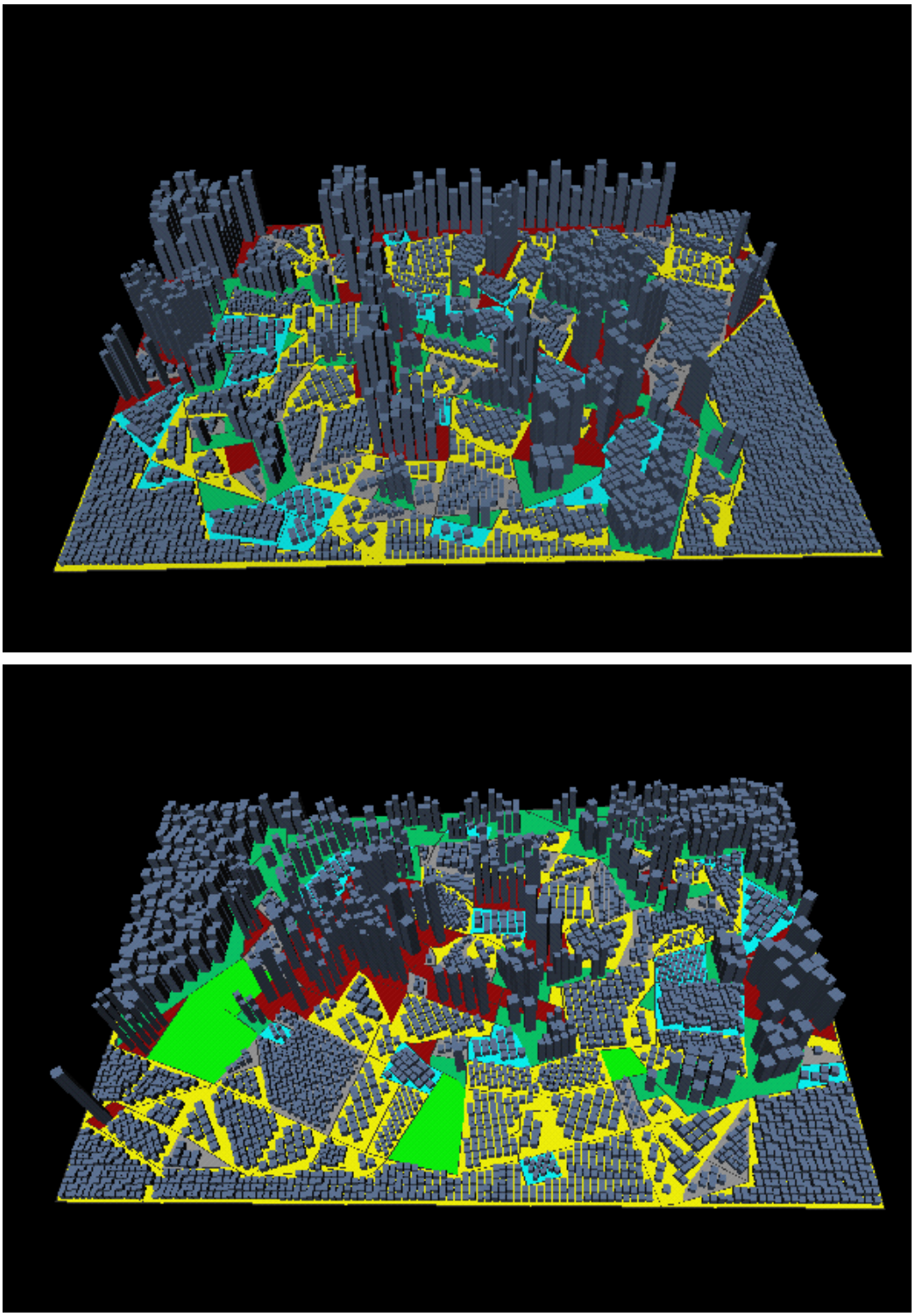


Figure 4-4: Sample generated cities using the Paris street plan

Chapter 5

Genesis: Data Structure Generation and Visualization

One of the most important features of our system is the ability to visualize and modify the data structures interactively. In this chapter, we describe "Genesis", which is the sub-system that performs all the data structure construction and modification.

5.1 Overview

Genesis has the capacity to read in 3 data files: street data, building geometry (inventor file), and terrain geometry (inventor file). It then divides all the geometry into coherent blocks and assigns each edge its geometry. All the impostor data is created and assigned to the edges. The user automatically and interactively identifies city elements and has the system create default parameters for these elements. In addition, the user can modify impostor data structures for any of the edges in the system. This process can be incremental since Genesis can reload and modify any previous work that has been saving in the form of a CIT file (defined in Appendix B).

5.1.1 Process

Genesis performs a series of intricate steps that lead to a set of data structures that is necessary for visualization. Below is a description of these steps:

1. All data must be read in.

2. The line segments in the street file are then fed into a constrained 2d Delaunay triangulation algorithm[DP92, Jun88, Sei88, GR91a, Slo91, KM92, WT92], as constrained edges. The reason this is used is that the triangulation splits up intersecting edges. The triangulation, also, is used to construct a simplified version of the terrain. To create the simplified terrain all the points in the triangulation need to be reprojected back into 3D. This is done by keeping indices to the original data in the triangulation and then computing, by interpolation, the corresponding 3D points. The edges and vertices produced by the triangulation are used to construct the winged edge data structure. In addition, the triangles are modified to include adjacency information.

3. After the triangles and adjacency information are constructed blocks are identified and extracted. The algorithm chosen for this step marches around the triangles and flags visited triangles until a block is constructed. The algorithm then moves on to the next unflagged triangle until all the triangles are flagged. (Figure 5-1).

4. All the geometry is converted into triangles and then placed into the appropriate block. This is done by a point intersection of the midpoints of the geometry triangles with the simplified terrain triangles that represent the blocks. The assumption is that the geometry has the correct position, texture coordinates, normals, and material properties.

5. Now that all the geometry is in the blocks. The next step is to extract all the street geometry and associate it with the street segments. This is done by finding the closest edge to a street triangle.

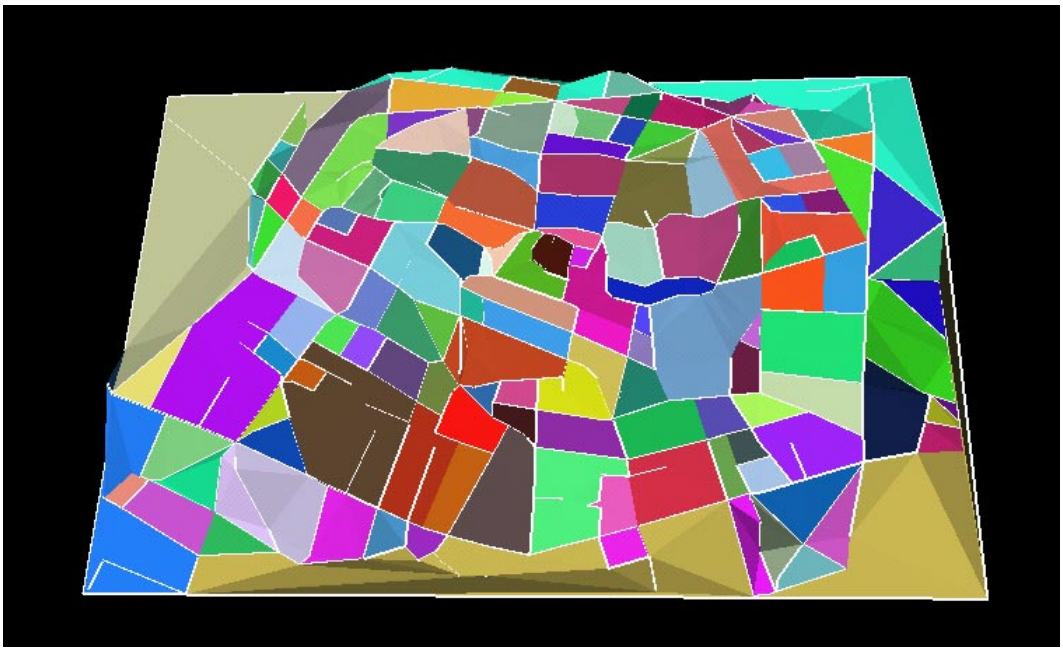


Figure 5-1: Triangulation of map and placing the edges in 3D

6. Here, the impostor data structures are constructed. Each street segment is assigned an impostor local model object and two impostor camera objects. The local model is defined as all the blocks that touch each end point of the street segment. Each street has two impostor cameras, one positioned at the end of the street and the other at the beginning. These parameters represent a set of cameras used to construct impostors. The cameras are positioned at the endpoints with a viewing direction defined in the direction of the street segment elevated by some fixed value. This is explained in more detail in Section 5.2.2 and Section 5.2.1.

7. The final step is the automatic identification of urban morphology. The algorithms used are defined in Chapter 7.

8. The user is now able to modify the data structures using the user interface.

9. The user still has the ability to further modify these data structures and experiment with some obscure cases the current model imposes.

10. All the information is then saved to a CIT file.

11. The process can start again at step 7.

5.2 Default parameters

5.2.1 Local model default definitions

Typological information is used to extract the default local 3D model around a point (a street) as the set of blocks "near" that street. The simplest definition of near is the blocks that touch the current street segment (Figure 5-2). Near encompasses all the blocks that surround an endpoint. The blocks can be easily extracted from the winged edge data structure.

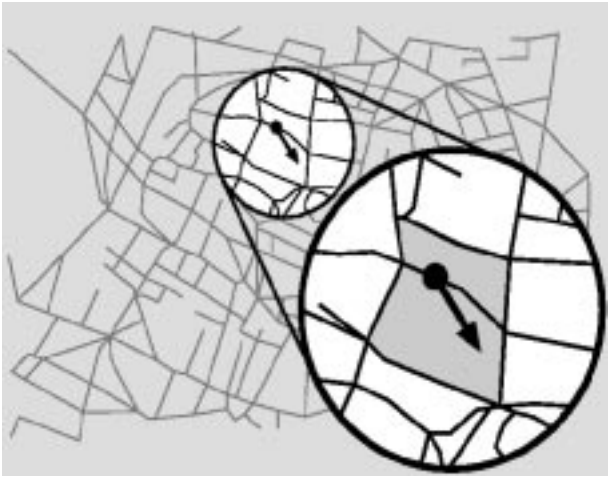


Figure 5-2: Local model definition: blocks adjacent to the current street are selected to compose the local model.

5.2.2 Impostor camera default definitions

The default impostor position assumes that streets are in densely built environments where visibility is blocked by buildings along each side of the street. This is the required behavior for most streets; otherwise, they would be identified as an urban

structure such as a city edge, square, et cetera. In this default case, the visibility of distant objects is in practice mostly limited to the directions of the street ends. Under this pretense, associated, with each street, are two impostors (one with each directed edge). Each impostor is constructed from a camera position defined at the beginning of the street (each directed edge) in the direction of the street.

5.3 User interface

Genesis's user interface can be divided into six components: the main window, the information window, the selection window, the linking window, the model window, and the impostor window. Each window has a specific job and is capable of affecting other windows.

5.3.1 Main window

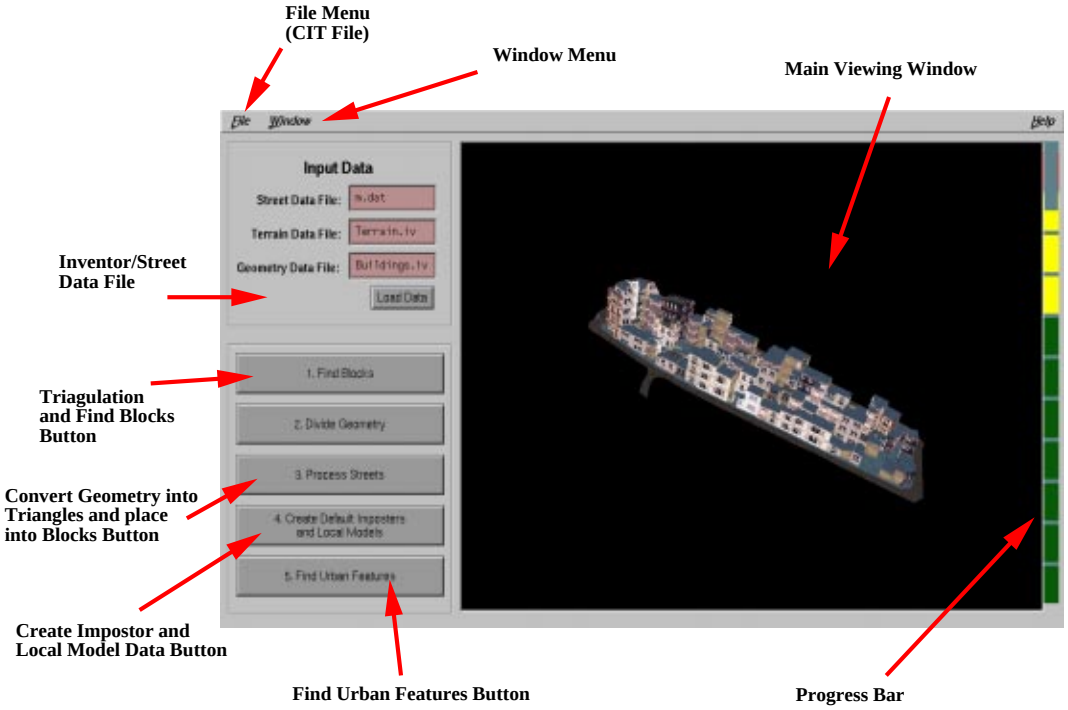


Figure 5-3: Genesis: main window

Figure 5-3 shows the main Genesis window. It is divided into five parts. The first is the progress bar which is a meter that oscillates to indicate the program is processing. The second component is the main viewing window. This is a 3D window that shows model geometry. The user controls what is being displayed in this window using the selection window (section 5.3.3). The third component is the menus. The file menu is where the system loads and save CIT files. The window menu is used to open all other child windows used by Genesis. The fourth component is the Input Data section which defines the filenames of the street file, terrain file, and the building file. The last component encapsulates the controls provided by Genesis. These include finding the blocks (step 3), dividing the geometry (step 4), processing the streets (step 5), creating the default impostors and local models (step 6) and finally finding the urban features (step 7). These need to be used once for each CIT file. All other modifications are performed in the other windows. However the main viewing window remains as the only place selected geometry can be viewed.

5.3.2 Information window

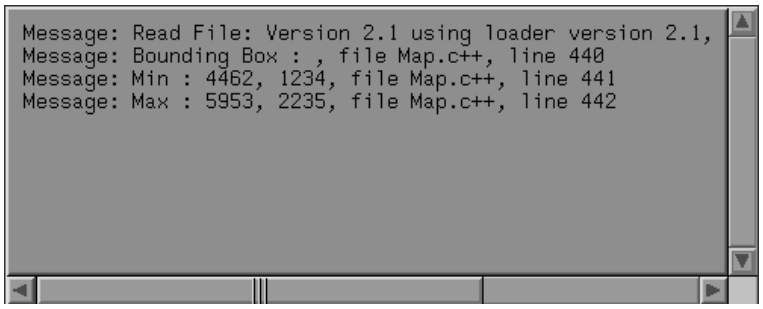


Figure 5-4: Genesis: information window

Figure 5-4 show the information window. It is a simple window used to send progress messages to the user. For example, the bounding box for the map is currently displayed in this screen capture.

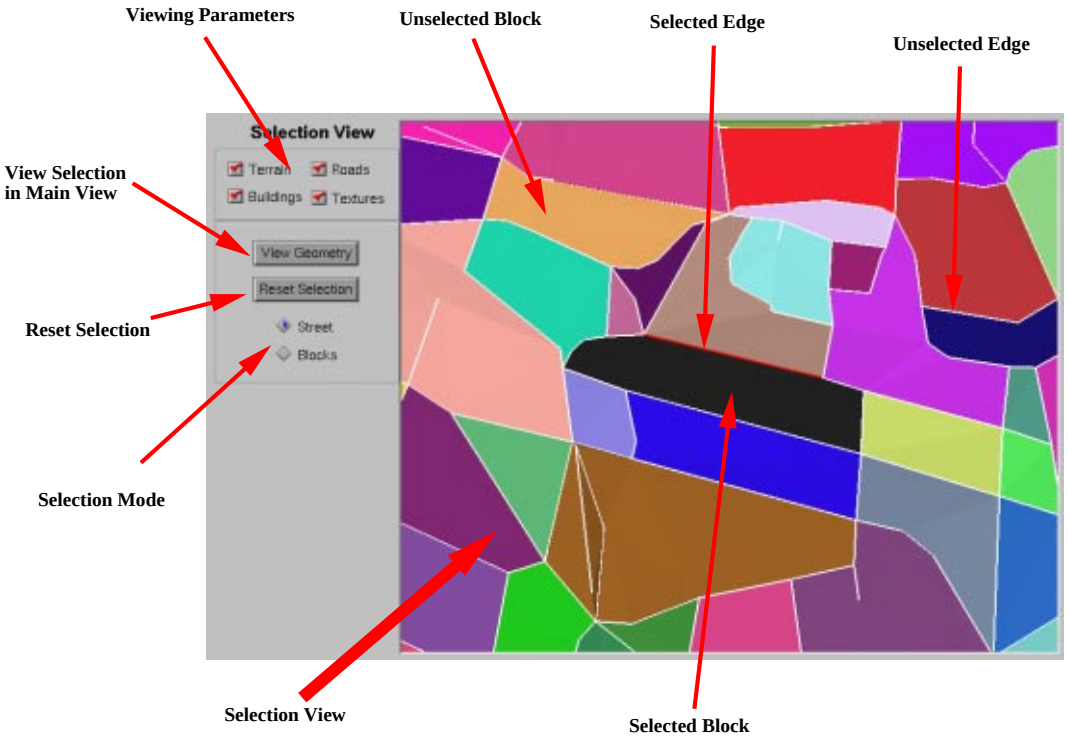


Figure 5-5: Genesis: selection window

5.3.3 Selection window

The selection window (Figure 5-5) is where most of the data structure modifications and visualizations take place. It is a cheap and simple method for selecting blocks and edges for further modification. The selection view is a 3D window that shows a simplified 3d map/terrain of the model. Unselected blocks appear with pastel colors, while selected blocks are colored in dark grey. Unselected edges are colored in white, while selected edges are colored in red. The user can select and deselect any of blocks and edges while manipulating the map in 3D. The viewing parameters define what object pieces are viewed in the main window when a request is made. Possible selections include terrain geometry, road geometry, building geometry and whether or not texturing should be used. There is a toggle for selection mode and a user can either select blocks or edges. The "view selection" button is used to show the selected geometry in the main view. The "reset selection" button is used to reset all the blocks and edges to unselected. The ability to select edges and blocks here is a

critical component of the the system as it provides the framework to visualize any of the data structures, as well as visually modify them.

5.3.4 Linking window

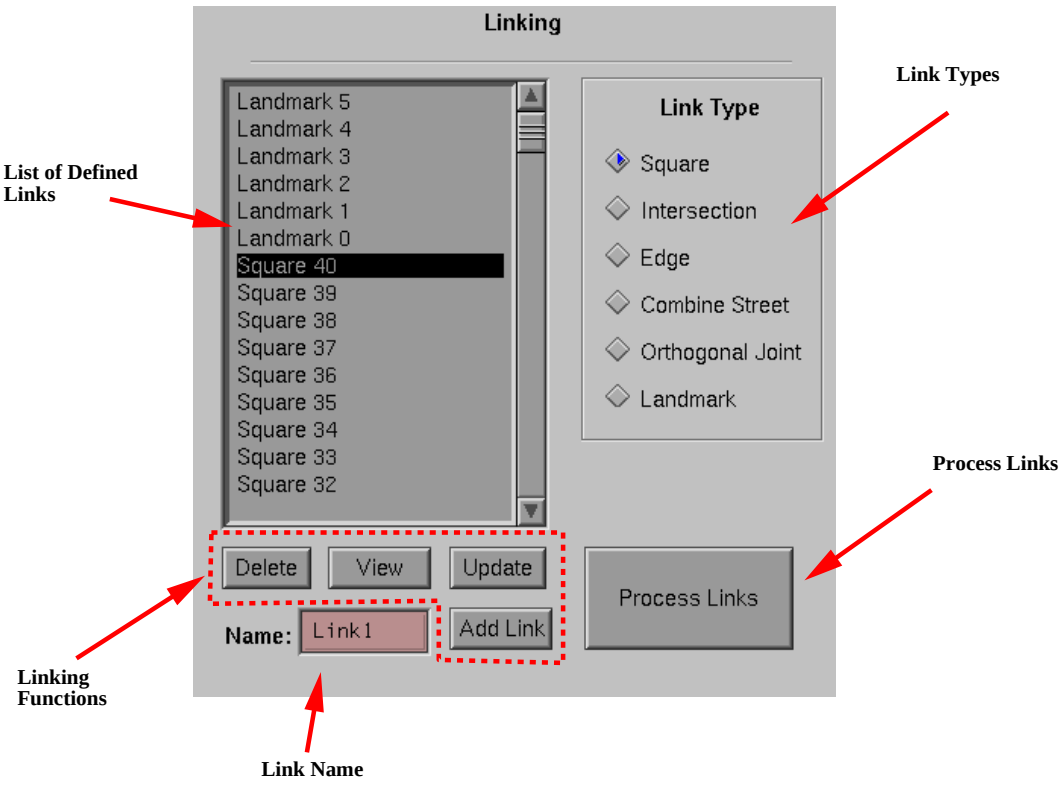


Figure 5-6: Genesis: linking window

The linking window (Figure 5-6) is where all the linking data structures are interactively modified deleted and created. There is a list of defined linked objects with their corresponding names. Here links are selected for manipulation. The link types shows the types associated with a selected link. The linking functions are the tools used to manipulate these linked objects. A linked object may be deleted, which involves removing its information from the system; viewed, which involves updating the linking window with its properties and selecting the appropriate blocks and edges in the selection window; updated, which uses the new information in the linking window and selection window to update the contents of the current link; and added,

which involves creating a new linked object that contains all the information in the linking window and selection window. Finally, process links uses the information in the links to modify all the linked objects to adhere to the local model definitions and imposter definitions defined in section 7.6. In addition this function also creates all the cross links (defined in Chapter 7.6.5) between the linked objects for further use during visualization.

5.3.5 Model window

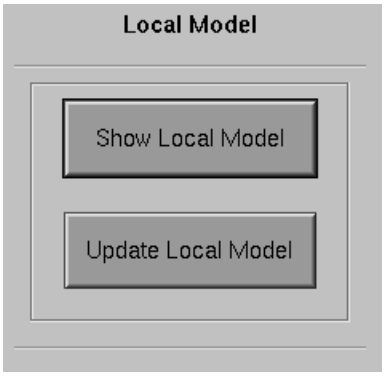


Figure 5-7: Genesis: model window

The model window (Figure 5-7) is a very simple method to view and update the local model associated with an edge or a set of edges. The "show local" button is used to view the local model for a selected edge in the selection view. If the local model contains more edges that share the local model, they too are highlighted in the selection window. The "Update Local Model" button uses the currently selected blocks and edges in the selection window to update the respective edges' local model definition.

5.3.6 Impostor window

The impostor window is very similar to the linking window in that it provides visualizations of impostor cameras in the model and provides users with the ability to

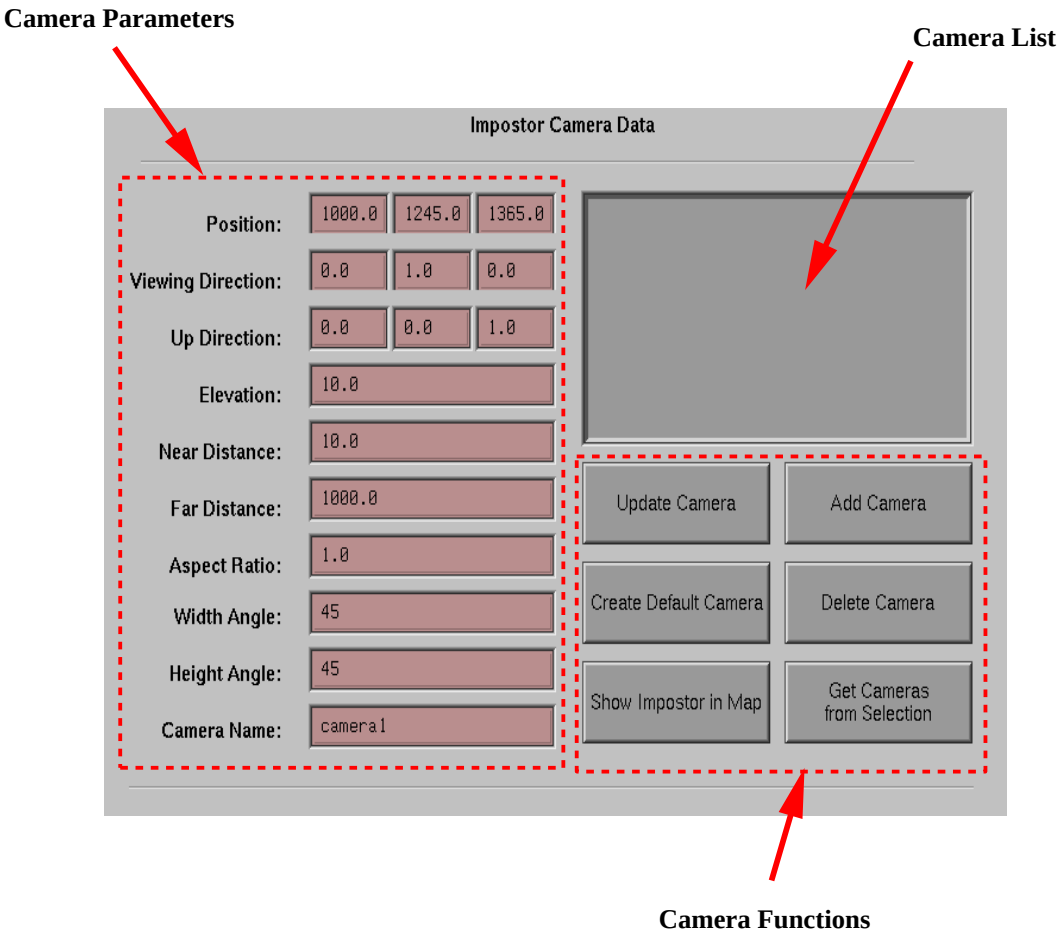


Figure 5-8: Genesis: impostor window

modify this information. The camera list shows all the cameras used by the currently selected edge. The user can select, modify, delete, and create new cameras for the selected edge(s). Associated with each impostor camera are a set of parameters that can also be modified. The camera functions are used to make changes as well as visualize impostor cameras as viewing frustums in the selection view.

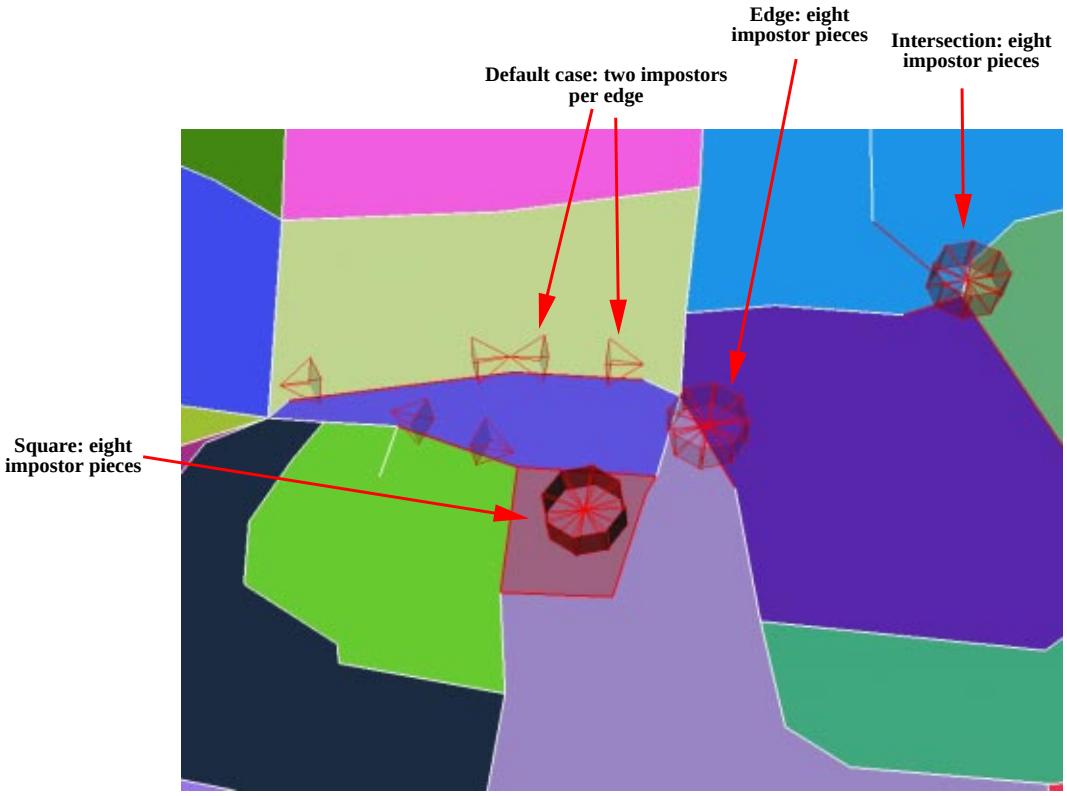


Figure 5-9: Impostor camera visualization

5.4 Visualizations

5.4.1 Impostor cameras

Figure 5-9 and figure 5-10 represent 2 very important visualizations in "Genesis". Figure 5-9 shows a variety of impostor cameras for several urban conditions. Each rotated pyramid looking object represents a camera. The peak of the pyramid represents the camera position, while the base of the pyramid represents the image plane for that camera.

5.4.2 Local model visualization

Figure 5-10 shows a square's local model. All the dark blocks represent the blocks that compose the local model. The red edges represent the edges that share this local model. It should be noted that the same edges share the same impostor camera parameters.

5.4.3 Landmark visualization

Figure 5-11 shows how a series of landmarks are visualized in the system. Initially, the block containing the landmark is highlighted (in the selection window) to show the position of the landmark. Then when the geometry is viewed in the main window, landmarks are rendered in red. In the figure there are two landmarks: the church and tower. The church as you might notice is not completely red. This is a flaw in the building definitions inputted into the system.

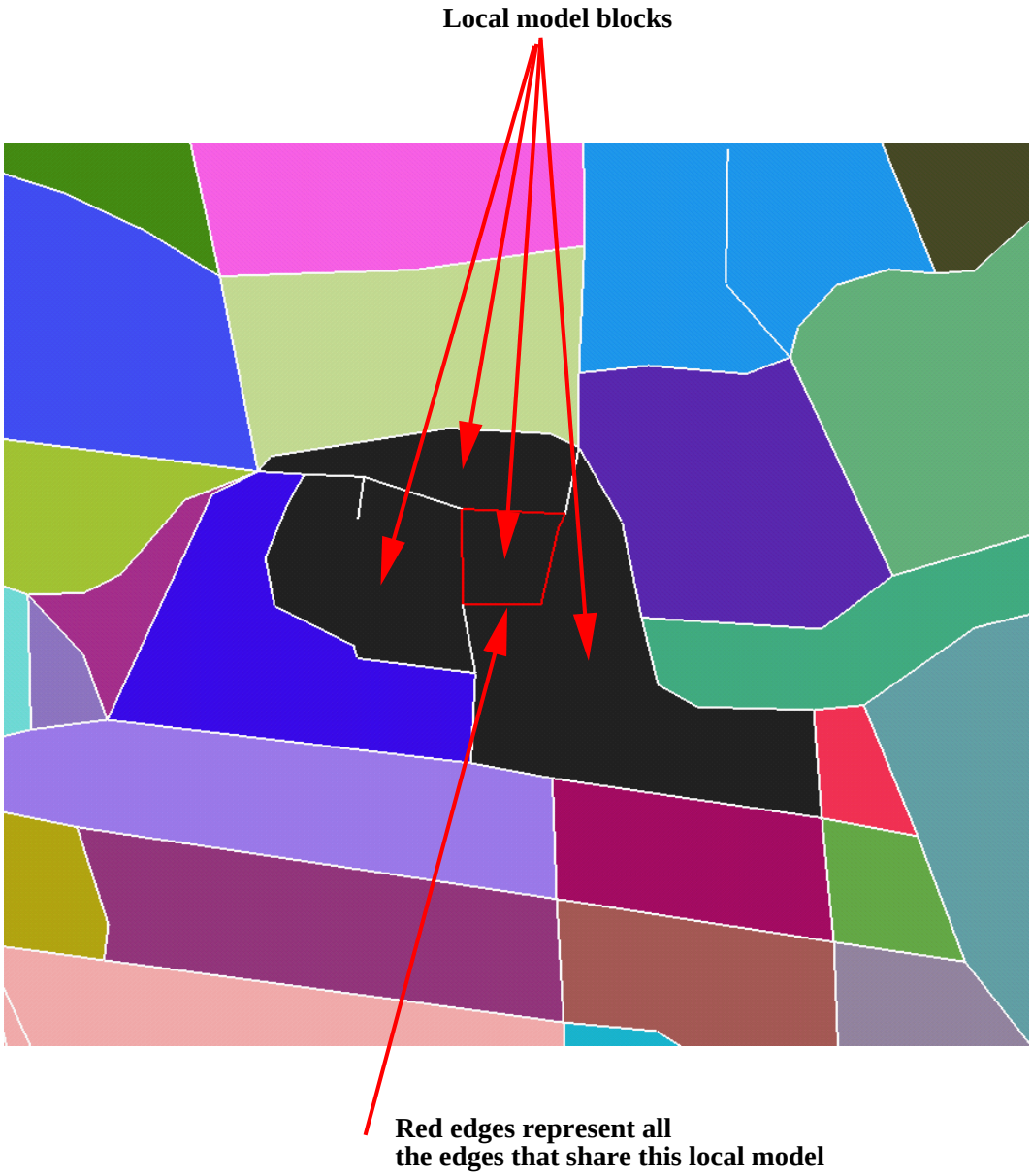


Figure 5-10: Local model visualization

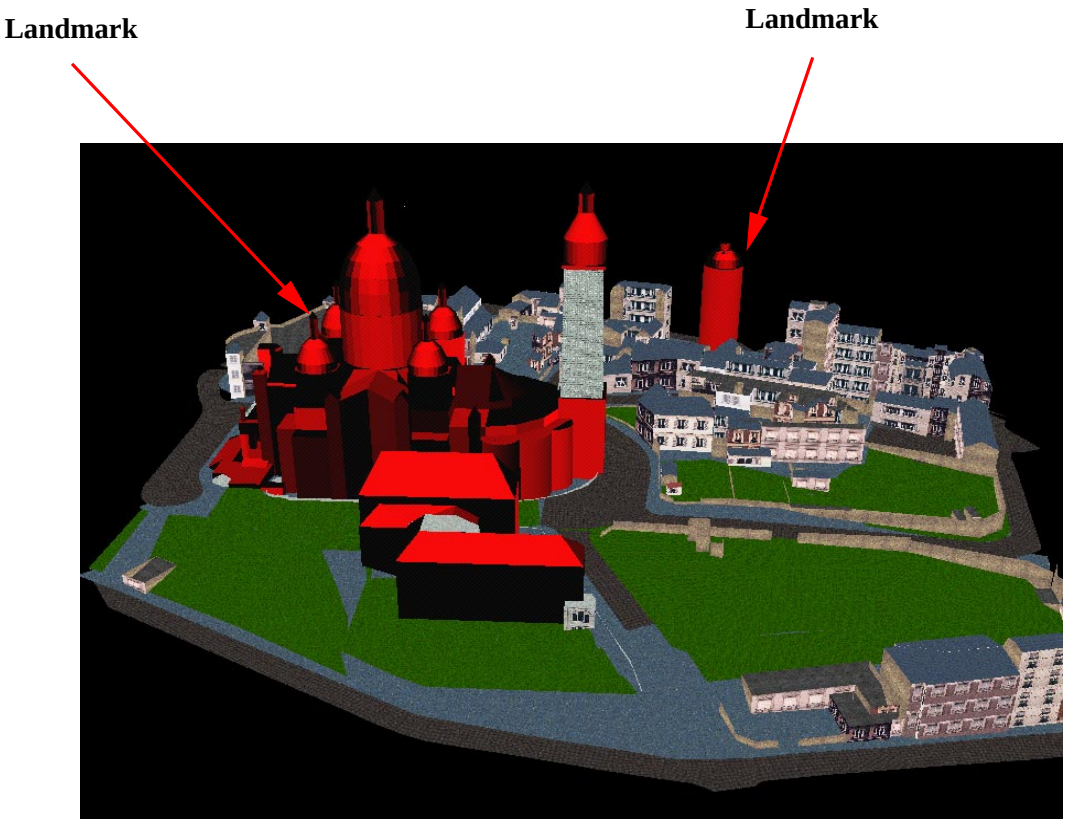


Figure 5-11: Landmark visualization

Chapter 6

Ville: Visualization

How are all the data structures used to visualize and navigate an urban scene? This chapter will explore the operations needed for the visualization process. It will discuss all the different modules, how they interact with each other, and how the data structures are used.

6.1 System design and modules

Figure 6-1 shows a high level overview of the modules that comprise the visualization program, "Ville". These modules can be grouped into two major parts: loading and visualization. Loading deals with reading in the model and populating the city database, while visualization deals with all the processes associated to render a frame.

6.1.1 CIT reader

This module is the system's interface to loading in the model and all its associated data structures. It reads the CIT file (appendix B), and constructs all the data structures discussed in chapter 3. The reader also accounts for big and little endian differences found between hardware architectures.

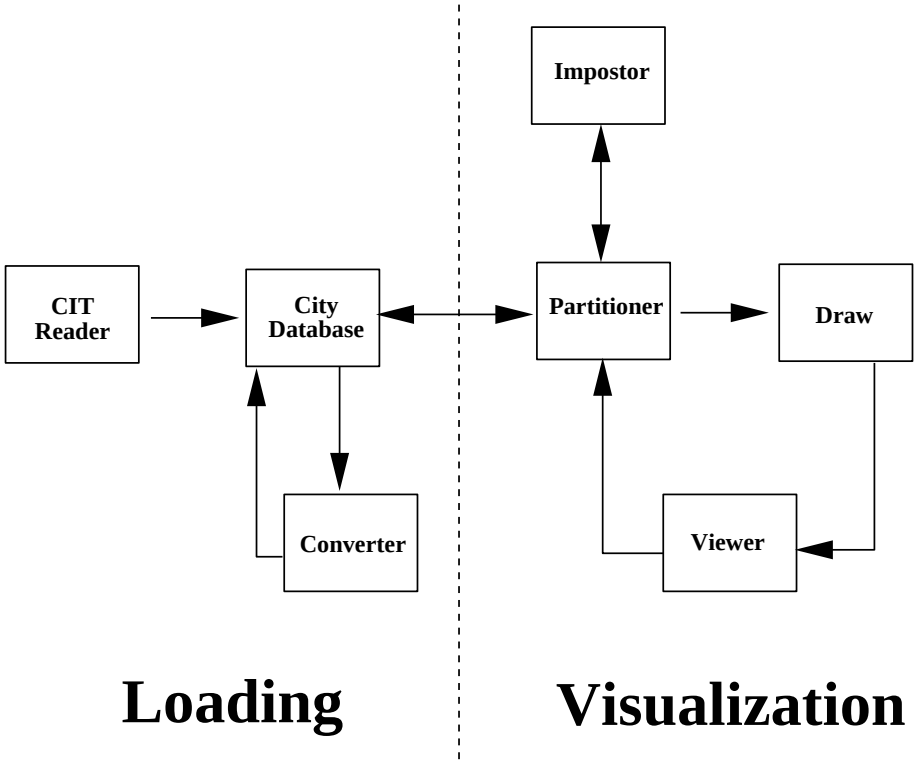


Figure 6-1: Ville modules

6.1.2 City database

This module contains all the data structures. It contains the segmentation information, winged edge data structure, city geometry, linked objects, and impostor parameters. All database queries are made to this module.

6.1.3 Converter

The converter deals with converting all the geometry into displayable objects. The current implementation converts all the generic geometry representations into *IRIS Performer* objects.

6.1.4 Partitioner

The partitioner is the heart of the system. It keeps track of which edge the user is on, and the current local, far, and impostor models. This is where all the information associated with visualization is updated. It deals with view location, updating impostors, traversing the winged edge data structure, and updating the models lists. It can run in two modes: flying and walking. Depending on the mode, data is updated accordingly.

6.1.5 Draw

This module receives camera parameters and geometry to display. It applies traditional frustum culling techniques, and renders all the visible geometry to the display. All the information it receives is updated by the partitioner.

6.1.6 Viewer

This module accounts for the callbacks associated with where the user wants to be and how the user travels around the environment. It sends all its messages to the partitioner. It work in two modes: "walking" and "flying". Walking involves walking around the environment along the edges defined in the map, while with flying the

user is able to travel anywhere in the scene. Mouse and key events are appropriately sent depending on the mode.

6.1.7 Impostor

The Impostor module is responsible for off-screen rendering, extracting the frame and z buffers. It is also responsible for creating impostors from the images.

6.2 Process

At the start of visualization the partitioner is initialized with an edge. At this point the user decides what visualization mode to use. Visualization modes include: local model only, far model only, local mode and impostor, and impostor only. Regardless of the mode, the partitioner constantly updates the local and far model lists, but depending on the visualization mode the partitioner sends the appropriate list to the draw process. When the user in using any of the impostor modes and if an impostor is not cached with the current edge, the partitioner calls the impostor module requesting it to create one or more impostors depending on the information contained in the edge. The impostor module creates the impostor and returns it to the partitioner. The partitioner updates its model lists and sends the draw list to the draw routine.

When the user moves around the scene, the partitioner updates where the user is in the winged edge data structure (only in walking mode). In addition, the partitioner queries the edges' local model and impostor parameters and then updates its lists of local and far models. If needed, it will call the impostor module to create an impostor. When in flying model, only the camera location is updated.

6.3 Impostor generation

As discussed earlier, far geometry is represented by a series of impostor images, which accordingly are used to construct the impostor. The impostor generation technique we use is based on Sillion *et al.*'s work [SDB97], the details of which can be found

in Section 1.2.4. Before the images of the impostor are constructed, camera parameters and the far model are sent to the impostor module. Using this, it determines how many images to generate and which geometry, to use. After the images are constructed, the impostors are created.

Chapter 7

Urban Morphology

In this chapter we explore issues related to urban morphology. We attempt to exploit legibility by identifying and constructing a number of representations, and allowing users to manipulate the scene. Algorithms for the identification of the different morphological components are discussed. Data structures and techniques for using morphology will be presented.

This chapter is organized as follows. Section 7.1 will motivate why urban morphology is useful for the visualization of urban environments. Section 7.2 defines a number terms used throughout the chapter. Section 7.3 identifies the problems with the Sillion *et al.* system that could potentially use urban morphology for a solution. Section 7.4 will provide visualizations of the information used for the identification of morphology. Section 7.5 provides the identification algorithms. Finally, in Section 7.6 will present a number of solutions based on urban morphology.

7.1 Motivation

Little research in computer graphics has attempted to use the morphology of 3D environments to improve high-performance visualization. The majority of the work develops techniques that ignore any specific structure inherent in the environment. Understanding morphology introduces the potential to accomplish previously unexplored approaches.

Of the diversity of environments used for high-performance visualization, cities form a unique subset. They possess a very clear sense of structure [Lyn60, Lyn96, Kos91, Bur71, Mum61], often called the "legibility of the cityscape", and their visual experiences strongly depend on the city's morphology. The structure of a city can be understood as the superposition of spatial, social, and historical relationships. The obvious spatial structure consisting of streets, parks, and built areas is important, but the history of the city development and its economic/social organization are also important for understanding urban environments.

Given a visualization scenario, users' visual experiences strongly correlate to how they interact with the environment. Knowing "important" features, for example, enables visualization algorithms to devote more computational power to these features as compared to structurally less important features. New representations may even be introduced to deal with these features. Also, knowing how a user at a particular location will probably behave, algorithms can predictively adapt.

7.2 Definitions

Before we embark on our study of urban morphology, a number of terms used throughout this chapter must be defined. Below are the terms used and their definitions:

End Point An end point is the point that starts or ends a street. Thus a street segment is composed of two end points.

Paths Paths are the channels along which the observer moves. They may be streets, walkways, transit lines, canals, railroads, et cetera. These are dominant elements in a city since observers use them to move around a city. A path is composed of one or more street segments starting and ending with an end point that has 1 or more than 2 street segments connected to it. All the in between endpoints are shared by 2 street segments.

Street Segment Street Segments are portions of paths. They are straight lines and can be of any length. A path is composed of one or more street segments. A

street segment has two end points.

Block Blocks are the smallest unit of organization and define the subset of buildings that contains no paths. The buildings may be densely or sparsely packed in the block.

City Edge City Edges are the linear elements not used or considered as paths by the observer, e.g. shores, railroads cuts, edges of developments, walls, bridges. City Edges are barriers between different regions. They are important organizing features, especially in the role of holding together generalized areas, as in the surrounding of a city by water or walls.

Nodes Nodes are points -the strategic spots in a city into which an observer can enter- that are the intensive foci to and from which he is traveling. These may by primary junctions, places of a break in transportation, a crossing or convergence of paths, airports, railway stations, moments of shift from one structure to another, or concentrations. Nodes are found in almost every city and may be the dominant feature.

Square A square is a type of node. Its underlying feature is that it is composed of terrain and contains very few buildings. Squares can be in the form of a garden, large gathering area, course yard, et cetera.

Intersection An intersection is a type of node. It is represented by a break in a path. The break is in the form of a series of streets a observer can choose to continue and travel in.

Landmarks Landmarks are point-references, which the observer does not enter; they are external. They are a simply defined object: building, sign, store, or mountain. Their use involves the singling out of one element from a host of possibilities. Some landmarks are distinct ones, typically seen from many angles and distances.

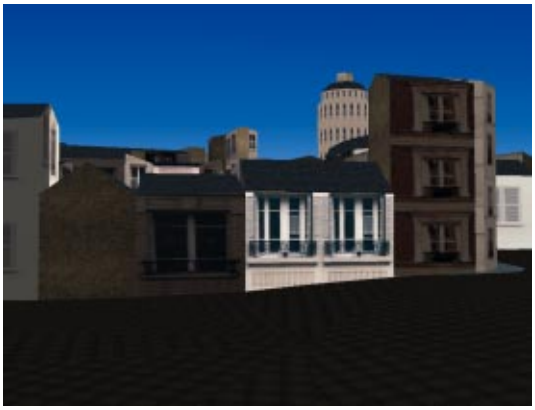
7.3 Analysis of Sillion *et al.*'s system

The system described by Sillion *et al.* [SDB97] offers interactive visualization of urban scenes, demonstrating the effective use of a hybrid geometry and image based approach. However, the approach breaks down in a number of cases. The fundamental problem is a result of the assumption that buildings occlude the viewer on either side of the street. Thus, when a viewer decides to look at the side of the street, the buildings occlude distant geometry. However, this does not occur in a number of cases: squares, intersections, edges, short surrounding buildings, and short street segments.

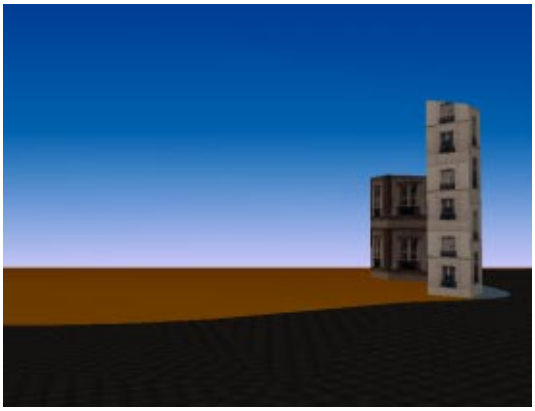
When a user is in a square, the block comprising the square contains very little geometry occluding the viewer. If the user decides to look inside the square, distant geometry is not occluded and not represented with an impostor. This results in the user seeing empty spaces, as seen in Figure 7-1b. What the user should be seeing is depicted in Figure 7-1a. The reason distant geometry is missing is that the local model is defined to contain all the blocks on either side of the street and the impostors are created in the direction of the street, as seen in Figure 7-1c. Since the user has the option to take viewing directions perpendicular to the direction of the street, no occlusion occurs in the square and empty spaces are seen.

The second case where this problem occurs is at an intersection. Here the problem is similar to the square, but happens for a different reason. It appears because the viewer is able to see down multiple streets at the intersection. There is no building geometry in the street; this enables the user to see far beyond the local model, in viewing directions different from the original street the user is on. Figure 7-2b shows a view taken at an intersection. When this view is compared to one rendered using the complete geometry (see Figure 7-2b), it is noticeable that a portion of the far geometry is not represented. To illustrate this further Figure 7-2c shows an overhead view of the local model and impostors.

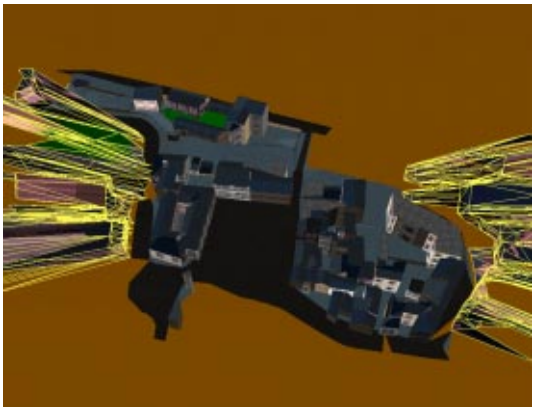
The third case occurs at a city edge. The problem faced here is similar to the square case. However, the difference is that no occlusion occurs on one or both sides



(a) - street level view using the complete model

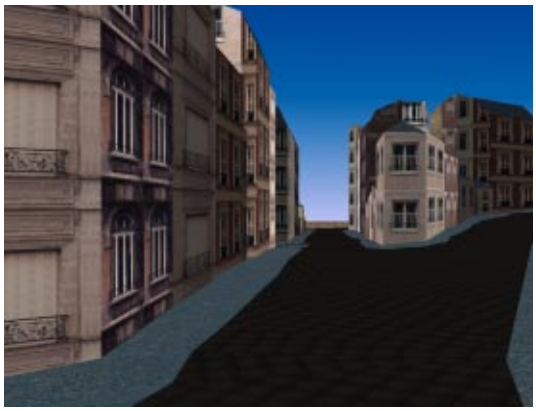


(b) - street level view using the Sillion *et al.*'s hybrid model

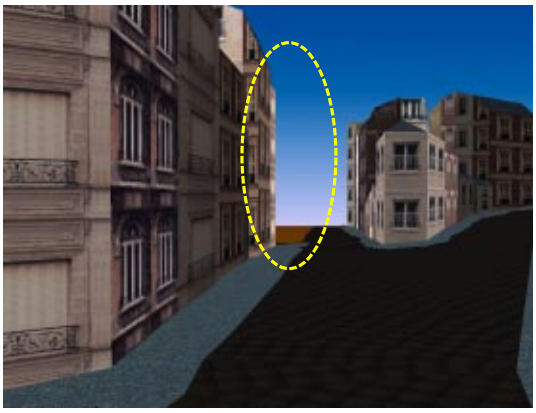


(c) - overhead view using the Sillion *et al.*'s hybrid model

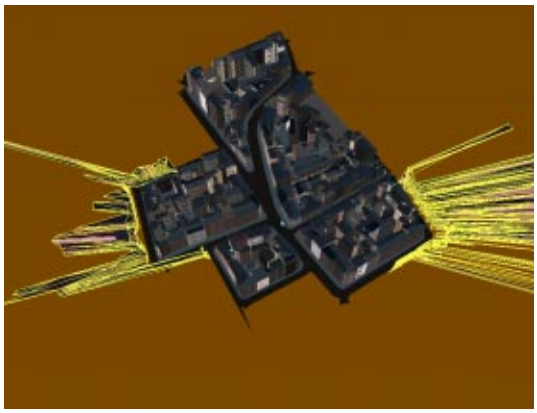
Figure 7-1: A comparison between the complete model view and geometry/impostor representation of a square in the original Sillion *et al.* system. Notice the differences between the image (a) and (b). A large portion of the model is missing. Image (c) shows an overhead view of the local model and the impostors used for the distant geometry.



(a) - street level view using the complete model

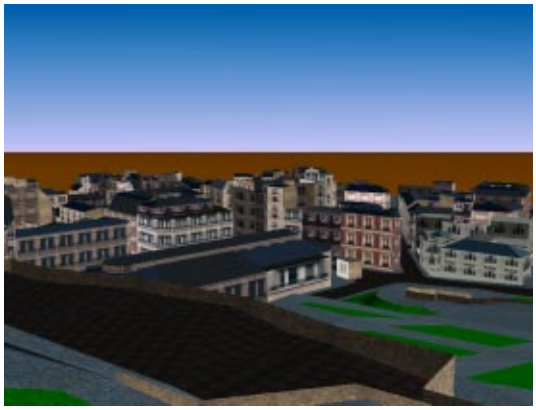


(b) - street level view using the Sillion *et al.*'s hybrid model

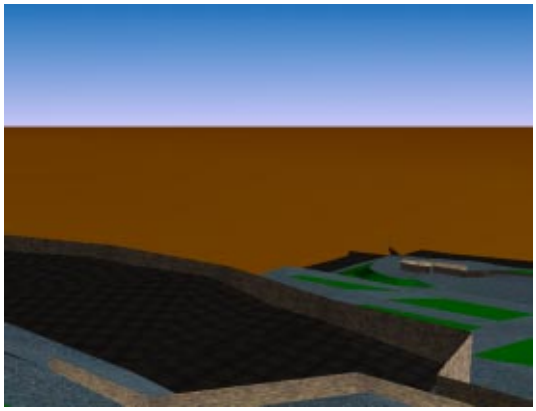


(c) - overhead view using the Sillion *et al.*'s hybrid model

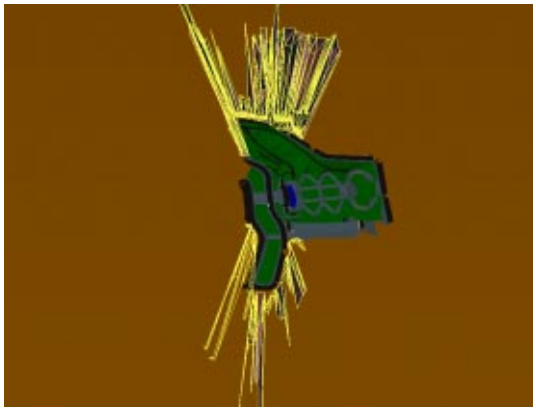
Figure 7-2: A comparison between the complete model view and geometry/impostor representation of an intersection in the original Sillion *et al.* system. Notice the differences between image (a) and image (b) at the end of the street circled in yellow. This portion of the model is missing because impostors are computed at this viewing direction. Image (c) shows an overhead view of the local model and the impostors used for the distant geometry.



(a) - street level view using the complete model



(b) - street level view using the Sillion *et al.*'s hybrid model



(c) - overhead view using the Sillion *et al.*'s hybrid model

Figure 7-3: A comparison between the complete model view and geometry/impostor representation of a city edge in the original Sillion *et al.* system. Notice the differences between image (a) and image (b). A large portion of the model is missing. Image (c) shows an overhead view of the local model and the impostors used for the distant geometry.

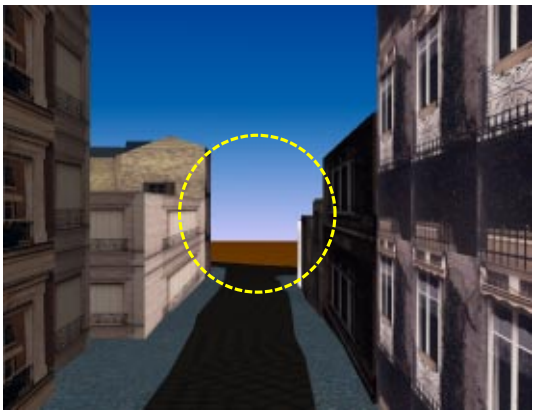
of the street. Figure 7-3b shows a view at a city edge. When compared to a rendering using the complete model Figure 7-3a, it is apparent that a large portion of the model is missing. To illustrate the problem further, Figure 7-3c shows an overhead view, identifying the local model and the impostors.

A problem also exists in the case of very short street segments. For example, if a short street segment is connected to two other streets directed almost orthogonally to the short street, the impostor representation fails. It fails because impostors are created in the direction of the short street, even though the travel time in these short street segments is momentary. Thus, if the user wishes to travel in a direction orthogonal to the short street, for a moment, there are no impostors representing the distant geometry. Figure 7-4 shows comparative views (street level and overhead views) of a location showing an occurrence of the problem.

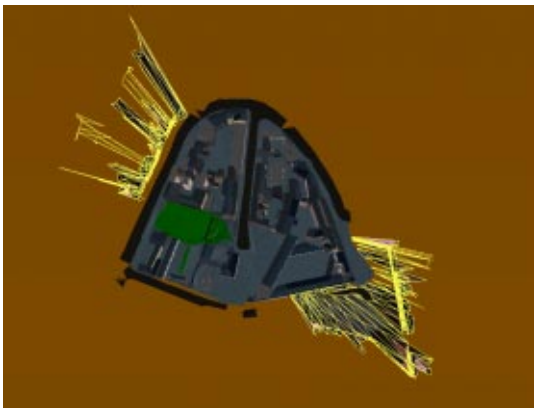
The final failure point can be represented in cases where users are able to view between and above peripheral buildings. The problem occurs in three cases. The first occurs when buildings are not densely packed on the sides of the street. The second case happens when builds are too short. Finally, the problem appears when the far geometry is too tall. This enables the user to see beyond the block. Impostors do not exist at these viewing directions, and consequently the distant geometry is not represented. This problem, however, is more apparent with landmarks since users are accustomed to seeing them from a number of locations. Figure 7-5 shows comparative views (street level and overhead views) of a location showing the occurrence of the problem with distant buildings. Figure 7-6 shows comparative views (street level and overhead views) of the another location of an occurrence of the problem with a landmark.

We attempt to address the cases defined above by introducing representations that exploit the urban morphology. Accordingly, visualization will adapt depending on where the user is located.

(a) - street level view using the complete model

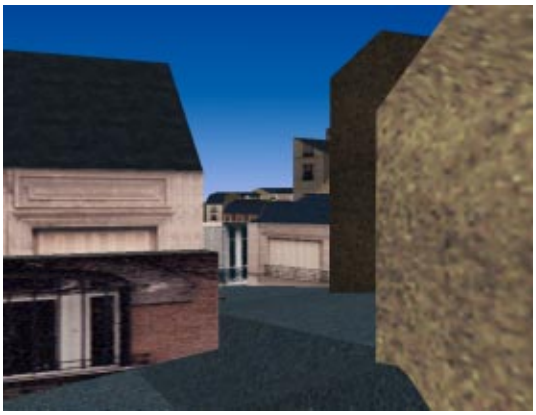


(b) - street level view using the Sillion *et al.*'s hybrid model

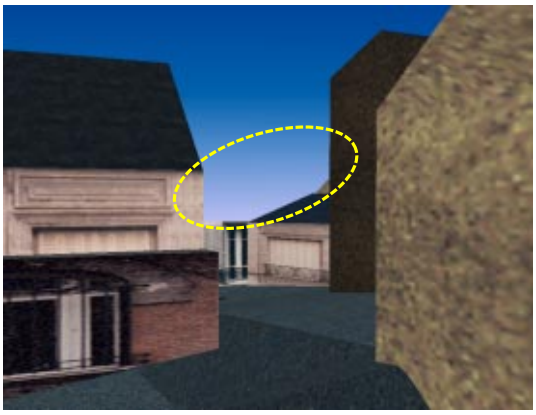


(c) - overhead view using the Sillion *et al.*'s hybrid model

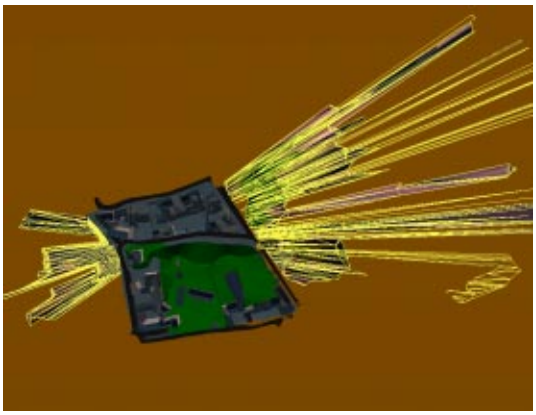
Figure 7-4: A comparison between the complete model view and geometry/impostor representation used to illustrate the problems of short street edges in the model in the Sillion *et al.* system. Notice the differences between image (a) and image (b). The circled portion in image (b) is missing because impostors are created using the wrong viewing direction. Image (c) shows an overhead view of the local model and the impostors used for the distant geometry.



(a) - street level view using the complete model

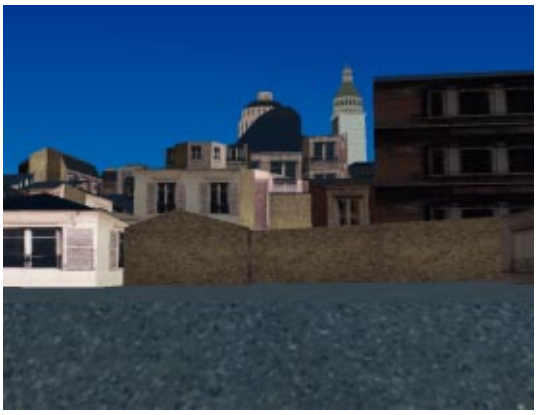


(b) - street level view using the Sillion *et al.*'s hybrid model

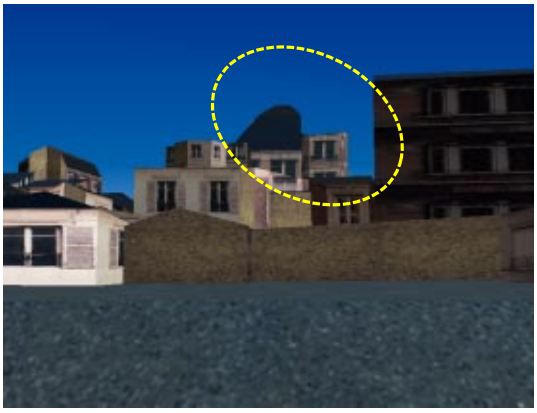


(c) - overhead view using the Sillion *et al.*'s hybrid model

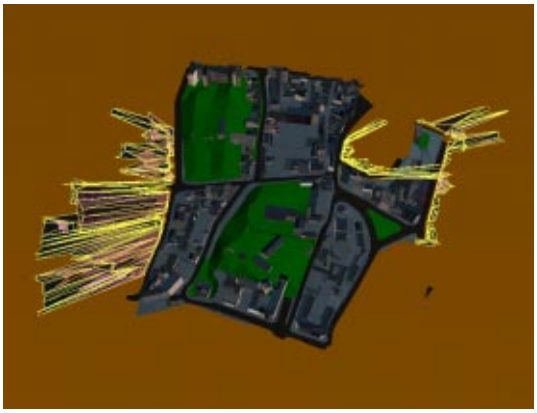
Figure 7-5: A comparison between the complete model view and geometry/impostor representation used to illustrate the problems of seeing past short peripheral buildings in the Sillion *et al.* system. Notice the differences between image (a) and image (b). The circled portion in image (b) is missing because impostors are created using a different viewing direction. Image (c) shows an overhead view of the local model and the impostors used for the distant geometry.



(a) - street level view using the complete model



(b) - street level view using the Sillion *et al.*'s hybrid model



(c) - overhead view using the Sillion *et al.*'s hybrid model

Figure 7-6: A comparison between the complete model view and geometry/impostor representation used to illustrate the problems of seeing landmarks past short peripheral buildings in the Sillion *et al.* system. Notice the differences between image (a) and image (b). The circled portion in image (b) is missing because impostors are created using a different viewing direction. Image (c) shows an overhead view of the local model and the impostors used for the distant geometry.

7.4 Urban conditions

The city elements that are of interest are: squares, intersections, edges, and landmarks. The urban planning definitions of each element are used for the identification algorithms.

In order to get a sense of how identification is performed, Figure 7-7 shows a visualization of the entire Paris model. The figure is an overhead view of the model, where each building is encoded with its relative height. By using heights and building densities, represented by this visualization, the system can infer the locations of the urban characteristics, shown in Figure 7-8.

7.5 Identification

Urban characteristic identification is an important component of Genesis (discussed in Chapter 5). The algorithms used are derived from the urban planning definitions of urban characteristics. All of the characteristics can be identified manually by inspection. However, we also describe techniques for identifying these characteristics automatically. Each characteristic and its associated identification techniques will be discussed in the following sections.

7.5.1 Squares

In a model, a square is a viewer accessible block that internally contains no building geometry. The observer should have the freedom to walk within the block. A square can be in the form of a garden, a large gathering area, a court yard, et cetera. Figures 7-9 show a series of squares in the Paris model and the blocks that surround them.

Identification

- A block is a square if its building density is concentrated around the periphery of the block.

Figure 7-7: An overhead view of the Paris model: encoded in the buildings are their relative heights. Dark red buildings represent the lowest buildings while bright red buildings represent the tallest buildings.

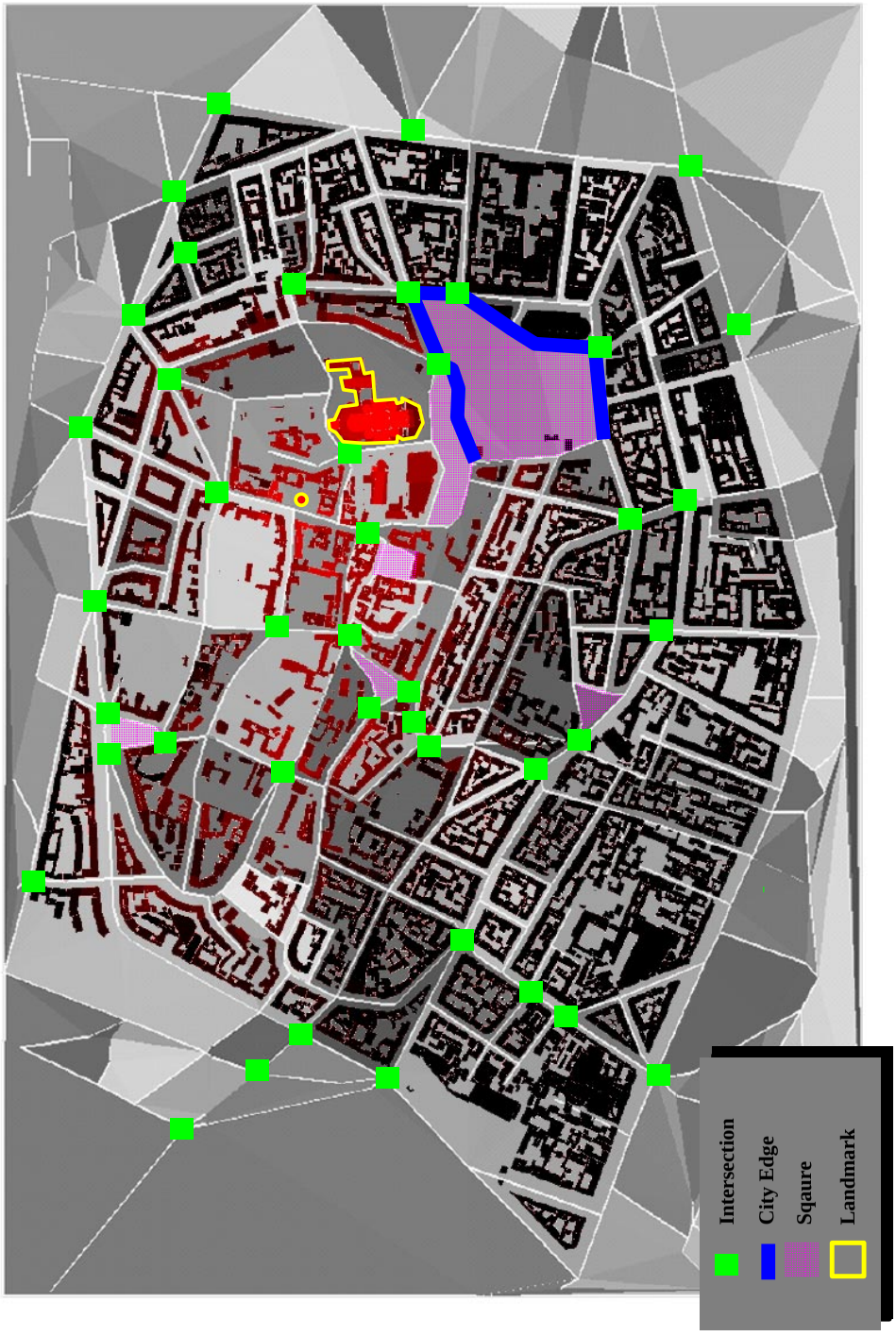


Figure 7-8: Urban features identified in the Paris model.

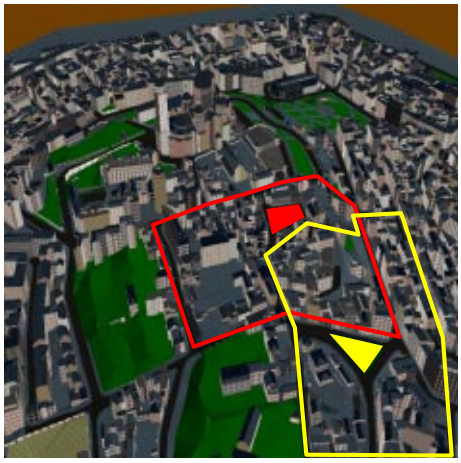


Figure 7-9: Squares in the Paris model

- By using the names of areas in the city. For example, "Copley Square" is a representative example. It can be identified as a square if the model contains this information.

Automated detection algorithm

$\mu \leftarrow \prec$*pre-defined value*$\succ$
foreach *block* $\in$ *city.blocklist*
  bbox $\leftarrow \prec$*block bounding box*$\succ$
  $\prec$*shrink bbox by* $\mu \succ$
  if ($\prec$*all buildings outside bbox*$\succ$) **then**
    AddSquare (*block*)

This algorithm is rather simple and computationally non-intensive. It is an approximate approach to computing the building density (using bounding boxes) in the midst of the block.

7.5.2 Intersections

An intersection is a type of node and is represented by a break in a path. The break is in the form of a series of streets an observer can choose to travel through. For our

purposes, a break should be identified as an intersection if it exhibits a very diverse change in viewing conditions. Specifically, intersections have very variant viewing geometry along the different streets (viewing directions). Figure 7-10 shows a series of intersections in the Paris model with the blocks that surround them.

Figure 7-10: Intersections in the Paris model

Identification

- An intersection can be identified by looking at the number of streets at an intersection and the angles between them.

Automated detection algorithm

foreach *point* $\in$ *map.pointlist*
    if (*point.edgelist.count* > 4) **then**
        AddIntersection (point)
        continue
    if (*point.edgelist.count* $== 4$) **then**
        if ($\prec$*smallest angle between any pair of edges in point.edgelist*$\succ < \theta$) **then**
            AddIntersection (point)

The above algorithm is computationally non-intensive and detects intersections that fall into two categories. The first selects intersections with greater than four streets. The second looks at intersections with exactly four streets and searches for streets that are very close. The latter assumes that the subset of valid intersections (composed of four streets) are those that contain non-orthogonal streets.

7.5.3 City edges

City edges are the linear elements not used or considered as paths by the observer, e.g. shores, railroads cuts, edges of developments, walls, bridges, et cetera. City edges are barriers between different regions. They have important organizing features, especially in the role of demarking specific areas. A city edge can be thought of as being a street near a relatively large open space that overlooks portions of the urban environment.

Figure 7-11: Edges in the Paris model

Identification

- A city edge can be identified by looking at the surrounding building densities.

Automated detection algorithm

$\ell \leftarrow \prec$*per-defined value*$\succ$
foreach *edge* $\in$ *map.edgelist*
    bbox $\leftarrow \prec$*create 2d bounding box oriented by edge with*
        bbox.width $= \ell$ *and bbox.height* $=$ *edge.length*$\succ$
    lblock $\leftarrow$ *edge.leftblock*
    rblock $\leftarrow$ *edge.rightblock*
    if ($\prec$*all buildings in lblock outside 2d bbox*$\succ$) **then**
        AddEdge (*edge*)
    else if ($\prec$*all buildings in rblock outside 2d bbox*$\succ$) **then**
        AddEdge (*edge*)

The algorithm searches for buildings in an oriented bounding box. The bounding box has the street as its central axis. Each building's bounding box is used to test overlaps with the oriented bounding box. This, again, searches for buildings in the left (lblock) and right blocks (rblock), in this oriented bounding box. If there are none, then the edge is defined as a city edge.

7.5.4 Landmarks

Landmarks for our purposes are buildings that act as reference points with respect to the environment that surrounds them.

Identification

- A landmark can be identified by examining the surrounding building densities and building heights.
- It can be performed by using a guidebook and then manually identifying the buildings.

Figure 7-12: Landmarks in the Paris model.

Automated detection algorithm

$\mu \leftarrow \prec$*pre-defined value*$\succ$
$\alpha \leftarrow \prec$*pre-defined value*$\succ$
foreach *block* $\in$ *city.blocklist*
  foreach *building* $\in$ *block.buildinglist*
    if (*building.height* $>$ *h*) **then**
      AddLandmark (*block, building*)
      continue
    bbox $\leftarrow \prec$*get building bounding box*$\succ$
    $\prec$*expand bbox by* $\mu \succ$
    SmallerBlocklist $\leftarrow \prec$*all other buildings with*
      height $>$ *building.maxheight*$+\alpha \succ$
    if ($\prec$*all buildings in SmallerBlocklist outside bbox*$\succ$) **then**
      AddLandmark (*block, building*)

The above algorithm uses building densities and building heights to identify landmarks that stand out from the surrounding cityscape. The algorithm looks for buildings that surround the current building. If there are none, then this building stands out from its context and is visually a landmark. If all the surrounding buildings are shorter than the building in question, then the building is identified as a landmark.

7.6 Exploiting urban morphology for improved visualization

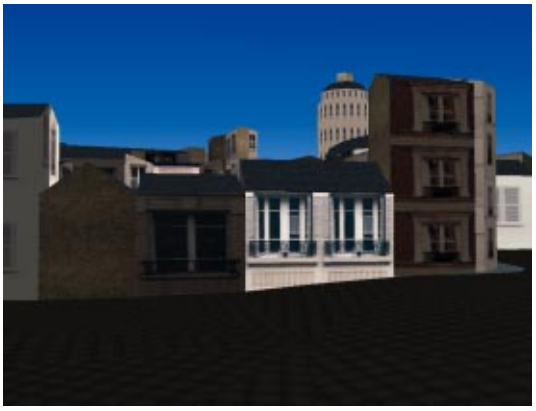
How can morphology be used to increase the fidelity of the user's visual experience? In this section we present a preliminary analysis of several possibilities. Different treatments are introduced for different urban conditions, defined under the framework of linking (discussed in section 7.6.1). We will look at each individually, developing and proposing solutions for each case (Section 7.3).

7.6.1 Linking

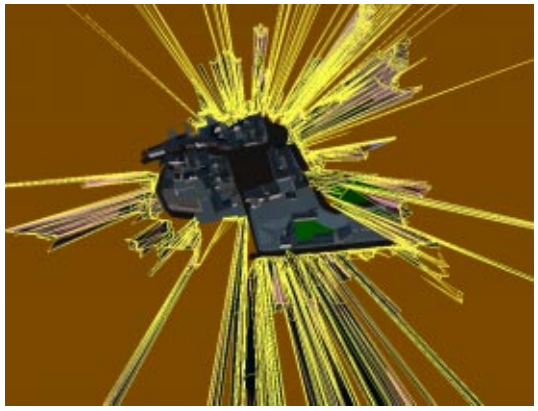
Linking is the framework needed to connect objects together under a defined property. Linked objects are composed of streets, blocks and/or buildings. Each linked object is given a property based on the contents of the link. The properties linked objects can represent are: squares, intersections, edges, and landmarks.

7.6.2 Squares

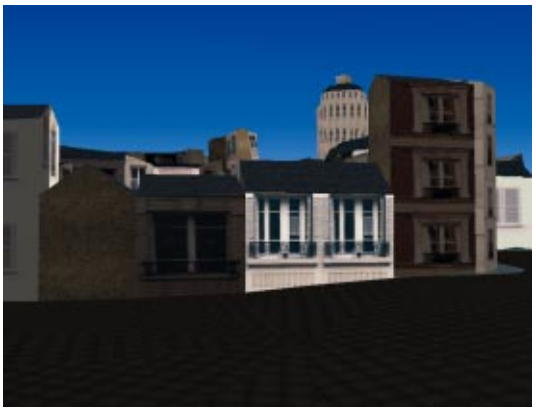
To overcome the problems associated with visualizing squares (seen in Figure 7-1 and discussed in Section 7.3), we use an alternate representation that uses the knowledge provided by the morphology to segment the model and create the associated impostors. Figure 7-13 shows a series of views of the alternate representation. Figure 7-13a shows the view using the complete model. Figure 7-13c and Figure 7-13d show the same view using our representation. Finally, Figure 7-13b shows an overhead view of the geometry depicting both the impostor and local model. From the figures, the definitions of the local and far models are improved, and the view is a much closer approximation of the complete model. Below are the details used to construct the representation associated with a square.



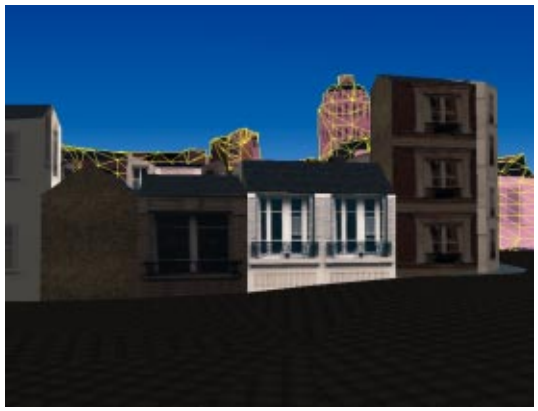
(a) - street level view
using the complete model



(b) - overhead view using the morphology
information concerning squares



(c) - street level view using the morphology
information concerning squares



(d) - same as (c) with the
impostor highlighted

Figure 7-13: A comparison, at a square, between views constructed using the complete model (a) and a geometry/impostor based representation (b,c,d). The geometry/impostor based representation was constructing using the morphology information concerning squares.

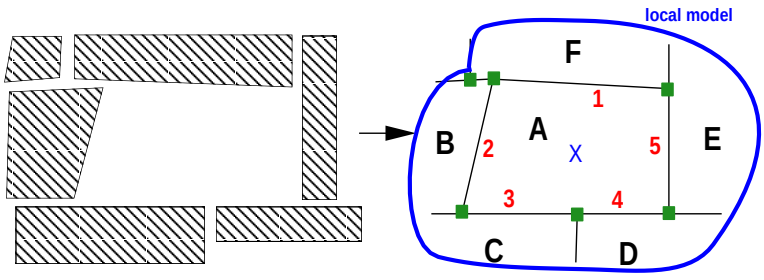


Figure 7-14: Square's local model.

Representation

A square defined as a linked object is composed of a block and the edges that surround it. In Figure 7-14, edges 1-5, and block A are linked together to define a square.

Impostor

Figure 7-15: A visualization of the square's local model and the associated cameras used for the impostors.

In Figure 7-14, edges 1-5, and block A are linked together to define a square. Blocks A-F define the local model. Images used to create the imposter are generated by a series of cameras positioned in the center of the block and cover the entire 360 degrees. Figure 7-15 shows how 8 cameras where used to partition the entire 360 degrees. Each camera had a field of view of 45 degrees.

7.6.3 Intersections

To overcome the problems associated with visualizing intersections (seen in Figure 7-2 and discussed in section 7.3), we provide an alternate representation that uses the knowledge of being in an intersection to segment the model and create the associated impostors. Figure 7-16 shows a series of views of the alternate representation. Figure 7-16a shows the view using the complete model. Figure 7-16c and Figure 7-16d show the same view using our representation. Finally, Figure 7-13b shows an overhead view of the geometry depicting both the impostor and local model. The definition of the local and far models have improved, and the view is a much closer approximation of the complete model. Below are the details used to construct the representation associated with an intersection.

Representation

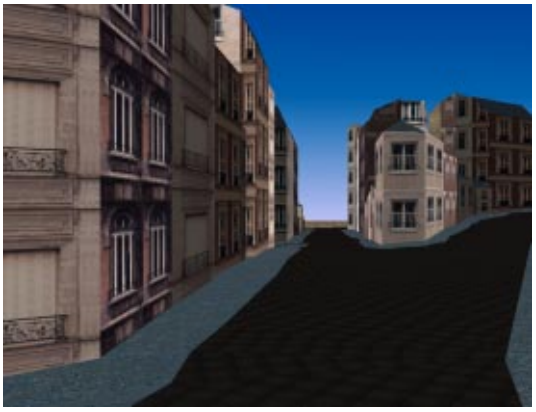
An intersection is represented in a linked object as the set of streets that compose the intersection. In Figure 7-17, edges 1-7 represent an intersection.

Impostor

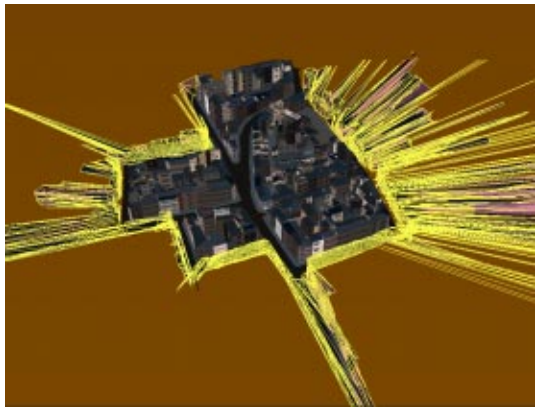
In Figure 7-17, edges 1-7 are linked together to define an intersection. The local model is defined by the blue outline and is represented by the blocks that surround the linked edges. Impostors are generated with cameras positioned at node A, i.e. the center of the intersection. The images should cover the entire 360 degrees. Figure 7-18 shows how 8 cameras where used to partition the entire 360 degrees. Each camera had a field of view of 45 degrees.

7.6.4 City edges

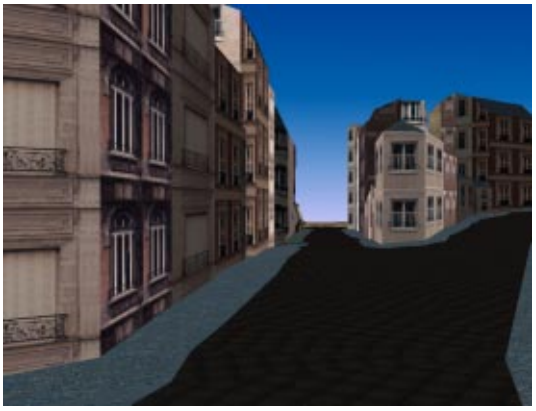
To overcome the problems associated with visualizing city edges (seen in Figure 7-3 and discussed in Section 7.3), we use an alternate representation that uses the knowledge of being in a city edge to segment the model and create the associated impostors. Figure 7-19 shows a series of views of the alternate representation. Figure 7-19a shows



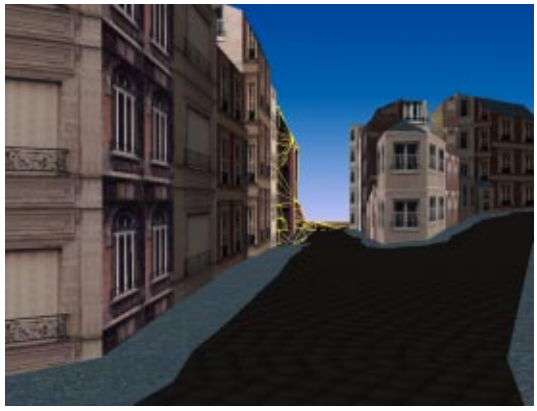
(a) - street level view
using the complete model



(b) - overhead view using the morphology
information concerning intersections



(c) - street level view using the morphology
information concerning intersections



(d) - same as (c) with the
impostor highlighted

Figure 7-16: A comparison, at an intersection, between views constructed using the complete model (a) and a geometry/impostor based representation (b,c,d). The geometry/impostor based representation was constructing using the morphology information concerning intersections.

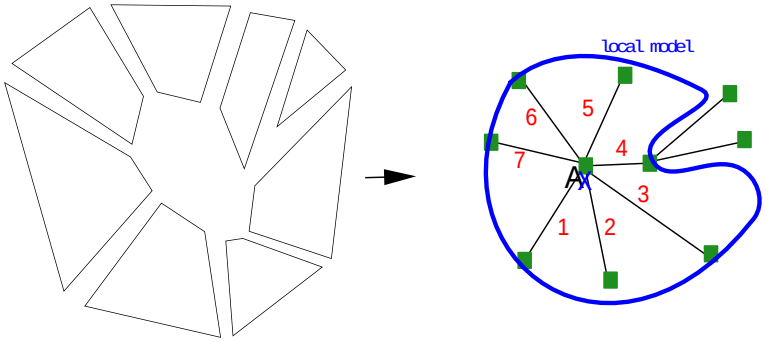


Figure 7-17: An intersection's local model.

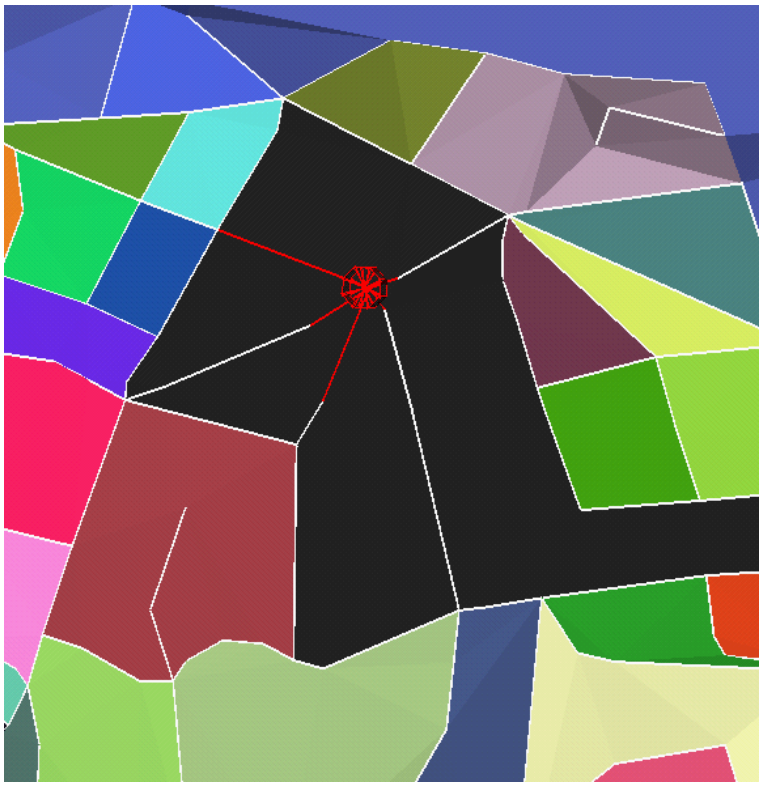
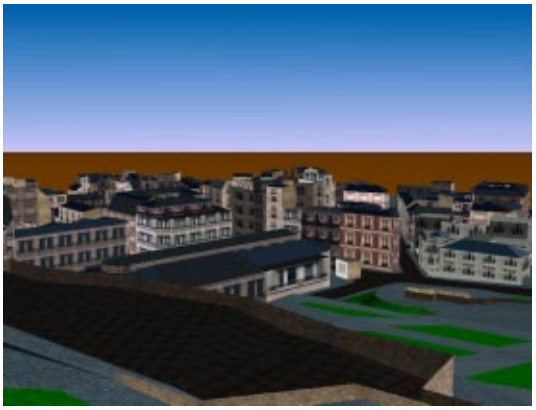
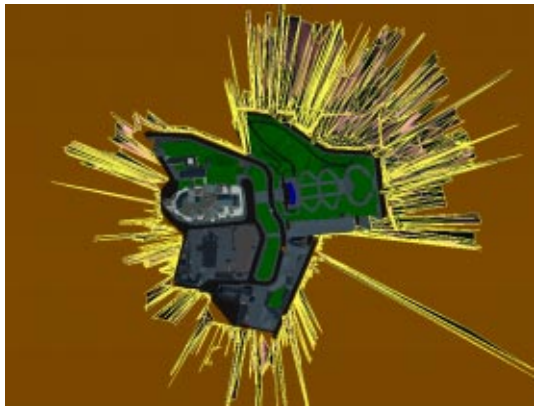


Figure 7-18: A visualization of the intersection's local model and the associated cameras used for the impostors.

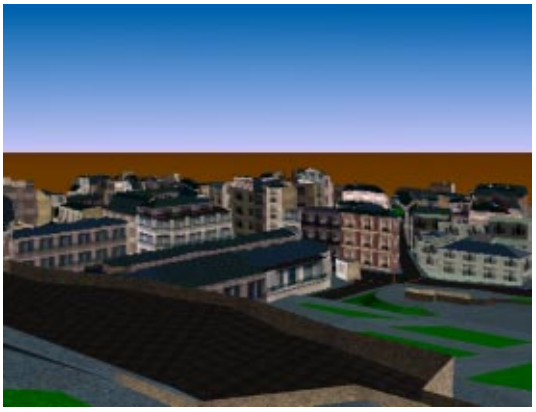
the view using the complete model. Figures 7-19c and 7-19d show the same view using our representation. Finally, Figure 7-13b shows an overhead view of the geometry depicting both the impostor and local model. From the figures, the definitions of the local and far models are improved and the view is a much closer approximation of the complete model. Below are the details used to construct the representation associated with a city edge.



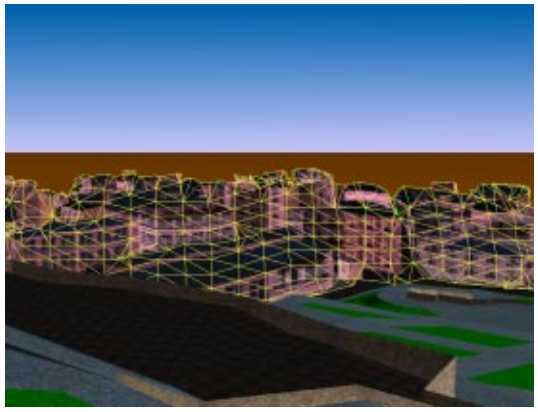
(a) - street level view using the complete model



(b) - overhead view using the morphology information concerning city edges



(c) - street level view using the morphology information concerning city edges



(d) - same as (c) with the impostor highlighted

Figure 7-19: A comparison, at a city edge, between views constructed using the complete model (a) and a geometry/impostor based representation (b,c,d). The geometry/impostor based representation was constructing using the morphology information concerning city edges.

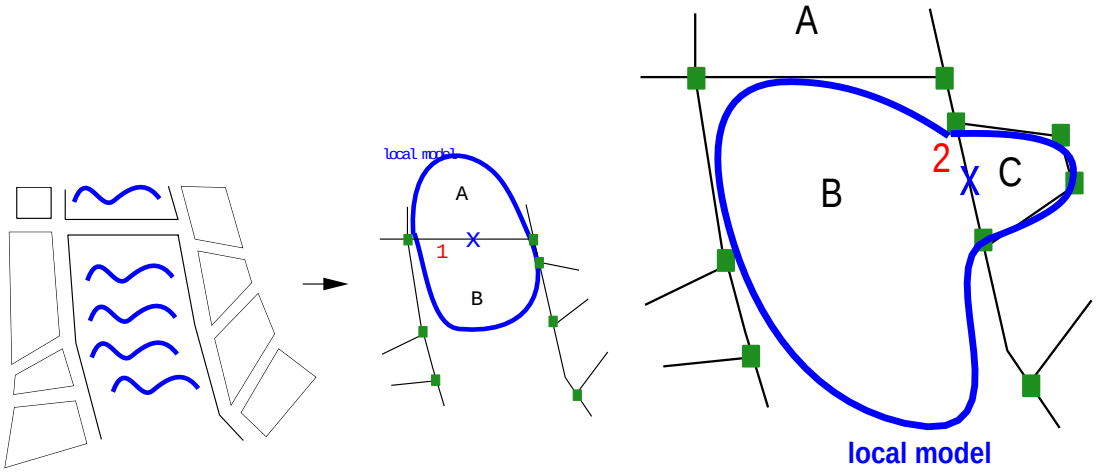


Figure 7-20: City Edges, and local models.

Representation

A City Edge is represented as a linked object composed of one or more streets. In Figure 7-20a, edge 1 is a city edge, and in Figure 7-20b, edge 2 is another city edge.

Imposter

In Figure 7-20a, edge 1 is added to a linked object to define a city edge. Blocks A, B define the local model. Impostors are generated at the center of edge 1, i.e. the center of the bridge.

In Figure 7-20b, edge 2 is added to a linked object to define a city edge. Blocks B and C define the local model. Impostors are generated at the center of edge 2, i.e. the center of the street.

Images used to create the imposter are generated by a series of cameras that are positioned at the center of the edge and cover the entire 360 degrees. Figure 7-21 shows how 8 cameras where used to partition the entire 360 degrees at a city edge. Each camera had a field of view of 45 degrees.

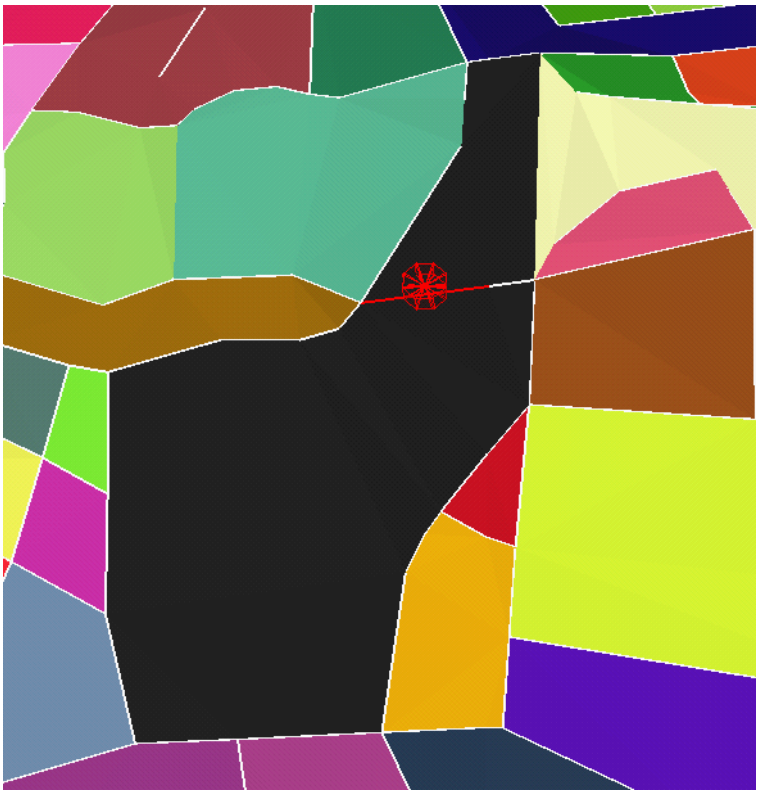


Figure 7-21: A visualization of the city edge's local model and the associated cameras used for the impostors.

7.6.5 Proposals

Connecting the linked objects (cross-linking)

Linked objects have to know about other linked objects with which they share objects. We call this *cross-linking*. Linked objects contains pointers to other linked objects they share overlaps with. Thus, after links are constructed, overlaps between links are identified and the information is stored in each of the overlapping linked objects. For example, when a square is connected to an intersection. In this case, both the square and the intersection have overlapping edges. Each linked object representing the square and intersection contains a pointer to the other, and thus each would know about the existence of the other.

Cross-links would be very useful during visualizations because of the varying visual conditions they introduce. Special treatments may be introduced when moving from a square to an intersection, or if a square is near a landmark. Also, transitioning between two or more characteristics is easier to perform when this information is easily at hand.

Landmarks

A landmark is an association of a building and the block with which the building is a member. The linked object, thus, contains a block (that the landmark is a part of) and the building identified as a landmark. The block containing the landmark is necessary for the construction of cross links.

Currently, the system does not use the landmark information for visualization, but its potential uses would improve visualization. The underlying idea is that during impostor creation, the knowledge that a building, in an image, is a landmark enables the impostor creation algorithm to adapt and try to do a better job of representing the landmark as compared to the other parts of the image. This potentially can be achieved by using several images to construct the landmark part of the imposter, and/or using a denser mesh for the landmark part of the impostor.

Connectivity

In the case of the existence of cross links, connectivity between the urban conditions may be used to reduce the large memory swaps that occur when a user jumps from one urban condition to another. These swaps cause the local model to change and a large impostor is either computed or swapped in. To reduce this, the local model may be represented as the union of the local models that compose the cross links. Also, the field of view of both impostor representations may be merged and used to define a new field of view for further impostor creation. Care must be taken, however, when using this approach. The local model may end up being too large, and the size of the local model must be controlled. Distance from the current position should play a role when new blocks are added to the local model. If this distance is greater than some threshold, the local model should stop growing. Accordingly, all the visualization data structures need to be updated so that the system is able to know when to make the next impostor and local model switch.

To reduce the number of impostor computations, connectivity between linked objects may be used to reduce the number of images used. If two or more images share similar camera parameters, it should be possible to combine the two into one impostor computation.

Short orthogonal streets

The problem associated with short orthogonal streets (discussed in Section 7.3) can be approached in two different ways. The first can be done by eliminating the problem completely, and removing all occurrences of short streets that are orthogonally surrounded by other streets and reconnecting the resultant streets. The other approach is to detect their occurrence and use the surrounding street directions as viewing directions, defining where impostors are generated. Care must be taken, however, with overlapping impostors. Thus, combination algorithms should be constructed to combine images for further triangulation.

Views between and above peripheral buildings

The problem of not seeing any far geometry in views between and above peripheral buildings (discussed in Section 7.3) is rather complex because of its detection difficulty. In general, this issue is not too noticeable. However, the problem is more apparent with tall landmarks that are visible from most of the city. A possible solution is to keep track of where the landmarks are located relative to the user and to show them when they fall into the current viewing frustum. Multiple representations, composed from different viewpoints, should ideally be used for the landmarks. The representation chosen should be based on the viewing direction and the distance of the landmark from the user. For example, a textured quadrilateral is sufficient when the landmark is very far, and the impostor representation is more effective when the landmark is closer. This would result in giving the user the points of reference needed to navigate the urban environment as well as improving the user's visual experience.

Chapter 8

Results

The pipeline for the visualization of urban environments was described in the previous chapters. We will elaborate on our methods and report the performance of the system. We will present some examples in this section to demonstrate the generality of the system and how the current implementation deals with the different urban conditions.

8.1 Generalization and morphology

We present in Figure 8-1 and Figure 8-2 the current pipeline being used to visualize several models. The different stages in the system correctly construct all the data structures for their respective visualization. The modifications made to the data structures can be incrementally changed and thus modified to perfection.

The *initial* identification of the blocks, segmentation of the model, and identification of the urban structures takes approximately 15 minutes, while all other functions performed are interactive. The time taken to finalize the data structures is a function of the user's comfort with the system.

8.2 Visualization results

We present in Figure 8-1 and Figure 8-2 comparative images of different viewer positions in the same area, in the Paris model and a generated model. The images

in the center column are produced by the combination of the local 3D model and the impostor, while the images in the right column are produced using the complete model). The images in the central column are produced at a sustained rate of between 11 to 20 frames per second, while a complete 3D rendering only achieves between 1 to 2 frames per second (using traditional frustum culling techniques provided by the *Performer* high-performance graphics library).

Second, the three dimensional model impostors succeed in providing an accurate view of distance landscapes, taking into account some parallax changes.

Third, a small amount of cracking is visible when the user comes very close to the boundary between the local model and the impostor. Also, the life time of impostors corresponds to the time the user spends in a given street; the impostor remains active for periods whose size exceed several meters.

Finally, different impostors are automatically created in regions of the model that require more comprehensive treatment. These regions are automatically and interactively defined and modified from a general set of input files. Figure 8-5 shows different viewer positions at a square. Figure 8-4 shows different viewer positions at an intersection. Figure 8-5 shows different viewer positions at a city edge. In all the figures, the images in the center column are produced by the combination of the local 3D model and the impostor, while the images in the right column are produced using the complete model. The images in the left column are the same as the images in the center column with the impostors emphasized.

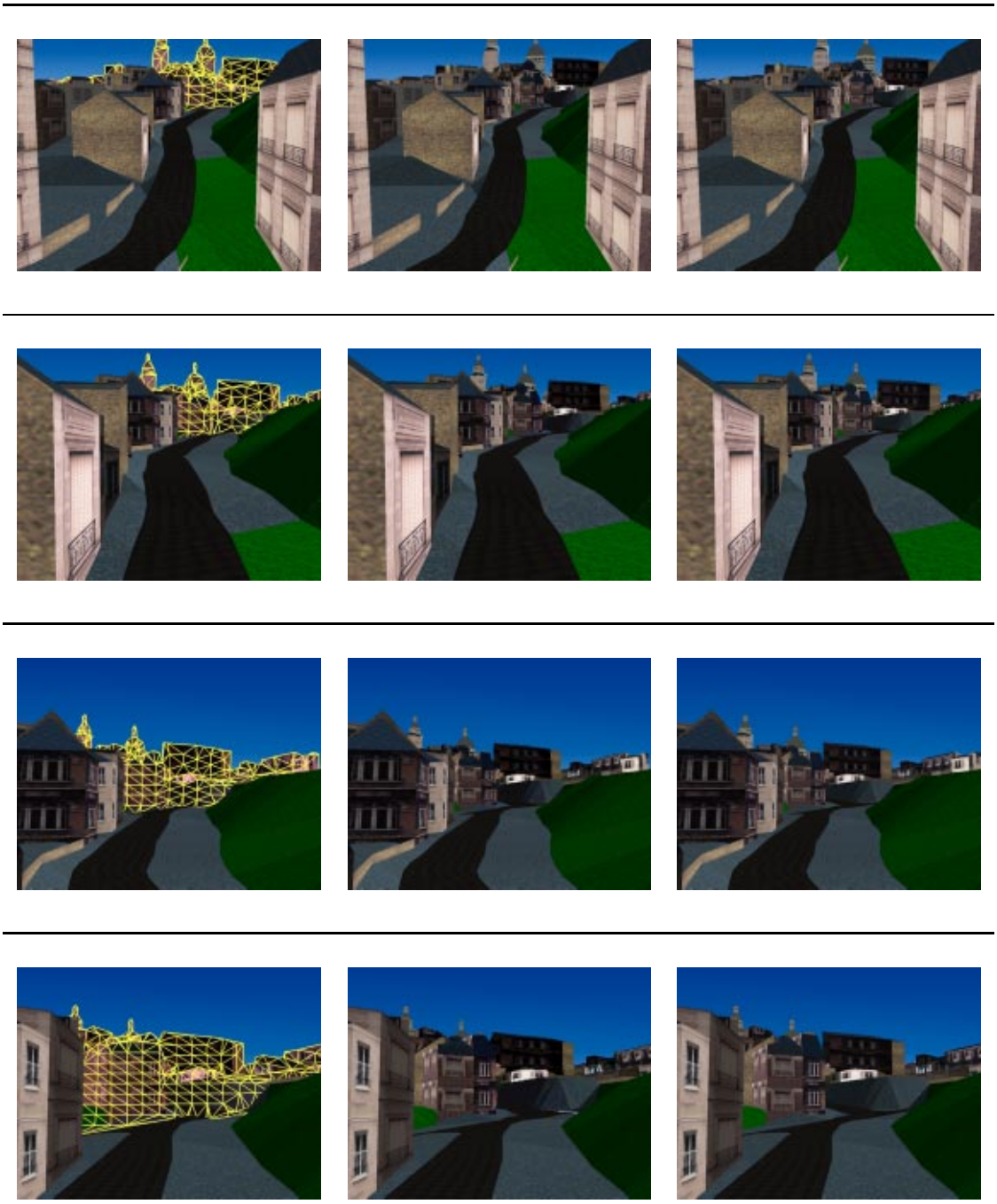
8.3 Visualization performance

The results of this implementation are quite satisfactory. The techniques achieve and average of 15 frames per second. The Paris model is composed of 140,000 polygons. All timings are computed on an SGI O_2 150 Mhz R10000 computer. The speed up achieved is very dependent on the complexity of the model used. A more complex model containing more detail in the facades, for example, would not affect the complexity of the impostors and thus may produce an even greater speed up.

In terms of space, a typical impostor piece, starting with a texture of 512x512, results in about 1,000 polygons. This is much less the than the visible distant model. The average size of the local model is approximately 4,000 polygons, therefore the total 3D geometry being drawn is reduced to 5,000 polygons.

The creation of the impostor takes about 4 seconds. All the operations needed to create the impostors can be thought of as a pre-process and accordingly stored with the model.

The model is composed of 1160 edges. Recomputing and storing all the impostors images clearly is significant and memory requirements can be excessive.



Local Model and Impostor (colored)	Local Model and Impostor	Complete Model

Figure 8-1: Images extracted from the Paris model.

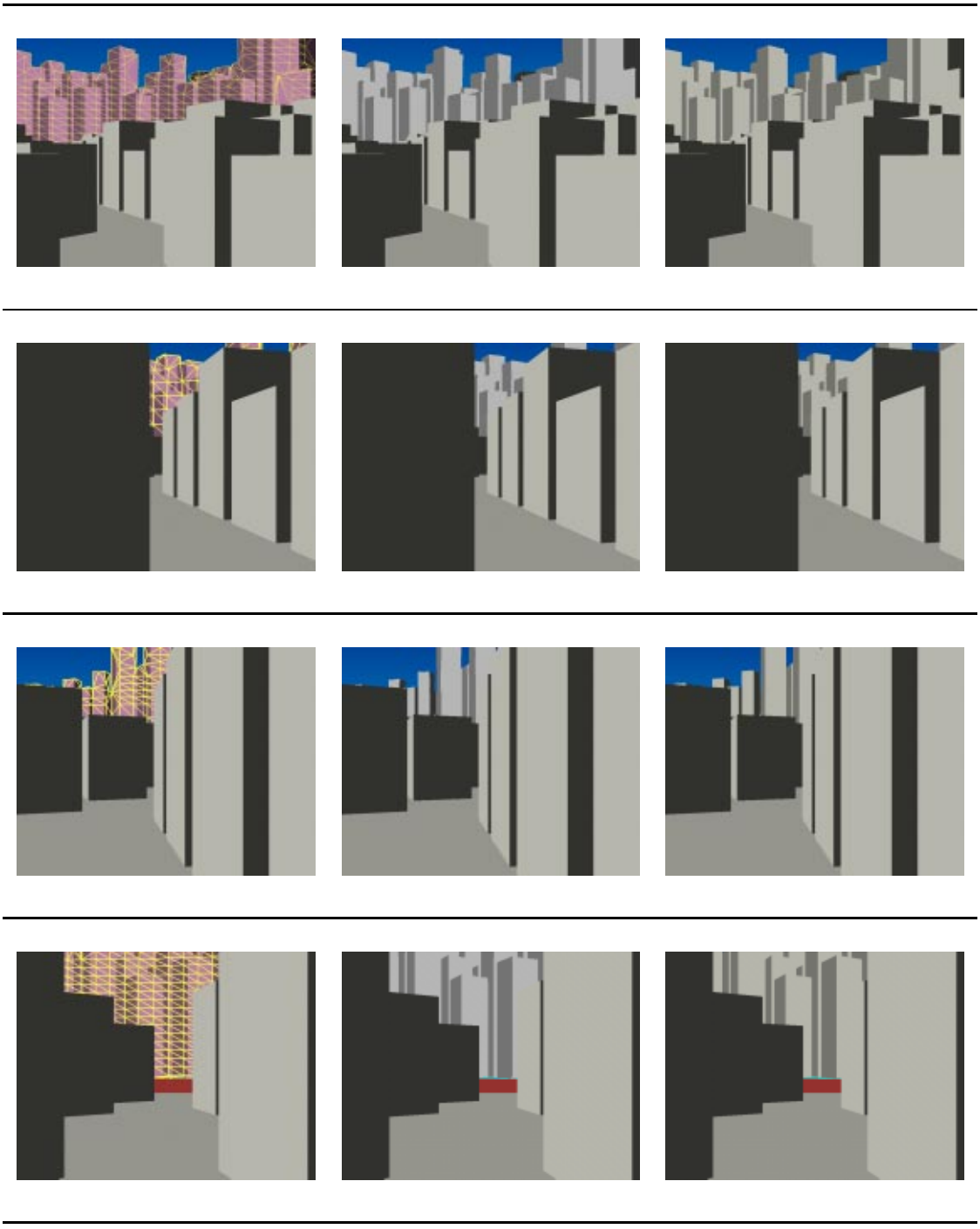
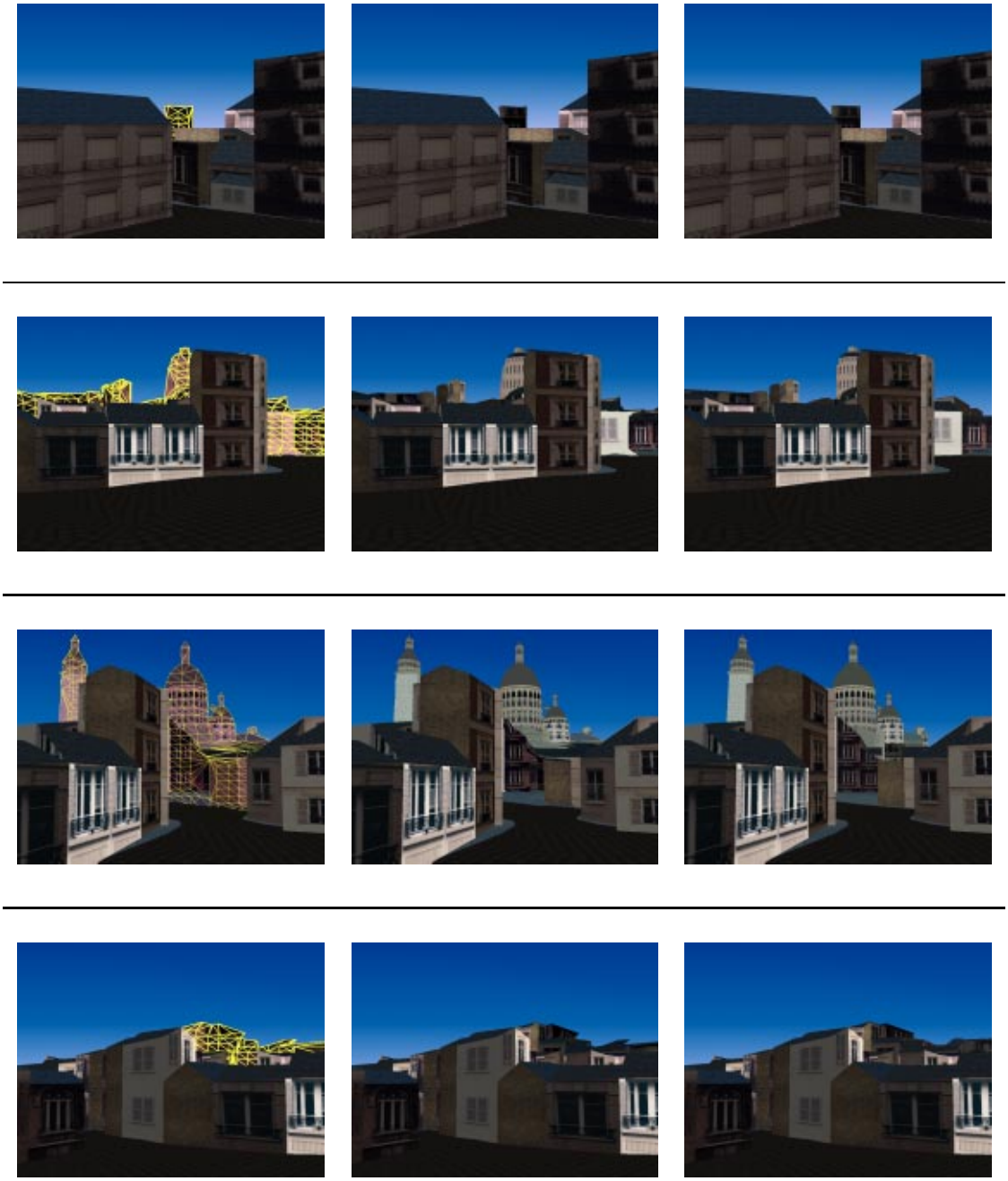
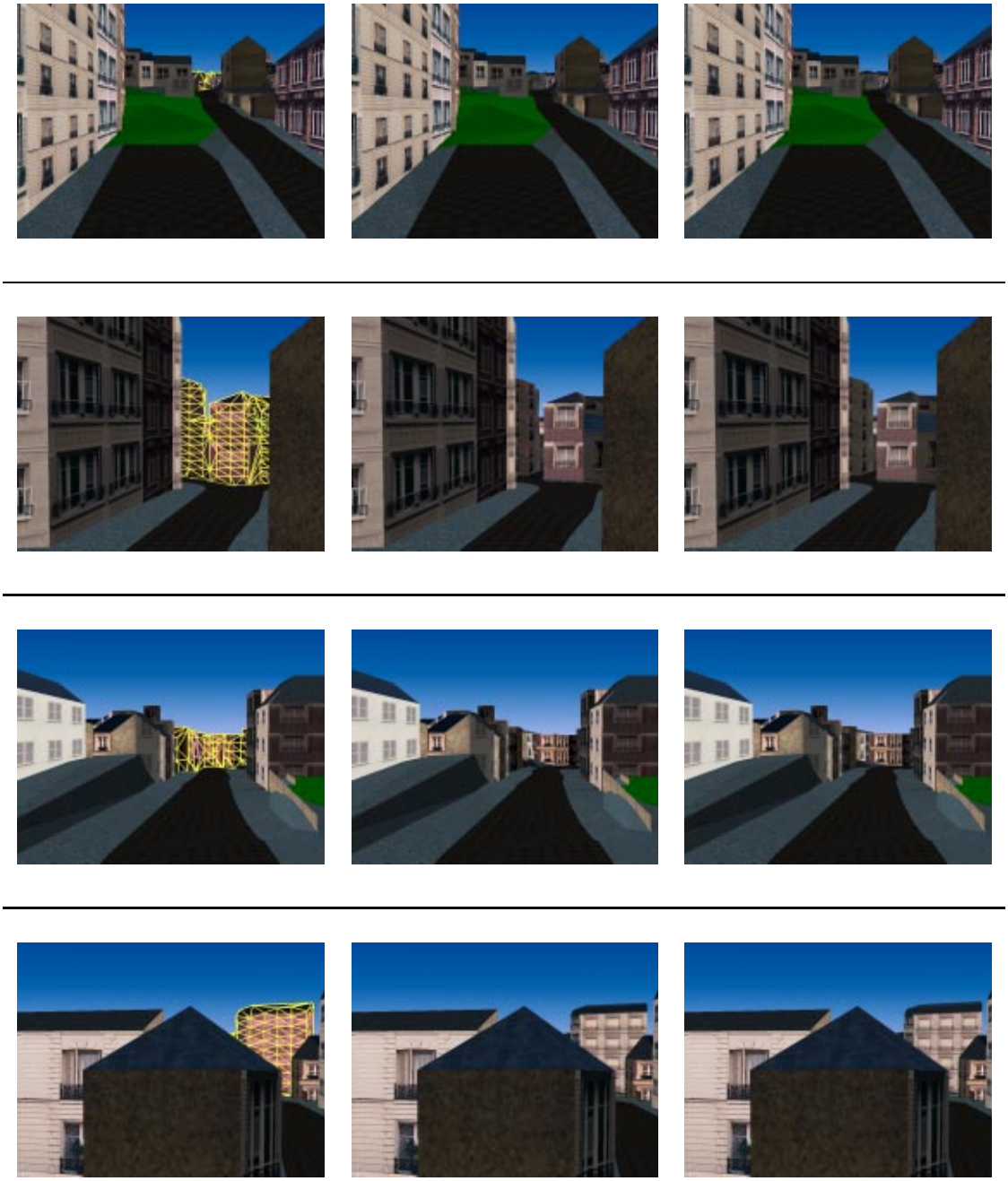


Figure 8-2: Images extracted from the generated model.



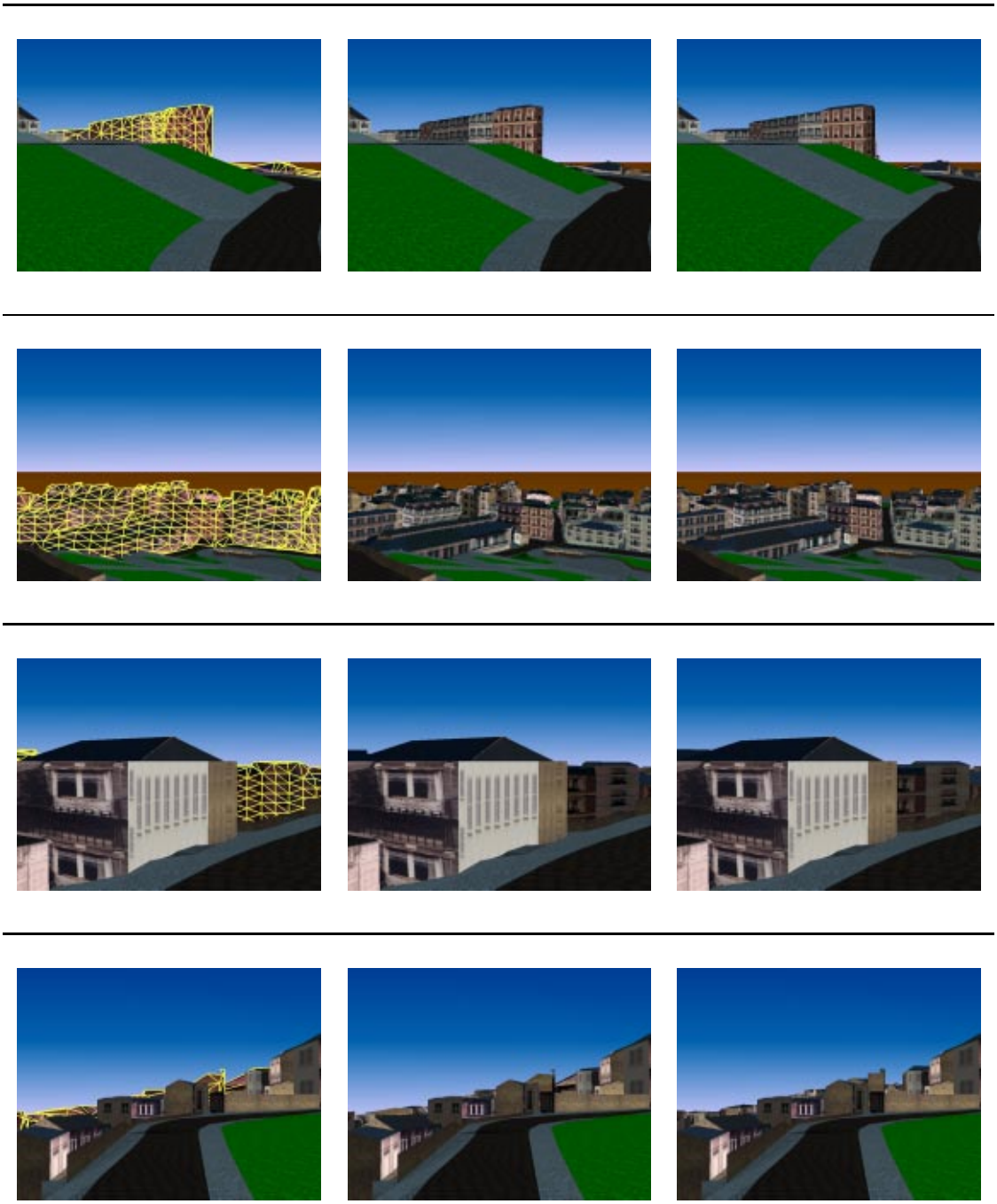
Local Model and Impostor (colored)	Local Model and Impostor	Complete Model

Figure 8-3: Images extracted from the Paris model at a square.



Local Model and Impostor (colored)	Local Model and Impostor	Complete Model

Figure 8-4: Images extracted from the Paris model at an intersection.



Local Model and Impostor (colored)	Local Model and Impostor	Complete Model

Figure 8-5: Images extracted from the Paris model at a city edge.

Chapter 9

Conclusion and Future Work

This thesis introduces a pipeline for the definition, segmentation and visualization of urban environments. The central idea is the development of a generalized model for the segmentation of the urban environment and the introduction of data structures that capture urban morphology. This results in morphological information usable during visualization. Algorithms for the identification of urban morphology are defined and a treatment using the morphology is used and proposed. The framework is based on Sillion *et al.*'s work [SDB97] of dividing the model into two distinct sub-sets at each frame: the local neighborhood and distant scenery. The local neighborhood is rendered using the full 3D model to provide detailed geometric information to the viewer. For the distant scenery, an "impostor" constructed from images is used. This impostor can then be used for a large number of frames.

9.1 New impostor types

In the current implementation impostor pieces are constructed using the approach developed by [SDB97]. These impostors, however, have problems associated with them. They were originally developed for an environment that assumes that streets are densely surrounded by buildings. Here, the visibility is blocked by the buildings along each side of the street. The impostor model they use works well for most streets. We have extended this implementation to deal with other urban conditions, namely

squares, city edges, and intersections. However, there are other conditions (discussed in section 7.3) that are not addressed. To solve these problems alternate impostor representations should be investigated.

A possible representation to investigate is a layered and hierarchical impostor, coupled with a notion of importance. A layered and hierarchical representation can be similar to the representation proposed by Shade *et al.*[SLS+96]. The work of Shade *et al.* dynamically and automatically creates view-dependent, image-based LOD models. They use a spatial hierarchy to divide the scene so that the LOD models represent regions of the scene. This allows them to properly depth-sort the LOD models for rendering. Using a similar hierarchy, based on our impostor representation, would probably provide some good results.

To use importance in the impostor representation, several LODs need to be used to render the far geometry. The LOD representation allows the fidelity of the geometry being viewed to change depending on the object's importance, and distance from the camera. LOD models could take the form of actual geometry, followed by an impostor, and finally by a textured quadrilateral. Each LOD is selected appropriately depending on the environment and what is being viewed.

9.2 Transitions

Transitions are another important issue needing to be addressed. In the current implementation, impostors are correct at the viewpoint from which they were generated. When the user changes streets or comes to the end of a street, a new impostor is constructed and used. Combination algorithms, which deal with transitions between impostors and new impostors constructed at other positions, could be a possible approach. Also, when the local model changes as the user moves from one street segment to another, parts of the previous impostor turn into 3D geometry. This transition needs to be addressed and made smoother. One approach is to 3D morph the parts of the impostor geometry that are going to turn into 3D geometry.

Another issue is when new impostor representations are introduced, how can they

smoothly transition from one representation to another. For example, when moving from a square to an intersection, what is the smoothest technique with which these two representations can transition.

9.3 Error analysis

An error measure needs to be used to determine the valid range for impostors. The impostor images contain 3D information and analyzing this 3D information would enable the system to compute the validity of the impostor for a new viewing position. To compute error, ones needs to look at the difference in appearance between the actual geometry and that of the impostor. If the difference is smaller than some user-specified threshold, the impostor representation is deemed acceptable. One point to consider is that this error metric must be fast to compute. A possible analysis technique would be to use angular disparity to measure the maximum difference between a point in the impostor and the same point in the real geometry.

9.4 Cracking

Finally, with the impostor representation, the problem of cracking is significantly reduced. It is not completely diminished. Further research needs to address the problem. Possible solutions include: sharing boundaries or vertices between the model and the impostor; and defining better heuristics that extract relevant data in the impostors.

Appendix A

Street File Format

It is an ASCII file with the following properties :

$$
\begin{array}{cccccc}
V_{x_1} & V_{y_1} & V_{z_1} & V_{x_2} & V_{y_2} & V_{z_2} \\
V_{x_1} & V_{y_1} & V_{z_1} & V_{x_2} & V_{y_2} & V_{z_2} \\
V_{x_1} & V_{y_1} & V_{z_1} & V_{x_2} & V_{y_2} & V_{z_2} \\
V_{x_1} & V_{y_1} & V_{z_1} & V_{x_2} & V_{y_2} & V_{z_2} \\
. & & & & & \\
. & & & & & \\
. & & & & &
\end{array}
$$

where (V_{x_1}, V_{y_1}, V_{z_1}) is the position of the start of a street segment and (V_{x_2}, V_{y_2}, V_{z_2}) is the end of a street segment. There are no restrictions on the length of the streets. It is assumed that a single path may be composed of several street segments.

Appendix B

CIT File Format

B.1 Basic syntax

CIT is a binary file with the following format. Support for ASCII will come later. Thus it has the disadvantage of platform-dependence; concerns about word lengths, byte-ordering, floating-point formats and so on.

The file consists simply of a series of records consisting of the following fields:

Separator	Datatype	NumberOfObjects	Data
(int)	(int)	(int)	(Datatype*NumberOfObjects)

The representation of data in this binary file can be thought of as being represented by a hierarchy of the fields defined above:

top-level-record (start)
    second-level-record (start)
        third-level-record (start)
            ...
        third-level-record (end)
        another-third-level-record (start)
            ...
        another-third-level-record (end)
    second-level-record (end)

another-second-level record (start)

...

another-second-level record (end)

top-level-record (end)

Any unrecognized separators in the records are simply ignored; this ensures that the format is extensible and that old parsers are still able to read newer versions of the format. Ignoring a tag consists of simply skipping unknown markers. The ordering of tags is generally unimportant, except as specifically noted below.

B.2 Fundamental data types

Separator This is has datatype *int* and represents what the data in the field represents. The values and corresponding fields are shown in Appendix B.4.

Datatype This is has datatype *int* and represents the data type of the data stored in the field. The values and their corresponding data types as shown in Table B.5.

NumberOfObjects This is has datatype *int* and represents the number of data elements stored in the field Data.

Data (Datatype) The Data of type "Datatype" is stored here. There are "NumberOfObjects" to read.

B.3 Overall file structure

The file is made up of a number of sections. More sections may be defined later, but those that are currently defined are File, Map, City, Block, Street, Triangle, Building, TerrainGeometry, BuildingGeometry, StreetGeometry, MaterialDefs, Material, EdgeData, Links, and LinkedObjects. A hierarchy exists as to where all these objects should be defined, but everything is defined in a File. Each object may contain other

pre-specified objects in addition to attributes. We will use braces "{" "}" to define the start and end of objects.

B.3.1 File

File
{
    FileVersion XXX
    City { ... }
    Map { ... }
    MaterialDefs { ... }
    Links { ... }
}

B.3.2 Map

Map
{
    Street { ... }
    Street { ... }
    Street { ... }
    ....
}

B.3.3 City

City
{
    Block { ... }
    Block { ... }

```
    Block { ... }
    ....
    ImpostorParams { ... }
    ImpostorParams { ... }
    ImpostorParams { ... }
    ....
    ImpostorLocalModel { ... }
    ImpostorLocalModel { ... }
    ImpostorLocalModel { ... }
    ....
}
```

B.3.4 Block

```
Block
{
    BlockID XXX
    EdgeData { ... }
    BuildingGeometry { ... }
    TerrainGeometry { ... }
}
```

B.3.5 Street

```
Street
{
    StreetStart XX XX XX
    StreetEnd XX XX XX
    StreetEdgeID XX
```

```
        StreetRightBlockID XX
        StreetLeftBlockID XX
        StreetLeftBlockID XX
        StreetImpostorParamsID XX
}
```

B.3.6 Triangle

```
Triangle
{
        TriangleMaterialID XX
        TriangleVertex XX XX XX
        TriangleNormal XX XX XX
        TriangleTexture XX XX
        TriangleVertex XX XX XX
        TriangleNormal XX XX XX
        TriangleTexture XX XX
        TriangleVertex XX XX XX
        TriangleNormal XX XX XX
        TriangleTexture XX XX
}
```

B.3.7 Building

```
Building
{
        BuildingID XX
        BuildingTriangles { ... }
}
```

B.3.8 TerrainGeometry

TerrainGeometry

{

Triangle { ... }

Triangle { ... }

...

}

B.3.9 BuildingGeometry

BuildingGeometry

{

Triangle { ... }

Triangle { ... }

...

}

B.3.10 StreetGeometry

StreetGeometry

{

Triangle { ... }

Triangle { ... }

...

}

B.3.11 BuildingTriangles

BuildingTriangles

{

Triangle { ... }

Triangle { ... }

...

}

B.3.12 MaterialDefs

MaterialDefs

{

Material { ... }

Material { ... }

...

}

B.3.13 Material

Material

{

MaterialID XX

MaterialTextureName XXXXXXXXXXX

MaterialAmbient XX XX XX

MaterialDiffuse XX XX XX

MaterialSpecular XX XX XX

}

B.3.14 EdgeData

```
EdgeData
{
        BlockEdgeID XX
        BlockEdgeID XX
        ...
}
```

B.3.15 Links

```
Links
{
    LinkedObjects { ... }
    LinkedObjects { ... }
    ...
}
```

B.3.16 LinkedObjects

```
LinkedObjects
{
            LinkedObjectID XX
            LinkedObjectLinkType XX
            LinkedObjectLinkName XXXXXXXX

            LinkedObjectBlockID XX
            LinkedObjectBlockID XX
            ...

            LinkedObjectEdgeID XX
            LinkedObjectEdgeID XX
```

```
        ...

        LinkedObjectBuildingID XX
        LinkedObjectBuildingID XX
        ...

        LinkedObjectLinkedObjectsID XX
        LinkedObjectLinkedObjectsID XX
        ...

}
```

B.3.17 ImpostorParams

```
ImpostorParams
{
        ImpostorParamsID XX
        ImpostorCamera { ... }
        ImpostorCamera { ... }
        ...
}
```

B.3.18 ImpostorCamera

```
ImpostorCamera
{
        ImpostorCameraName XXXXXX
        ImpostorCameraPosition XX XX XX
        ImpostorCameraViewingDirection XX XX XX
        ImpostorCameraUpDirection XX XX XX
        ImpostorCameraElevation XX
```

ImpostorCameraNearDistance XX

ImpostorCameraFarDistance XX

ImpostorCameraAspectRatio XX

ImpostorCameraWidthAngle XX

ImpostorCameraHeightAngle XX

}

B.3.19 ImpostorLocalModel

ImpostorLocalModel

{

ImpostorLocalModelID XX

ImpostorLocalModelBlockID XX

ImpostorLocalModelEdgeID XX

...

}

B.4 Separators

Table B.1 and table B.2 are a quick reference to all the fields and where they are in the heiarchy.

Tag Name	Value	Data Type	Size	Parent	Compliment
StartOfFile	0	None	0	None	EndOfFile
EndOfFile	1	None	0	None	StartOfFile
StartOfMap	2	None	0	File	EndOfMap
EndOfMap	3	None	0	File	StartOfMap
StartOfStreet	4	None	0	Map	EndOfStreet
EndOfStreet	5	None	0	Map	StartOfStreet
StartOfCity	6	None	0	File	EndOfCity
EndOfCity	7	None	0	None	StartOfCity
StartOfBlock	8	None	0	City	EndOfBlock
EndOfBlock	9	None	0	City	StartOfBlock
StartOfTriangle	10	None	0	TerrainGeometry BuildingGeometry StreetGeometry BuildingTriangles	EndOfTriangle
EndOfTriangle	11	None	0	TerrainGeometry BuildingGeometry StreetGeometry BuildingTriangles	StartOfTriangle
StartOfBuilding	12	None	0	BuildingGeometry	EndOfBuilding
EndOfBuilding	13	None	0	BuildingGeometry	StartOfBuilding
StartOfTerrainGeometry	14	None	0	Block	EndOfTerrainGeometry
EndOfTerrainGeometry	15	None	0	Block	StartOfTerrainGeometry
StartOfBuildingGeometry	16	None	0	Block	EndOfBuildingGeometry
EndOfBuildingGeometry	17	None	0	Block	StartOfBuildingGeometry
StartOfStreetGeometry	18	None	0	Street	EndOfStreetGeometry
EndOfStreetGeometry	19	None	0	Street	StartOfStreetGeometry
StartOfMaterial	20	None	0	MaterialDefs	EndOfMaterial
EndOfMaterial	21	None	0	MaterialDefs	StartOfMaterial
StartOfMaterialDefs	22	None	0	File	EndOfMaterialDefs
EndOfMaterialDefs	23	None	0	File	StartOfMaterialDefs
StartOfEdgeData	24	None	0	Block	EndOfEdgeData
EndOfEdgeData	25	None	0	Block	StartOfEdgeData

Table B.1: File separators

B.5 Data type field values

Table B.3 shows the values used for each data type in the file format.

Tag Name	Value	Data Type	Size	Parent	Compliment
FileVersion	26	float	1	File	None
StreetStart	27	double	3	Street	None
StreetEnd	28	double	3	Street	None
StreetEdgeID	29	int	1	Street	None
StreetLeftBlockID	30	int	1	Street	None
StreetRightBlockID	31	int	1	Street	None
StreetType	32	int	1	Street	None
TriangleVertex	33	double	3	Triangle	None
TriangleTexture	34	double	2	Triangle	None
TriangleNormal	35	double	3	Triangle	None
TriangleMaterialID	36	int	1	Triangle	None
BlockID	37	int	1	Block	None
BlockEdgeID	38	int	1	Block	None
MaterialID	39	int	1	Material	None
MaterialTextureName	40	char	variable	Material	None
MaterialAmbient	41	float	3	Material	None
MaterialDiffuse	42	float	3	Material	None
MaterialSpecular	43	float	3	Material	None
StartOfLinks	44	None	0	File	EndOfLinks
EndOfLinks	45	None	0	File	StartOfLinks
StartOfLinkedObject	46	None	0	Links	None
EndOfLinkedObject	47	None	0	Links	None
LinkedObjectID	48	int	1	LinkedObject	None
LinkedObjectEdgeID	49	int	1	LinkedObject	None
LinkedObjectBlockID	50	int	1	LinkedObject	None
LinkedObjectLinkType	51	int	1	LinkedObject	None
LinkedObjectLinkName	52	char	variable	LinkedObject	None
StreetImpostorLocalModeID	53	int	1	Street	None
StreetImpostorParamsID	54	int	1	Street	None
StartOfImpostorParams	55	None	0	City	EndOfImpostorParams
EndOfImpostorParams	56	None	0	City	StartOfImpostorParams
StartOfImpostorLocalModel	57	None	0	City	EndOfImpostorLocalModel
EndOfImpostorLocalModel	58	None	0	City	StartOfImpostorLocalModel
StartOfImpostorCamera	59	None	0	ImpostorParams	EndOfImpostorCamera
EndOfImpostorCamera	60	None	0	ImpostorParams	StartOfImpostorCamera
ImpostorLocalModelID	61	int	1	ImpostorLocalModel	None
ImpostorLocalModelBlockID	62	int	1	ImpostorLocalModel	None
ImpostorLocalModelEdgeID	63	int	1	ImpostorLocalModel	None
ImpostorParamsID	64	int	1	ImpostorParams	None
ImpostorCameraPosition	65	double	3	ImpostorCamera	None
ImpostorCameraViewingDirection	66	double	3	ImpostorCamera	None
ImpostorCameraUpDirection	67	double	3	ImpostorCamera	None
ImpostorCameraElevation	68	float	1	ImpostorCamera	None
ImpostorCameraNearDistance	69	float	1	ImpostorCamera	None
ImpostorCameraFarDistance	70	float	1	ImpostorCamera	None
ImpostorCameraAspectRatio	71	float	1	ImpostorCamera	None
ImpostorCameraWidthAngle	72	float	1	ImpostorCamera	None
ImpostorCameraHeightAngle	73	float	1	ImpostorCamera	None
BuildingID	74	int	1	Building	None
StartOfBuildingTriangles	75	None	0	Building	EndOfBuildingTriangles
EndOfBuildingTriangles	76	None	0	Building	StartOfBuildingTriangles
LinkedObjectBuildingID	77	int	1	LinkedObject	None
LinkedObjectLinkedObjectsID	78	int	1	LinkedObject	None
ImpostorCameraName	79	char	variable	ImpostorCamera	None

Table B.2: File separators

Separator	Value
floatType	0
doubleType	1
intType	2
longType	3
charType	4
noneType	5

Table B.3: Data type field values

B.6 Sample code

B.6.1 Reading data

```
int ReadObject (int *pType,
                int *pDataType,
                int *pNumberOfObjects,
                void **ppData)
{
  if (!i_pfFilePointer)
    return TRUE;

  if (feof (i_pfFilePointer))
    return TRUE;

  if (!fread (pType, sizeof(int), 1, i_pfFilePointer))
    return TRUE;
  if (!fread (pDataType, sizeof (int), 1, i_pfFilePointer))
    return TRUE;
  if (!fread (pNumberOfObjects, sizeof(int), 1, i_pfFilePointer))
    return TRUE;

#ifdef _M_IX86 /* PC SPECIFIC BYTE CONVERSION */
  ConvertByteOrdering (1, DataType::intType,
                       pType, pType);
  ConvertByteOrdering (1, DataType::intType,
                       pDataType, pDataType);
  ConvertByteOrdering (1, DataType::intType,
                       pNumberOfObjects, pNumberOfObjects);
#endif
```

```
  if (*pNumberOfObjects == 0)
    return FALSE;

  (*ppData) = (void *) malloc (DataType::GetSize (pDataType)
                               *(*pNumberOfObjects));
  assert (*ppData);

  if (!fread ((*ppData), DataType::GetSize (pDataType),
              (*pNumberOfObjects), i_pfFilePointer))
    return TRUE;

#ifdef _M_IX86 /* PC SPECIFIC BYTE CONVERSION */
  ConvertByteOrdering ((*pNumberOfObjects), (*pDataType),
                       (*ppData), (*ppData));
#endif

  return FALSE;
}

/* PC byte ordering code */
Bool
File:: ConvertByteOrdering (int Size,
                            int dataType,
                            void *fromData,
                            void *toData)
{
  for (int CurrentByte=0; CurrentByte < Size; CurrentByte++) {
    switch (dataType) {
    case DataType::floatType: {
    union FloatConvert {
        float i;
        char c[4];
      } fromConvert, toConvert;
      fromConvert.i = ((float *)fromData)[CurrentByte];
      for (int i=0; i < 4; i++)
        toConvert.c[i] = fromConvert.c[3-i];
      ((float *)toData)[CurrentByte] = toConvert.i;
      break;
    }
    case DataType::doubleType: {
      union DoubleConvert {
        double i;
        char c[8];
      } fromConvert, toConvert;
      fromConvert.i = ((double *)fromData)[CurrentByte];
```

```
      for (int i=0; i < 4; i++)
        toConvert.c[i+4] = fromConvert.c[3-i];
      for (i=4; i < 8; i++)
        toConvert.c[i-4] = fromConvert.c[7-i+4];
      ((double *) toData)[CurrentByte] = toConvert.i;
      break;
    }
    case DataType::intType: {
      union FloatConvert {
        int i;
        char c[4];
      } fromConvert, toConvert;
      fromConvert.i = ((int *)fromData)[CurrentByte];
      for (int i=0; i < 4; i++)
        toConvert.c[i] = fromConvert.c[3-i];
      ((int *)toData)[CurrentByte] = toConvert.i;
      break;
    }
    case DataType::longType: {
      union LongConvert {
        long i;
        char c[4];
      } fromConvert, toConvert;
      fromConvert.i = ((long *)fromData)[CurrentByte];
      for (int i=0; i < 4; i++)
        toConvert.c[i] = fromConvert.c[3-i];
      ((long *)toData)[CurrentByte] = toConvert.i;
      break;
    }
    case DataType::charType:
      ((char *)toData)[CurrentByte] = ((char *)fromData)[CurrentByte];
      break;
    case DataType::noneType:
    default:
      Warning ("Unknown Data Type, skipping");
      break;
    }
  }
  return FALSE;
}
```

B.6.2 Writing data

```
int
File:: WriteSeperator (int Seperator)
{
  fwrite (&Seperator, sizeof (int), 1, i_pfFilePointer);
  int dummy = DataType:: noneType;
  fwrite (&dummy,  sizeof (int), 1, i_pfFilePointer);
  dummy = 0;
  fwrite (&dummy, sizeof(int), 1, i_pfFilePointer);
  return FALSE;
}


int WriteData (int DataSeperator,
               int Size,
               int dataType,
               void *pData)
{
  fwrite (&DataSeperator, sizeof (int), 1, i_pfFilePointer);
  fwrite (&dataType,  sizeof (int), 1, i_pfFilePointer);
  fwrite (&Size,  sizeof (int), 1, i_pfFilePointer);

  switch (dataType) {
  case DataType::floatType:
    fwrite ((float *) pData, DataType::GetSize (dataType),
             Size, i_pfFilePointer);
    break;
  case DataType::doubleType:
    fwrite ((double *) pData, DataType::GetSize (dataType),
            Size, i_pfFilePointer);
    break;
  case DataType::intType:
    fwrite ((int *) pData, DataType::GetSize (dataType),
            Size, i_pfFilePointer);
    break;
  case DataType::longType:
    fwrite ((long *) pData, DataType::GetSize (dataType),
            Size, i_pfFilePointer);
    break;
  case DataType::charType:
    fwrite ((char *) pData, DataType::GetSize (dataType),
            Size, i_pfFilePointer);
```

```
    break;
  case DataType::noneType:
  default:
    Warning ("Unknown Data Type, skipping");
    break;
  }

  return FALSE;
}
```

Bibliography

[Ale77] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, NY, 1977.

[Ale87] Christopher Alexander. *A New Theory of Urban Design*. Oxford University Press, New York, NY, 1987.

[ARB90] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):41–50, March 1990.

[Bau74] Bruce G. Baumgart. Geometric modeling for computer vision. AIM-249, STA -CS-74-463, CS Dept, Stanford U., October 1974.

[Bla87] K. Blanton. A new approach for flight simulator visual systems. In *Simulators IV, Proceedings of the SCCS Simulators Conference*, pages 229–233, 1987.

[Bur71] B C Burnaby. *Urban Structure*. The Planning Department of the District of Burnaby B.C. Canada, B.C., Canada, 1971.

[CDL+96] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Proceedings of Graphics Interface '96*, pages 365–371, 1996.

[Cla76] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[CW93] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288, August 1993.

[DP92] L. De Floriani and E. Puppo. An on-line algorithm for constrained delauney triangulations. *CVGIP: Graphical Models and Image Processing*, 54(4):290–300, July 1992.

[DTM96] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 11–20. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Eas82] C. M. Eastman. Introducation to computer aided design. In *Course Notes*. Carnegie-Mellon University, Pittsburg, PA, 1982.

[FS93] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254, August 1993.

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 43–54. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[GK93] Ned Greene and M. Kass. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[GR91a] T. Gonzalez and M. Razzazi. Properties and algorithms for constrained Delaunay triangulations. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 114–117, 1991.

[GR91b] F. Goodchild and David W. Rhind, editors. *Geographical Information Systems: Principles and Applications.* John Wiley & Sons, Inc., New York, NY, 1991.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[JLF95] William Jepson, Robin Liggett, and Scott Friedman. An environment for real-time urban simulation. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 165–166. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[Jon71] C. B. Jones. A new approach to the 'hidden line' problem. *Computer Journal*, 14(3):232–237, August 1971.

[Jun88] Dz-Mou Jung. An optimal algorithm for constrained Delaunay triangulation. In *Proc. 26th Allerton Conf. Commun. Control Comput.*, pages 85–86, Urbana, IL, 1988. Univ. Illinois.

[Kaj85] James T. Kajiya. Anisotropic reflection models. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 15–21, July 1985.

[KM92] T. C. Kao and D. M. Mount. Incremental construction and dynamic maintenance of constrained Delaunay triangulations. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 170–175, 1992.

[Kos91] Spiro Kostof. *The City Shaped: Urban Patterns and the Meanings Through History.* Little, Brown and Co., Boston, MA, 1991.

[LG95] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 105–106. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Lyn60] Kevin Lynch. *The Image of the City*. MIT Press, Cambridge, MA, 1960.

[Lyn96] Kevin Lynch. *Good City Form*. MIT Press, Cambridge, MA, tenth edition, 1996.

[MB95] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 39–46. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[MS95] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 95–102. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[Mum61] Lewis Mumford. *The City in History*. Harcourt: Brace and World, New York, NY, 1961.

[RB93] J. Rossignac and P. Borrel. Multi-resolution 3D approximation for rendering complex scenes. In *Second Conference on Geometric Modelling in Computer Graphics*, pages 453–465, June 1993. Genova, Italy.

[RP94] Matthew Regan and Ronald Pose. Priority rendering with a virtual reality address recalculation pipeline. In Andrew Glassner, editor, *Proceed-*

ings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994), Computer Graphics Proceedings, Annual Conference Series, pages 155–162. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[SBJ97] Michael Southworth and Eran Ben-Joseph. *Streets and the Shaping of Towns and Cities*. McGraw Hill, New York, NY, 1997.

[SD96] Steven M. Seitz and Charles R. Dyer. View morphing: Synthesizing 3D metamorphoses using image transforms. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 21–30. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[SDB97] François Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum (Proc. of Eurographics '97)*, 16(3):207–218, September 1997.

[Sei88] R. Seidel. Constrained Delaunay triangulations and Voronoi diagrams. In *Report 260*, pages 178–191. IIG-TU, Graz, Austria, 1988.

[Slo91] S. W. Sloan. A fast algorithm for generating constrained Delaunay triangulations. Research Report 065.07.1991, The University of Newcastle Department of Civil Engineering and Surveying, New South Wales, 1991.

[SLS+96] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 75–82. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[SS96] Gernot Schaufler and Wolfgang Sturzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–235, August 1996.

[TS91] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 61–69, July 1991.

[Wei86] Kevin J. Weiler. *Topological structures for geometric modeling.* Ph.d. thesis, Rensselaer Polytechnic Institute, August 1986.

[Wer94] Josie Wernecke. *The Inventor Mentor.* Addison-Wesley, Reading, MA, 1994.

[WT92] C. A. Wang and Y. H. Tsin. Efficiently updating constrained Delaunay triangulations. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 176–181, 1992.