I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1546

REMOTE INTERACTIVE WALKTHROUGH OF CITY MODELS USING PROCEDURAL GEOMETRY

JEAN-EUDES MARVIE, JULIEN PERRET,
KADI BOUATOUCH

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Remote Interactive Walkthrough of City Models Using Procedural Geometry

Jean-Eudes Marvie, Julien Perret,
Kadi Bouatouch

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet SIAMES

Publication interne n ° 1546 — July 2003 — 32 pages

Abstract: This report presents a new navigation system built upon our client-server framework named *Magellan*. With this system one can navigate through a city model represented with procedural models transmitted to clients over a network. The geometry of these models is generated on the fly and in real time at the client side. These procedural models are described using an enhanced and open version of the L-system language we have developed. The navigation system relies on different kinds of preprocessing such as space subdivision, visibility computation as well as a method for computing some parameters used to efficiently select the appropriate level of detail of objects. The two last preprocessings are automatically performed by the graphics card connected to the used computer. We also show how to build and compress the different files used to represent the database (corresponding to a city model) once this preprocessing has been effected. Methods for prefetching, anticipating and caching during remote navigation are also presented. Finally, we show how the procedural models, the space subdivision and the visibility information data are encoded into VRML97 files.

Key-words: Walkthrough, City, Network, Streaming, L-system, Level Of Details, Visibility, Framework, Magellan

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Utilisation de Géométrie Procédurale pour la Navigation Interactive dans des Modèles de Ville Distants

Résumé : Dans ce rapport nous présentons un nouveau système de navigation basé sur notre plate-forme de développement *Magellan*. Ce système permet la navigation au sein de modèles de villes représentés à l'aide de modèles procéduraux, transférés à des clients via un réseau bas débit. La géométrie de ces modèles est générée à la volée et en temps réel sur la machine cliente. Ces modèles procéduraux sont décrits à l'aide d'une version étendue du language L-system que nous avons développés. Le système de navigation repose sur différents types de pré-calculs tels que la subdivision spatiale, le calcul de visibilité ainsi que le pré-calcul d'une métrique pour la sélection des niveaux de détails. Ces deux dernier traitements sont effectués à l'aide de matériel d'accélération 3D. Nous montrons aussi comment construire et compresser les différents fichiers utilisés pour représenter la base de donnée correspondant à un modèle de ville un fois les pré-traitements effectués. Des méthodes de pré-chargement, d'anticipation et de gestion de cache pendant la navigation distante sont aussi présentées. Finallement, nous expliquons comment les données associées aux models procéduraux, à la subdivision spatiale et à la relation de visibilité sont encodées dans notre extension du format VRML97.

Mots clés : Navigation, Ville, Réseau, Streaming, L-system, Niveaux de Détails, Visibilité, Plate-forme, Magellan

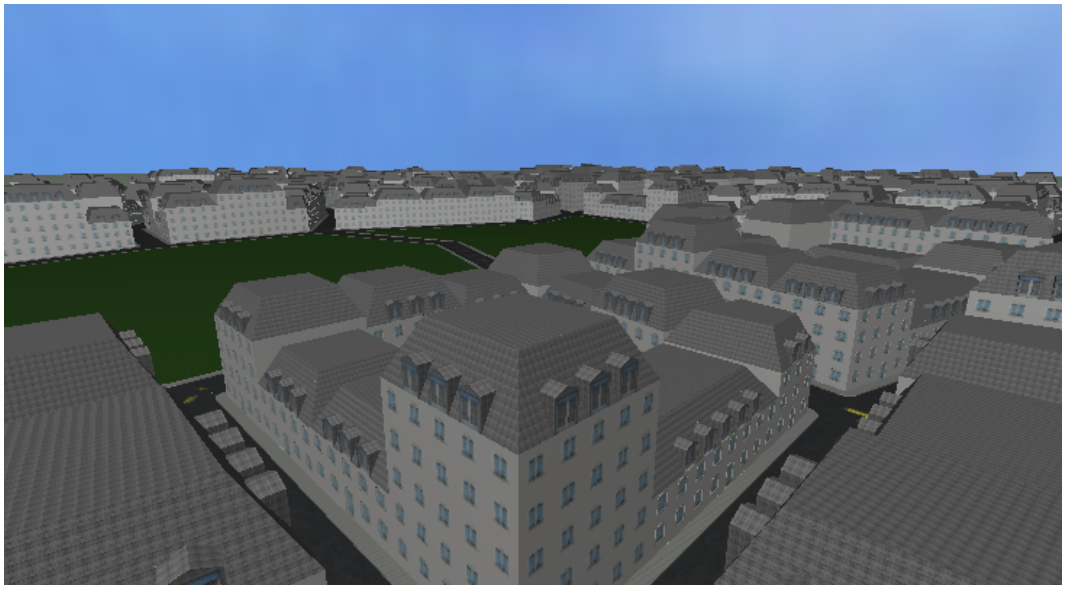

Figure 1: Bird's eye view of a generated city model.

1 Introduction and Related work

Transmission and real-time visualization of massive 3D models are constrained by the networks bandwidth and the graphics hardware performances. These constraints have led to two research directions that are progressive 3D models transmission over Internet or local area network and real-time rendering of massive 3D models.

With regard to progressive 3D models transmission, Schmalstieg and Gervautz [28] suggest the use of geometric levels of detail (LODs). In their method, as soon as one LOD is selected according to its distance from the viewpoint, the finer LOD is prefetched over the network. In the same spirit, Schneider and Martin [30] developed an adaptive framework that selects the LOD of 3D objects to transmit based on the available bandwidth, the client's computational power and its graphics capabilities. Teler and Lischinski [31] also cope with the same problem by employing a client-server approach to transmit only parts of a large model to the client at an appropriate LOD, depending on the viewpoint's position and direction, the available transmission bandwidth and the expected improvement in image quality that is achieved by transmitting an object. Hoppe [16] makes use of a progressive mesh (PM) data structure allowing smooth transitions between successive LODs by adding a variable number of new vertices to the current visualized model. This method avoids data redundancy between different levels. Moreover, Hoppe asserts that this data structure is adapted

to a progressive transmission of the geometry over the network. In [25] Rusinkiewicz and Levoy show how to incorporate view-dependent progressive transmission in their QSplat [24] system. Unlike LODs and PM methods, QSplat relies on point-based sampling. QSplat is a multi-resolution representation based on a hierarchy of bounding spheres that allows to render large 3D models interactively. The interest in all these methods is that levels of detail are also used to accelerate the rendering process. To reduce the amount of data to be transmitted over the network, Schmalstieg and Gervautz [29] propose the use of procedural models. In this case, the models geometry is reconstructed on the client side and the data transmitted over the network are procedural model parameters.

As for real time rendering of massive 3D models on a single computer, one can find many solutions in the literature. The most commonly used solution consists in subdividing the scene into cells and computing a potentially visible set (PVS) of objects for each view cell. During walkthrough, only the PVS of the cell containing the current viewpoint is used for rendering. Conservative visibility computation has been widely addressed in many previous publications. Many systems for interactive building walkthrough [1, 32, 13, 12, 20, 26] make use of this approach. For these systems the view cells are, most of time, the rooms of the buildings. City models can also be visualized using such a method [8, 33] where view cells are extruded road footprints. For a more general type of complex scenes, Schaufler et al. [27] propose a solution using occluder fusion and Durand et al. [11] propose the use of extended projections. However, in some cases certain PVSs may be too large to achieve interactivity. For this reason, Decoret et al [10] and Aliaga [2] substitute faraway geometry with textured depth meshes (TDM) impostors. In [3] Andújar et al. distinguish fully visible set from hardly visible sets (HVS) of objects. Using their HVS classification, they select the appropriate LOD for each hardly visible object.

Martin [19] has classified into three major categories the methods for rendering 3D models in client-server environments. The first category is called *client-side methods* [15]. The methods of this category do not involve any rendering on the part of the server. The geometry is downloaded to each client that requests it and the client is responsible for rendering it. Such methods are well suited for applications needing real time interaction. When using the methods of the second category, called *server-side methods*, the 3D model is fully rendered on the server side and the resulting images are sent to the clients [9, 5]. In [9], the sever generates the frames, encodes and transmit them to the client. The encoded frames are transmitted as a video stream to the client which decodes the stream and displays it. In [5], each client uses previous views of the scene to predict next view using image-based rendering techniques. The server performs the same prediction and sends only the difference between the predicted and the actual views. Compressed difference images require less bandwidth than the compressed images of each frame. Such methods get interesting when the clients have limited resources and limited graphics performances. As for the third category, called *hybrid-side methods*[18, 17], parts of the 3D model are rendered on the server and the other parts are downloaded and rendered on the client side. Such methods have the advantage of reducing the geometric complexity of the data to be transmitted by replacing parts of the geometry with images. However, deciding which part of a model should be

rendered on the server or on the client is not a trivial task. One possibility is that the server could render high and low resolution versions of a 3D model and send the residual error image and the low-resolution geometry to the client [18, 17]. In this case, the role of the client is to render the coarse model and to add the residual image to restore a full quality rendering.

To sum up, our system falls into the first category and we think that networked walkthrough of massive models such as the city presented in Figure 1 must take advantage of the two approaches described above, say the use of progressive models and the subdivision of the scene into 3D cells. For example a PVS could be composed of LODs, PM or QSplats. Furthermore, we think that using procedural models for network transmission is really more interesting than transmitting models geometry even in a progressive way.

Taking all these points into account we propose to combine visibility calculation with LODs generated by procedural models. The choice of procedural models is very interesting because city models contain several pattern based geometric objects easy to generate, such as buildings, roads and trees. Finally, LODs are also very easy and fast to generate for these models. We also present a new method that makes use of the visibility computation results to select the appropriate LOD for each object during the rendering process.

2 Overview

We present a system which allows real-time walkthrough of 3D city models located on a remote machine and transmitted over a low bandwidth network, using TCP/IP protocol. The global architecture of our system is illustrated by Figure 2. The server provides access to several city models, each one being represented by one database. Each database is a set of VRML97 [6] files describing the 3D city model. The database structure is detailed in section 3. Each remote client machine can connect to the server to walk through a city model using its associated database. There is right now no interaction between clients, and each client renders its own representation of the city model.

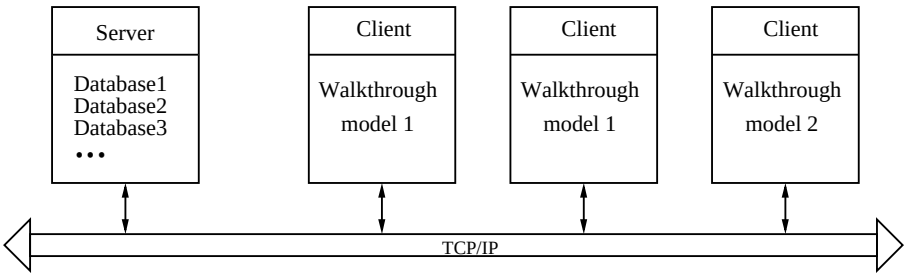

Figure 2: Global system architecture.

One of the main mechanism in our system is to transmit and visualize only the subset of the model that is potentially visible from the current viewpoint. With this aim in view, the scene is subdivided into cells and a PVS of objects is computed for each cell. In addition, we determine the adjacency relationship between cells. During walkthrough the cells adjacent to the current one, as well as their PVSs are prefetched over the network. A more pertinent prefetching technique is to determine the next visited adjacent cell using motion prediction. In this way, the database is progressively transmitted to the client. Furthermore the geometry used for the rendering process is the PVS of the visited cell. This PVS is frustum culled using the bounding box of each of its objects, and the obtained result is sent to the OpenGL hardware renderer. In addition, several acceleration techniques has been implemented in our system as explained hereafter.

Recall that network transmissions are distributed over navigation time using space subdivision driven data prefetching. In other words, each transmission concerns only one PVS. However the transmission of some PVSs may still require too much network bandwidth, which causes a latency time that cannot be compensated for by prefetching. To cope with this problem, we use procedural models to avoid geometry transmission. More precisely, the server database contains a library of procedural models as well as some sets of input parameters. Each of these sets is used by one procedural model to generate the geometry of one object. Whenever a client receives a set of input parameters related to a procedural model, it generates the associated geometry. To generate different geometric objects corresponding to the same procedural model, the client just needs to download the procedural model as well as the sets of input parameters for these geometric objects. One procedural model is then capable of generating several different geometric models of any size using a few number of input parameters, which reduces the amount of data transmitted over the network. Our procedural models are coded with a new L-system based VRML97 scripting language. When the client receives a set of input parameters it asks for the associated procedural model using external prototyping mechanism provided by VRML97. This mechanism allows to manipulate each procedural model as a VRML97 built-in node. Each received set of input parameters is used to instantiate a node of its associated prototype.

With regard to the rendering process running on the client side, it faces the same problems than those raised by network transmissions. Indeed, a PVS might still contain too much polygons to get an interactive frame rate (say, 25 frames per seconds). In order to reduce the amount of polygons to be rendered, the geometry of the PVS's objects is represented with LODs. LODs are generated by procedural models. That is to say, one procedural model computes all the LODs for the objects it generates. Another interest in procedural geometry generation is the possibility of producing any number of LODs. Usually, the suitable LOD is selected using an Euclidean metric giving the distance between the viewpoint and the object center. In our implementation, we propose a different method which takes advantage of the visibility computation results. While computing one cell visibility the system computes an ACH (average coverage hint) for each visible object. The ACH of an object represents the average surface area of this object when projected onto the projection plane of any viewpoint within the cell. During rendering, the ACH of each object is exploited as follows. For a

purpose of interactivity, the maximum total number of polygons to be rendered is set to a given number N_{max}. The ACH of each object represents the proportion of the N_{max} polygons allowed to be rendered. Thus, to each ACH corresponds a certain number of polygons N_{poly}. In this way, the N_{poly} associated with an object represented by LODs allows to select the LOD composed of a number of polygons closest to N_{poly}. In addition to the above optimization we introduce another VRML97 node named OneSideGroup. This node operates back face culling on a set of "flat and cooplanar" geometric objects rather than on polygons. As an example, this kind of node can encapsulate the set of objects making up the frontage of a building. All the objects of a frontage are supported by a same plane whose normal can be used to cull them in one go. As it will be explained later, visibility computation is performed on the entire buildings. More precisely, a building is either completely visible or not. Therefore when a building is visible, all its frontages are considered as visible. The *OneSideGroup* mechanism is used to cull back frontages during rendering.

The rest of this report is organized as follows. In the next section we describe the databases generation including from scratch city model generation, visibility computation and VRML97 database representation. Next, we give a short description of our L-system based scripting language we use for procedural models description and we explain how we construct the procedural models using this language. In a third section we describe our networked real-time rendering implementation developed using Magellan, our OpenGL based development framework. Then we discuss our results before concluding.

3 Database generation

This section shows how our system generates a database associated with an automatically generated 3D city model whose transmission over a network (for a purpose of navigation) must fulfill some constraints related to the bandwidth of the used network.

The operation of generating a database (Figure 3) is an off-line process that supplies a VRML97 file system describing a 3D city model either generated automatically or extracted from an existing $2D\frac{1}{2}$ city model. A $2D\frac{1}{2}$ city model is a model where buildings are described using their footprints and their heights. Such models are often used in town planning projects.

The first step of generation consists in building an intermediate city model. This model is made up of roads, crossroads, buildings and parks. Each of these four entities contain several information data:

- **Architectural style** which is used for buildings, roads and crossroads. It can be an architect style given a life period, a city typical style, etc.
- **Natural style** which is used for parks and mainly describe the style of vegetation.
- **Footprints** that are used for all the entities. Like for $2D\frac{1}{2}$ models we use these footprints to describe each entity position in space. Since we want the entities to fit well with the terrain elevations, they are expressed in $3D$ space.

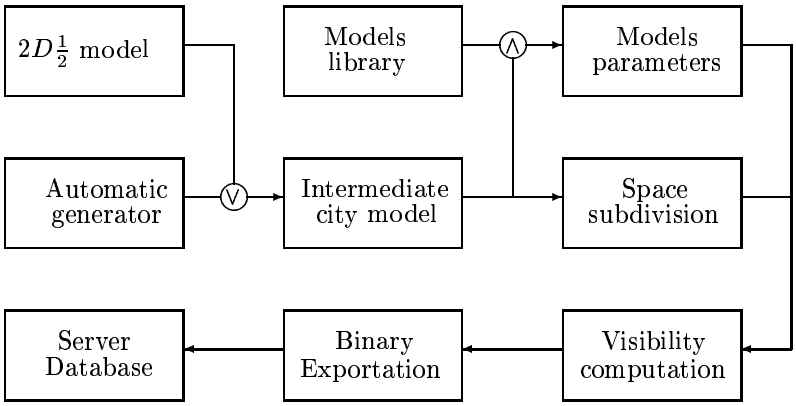

Figure 3: Database generation process.

- **Heights and numbers of floors** that are only used for buildings. Since we want to maintain a coherence between the heights of street's pavements, the roads and crossroads are assigned a constant height.
- **Adjacency relationship** between buildings. This relationship is used to handle party walls during the generation of buildings' frontages.

In addition, the intermediate model comprises an adjacency graph linking the roads and the crossroads of the city model. In this graph, a crossroad refers to the roads it accesses and reciprocally a road refers to the roads and crossroads it accesses. When using an existing $2D\frac{1}{2}$ city model, the intermediate city model is initialized with the information data of this model and the adjacency graph is generated from the information data related to the roads and crossroads. Our current system does not handle the generation of a city model from an existing $2D\frac{1}{2}$ model even though we think the use of such a model is very important when the aim is to obtain realistic street networks or to reconstruct existing cities. Rather, it generates automatically the intermediate city model using user input parameters.

Once the intermediate city model has been built, it is then used for generating the VRML97 files describing the models geometry. As said before, as our objective is to minimize the amount of data transmitted over the network, we make use of procedural models to describe the geometric models. Given the style information contained in the intermediate model as well as a library of procedural models, the generation process selects the appropriate procedural model for each entity and generates their input parameters using footprints and heights. The input parameters of each entity are written in a VRML97 file.

The intermediate city model is also used to generate the VRML97 files describing the space subdivision. As said before we use a space subdivision to optimize the network transmissions and the rendering process. As our goal is to provide city models for walkthrough

we naturally use roads and crossroads to generate the navigation space. Using the roads and crossroads footprints together with the adjacency graph contained by the intermediate model we generate a set of cells that constitute the navigation space. At the end of the space subdivision process, each cell is encoded in one VRML97 file that contains its convex hull description and its adjacency relationships.

The next step of the generation process consists in computing cell-to-objects visibility. Using the results of the two last steps we compute the potentially visible set of objects for each generated cell. The visibility process also generates the ACH for each potentially visible object. The results of this process complete the description of each previously generated cell. Consequently each cell contains its visibility relationship and the ACHs of its potentially visible objects.

The last step of the process exports the generated VRML97 file system into our VRML97 compressed memory mapped format.

3.1 Generating an intermediate city model

The process in charge of generating automatically an intermediate city model (Figure 3) first generates the street network of the city. As our main goal is real time navigation and not the quest of city realism, we have developed a simple and fast algorithm to generate the street network. For a better realism one could refer to Parish [21]. The main idea behind this algorithm is to generate a regular street network that is then modified by several disturbance operations consisting in adding noisy data to existing ones. The resulting street network is used to generate the footprints, elevations and styles of the different entities. The different steps of the generation process are given in figure 4.

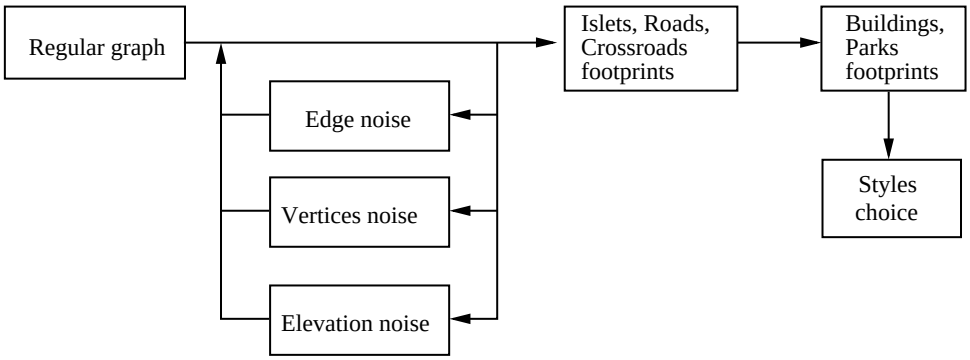

Figure 4: Automatic generation process.

First step generates a regular non-oriented graph. The vertices and edges of this graph represent crossroads and roads respectively (Figure 5a). The user input data are the size of

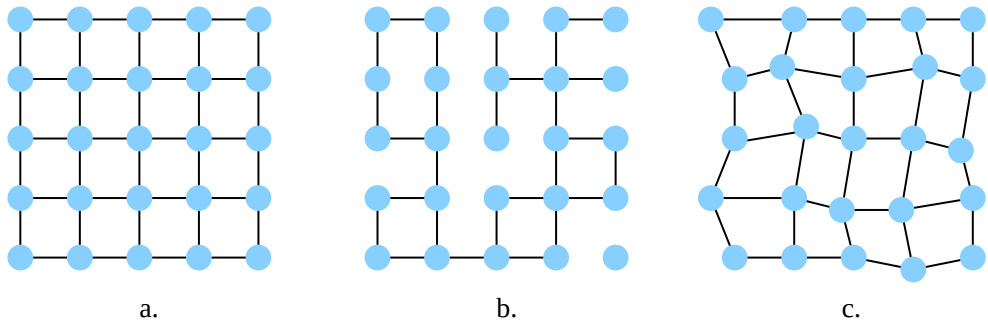

Figure 5: Generated street graph and effects of additional noise.

the town, the number of crossroads and the length of the roads. Each vertex is assigned its 3D coordinates with no height (i.e. the z coordinate is equal to 0).

Second step disturbs the regular graph by adding to it a certain noise. We make use of three types of noise parameterized according to the machine random number generator. The first noise (Figure 5b) deletes some of the edges (roads). The second noise moves the vertices (crossroads) on the ground plane while respecting the constraint that does not allow two edges (roads) to overlap (Figure 5c). The last noise generates elevation values for vertices (crossroads). The heights of the crossroads are generated according to a random seed value and a maximum height constraint. As this noise may give unsatisfactory results, the user can use an height map for the crossroads. At this point, the street network is completely defined. If the user is satisfied by the resulting network, the generation process can move on to the next step.

Third step generates roads, crossroads and islets footprints. According to a constraint on the minimum and maximum widths of roads, the system randomly generates footprints for roads and crossroads. Figure 6a shows the footprint generation for a crossroad C. The third step is implemented as follows. It first computes the vertices of the crossroad footprints. Knowing the road widths W_1 and W_2, it constructs the two rays R_1 and R_2 that are parallel to their associated roads as seen in figure 6a. The first footprint vertex P_1 is given by the intersection of the two rays. This operation is repeated for each other adjacent road to obtain the other footprint vertices. Once this operation has been completed for each crossroad, the algorithm generates footprints for roads and islets as shown in figure 6b. Note that the computed rays are stored in the data structure associated with the generated crossroad footprints so as to generate correct pavement models.

Fourth step generates building and park footprints using previously generated islet footprints. For a park, the islet footprint is used as is. For buildings, islet footprints are

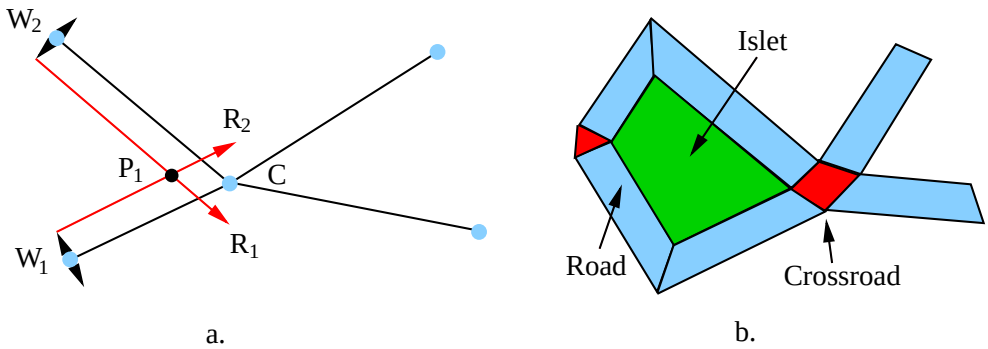

Figure 6: Footprints generation for crossroads, roads and islets.

transformed according to constraints on the building sizes. Figure 7 shows this transformation. The islet footprint is first scaled to obtain the back faces of the buildings (Figure 7a). Front and back faces are then linked by extending the back faces until they cross front faces. This results in intermediate footprints (Figure 7b). Finally, the obtained intermediate footprints are subdivided into building footprints (Figure 7c) and adjacency relationships between buildings are stored in the data structure associated with the generated building footprints. Here again, the building sizes are chosen randomly while respecting the constraint of minimum and maximum sizes. This method is simple and capable of generating building footprints on non convex islets.

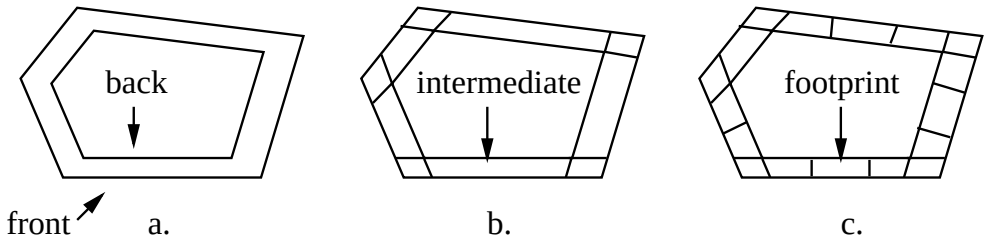

Figure 7: Footprints generation for buildings.

Last step generates architectural and natural styles. The styles are randomly chosen in a set of libraries and assigned to roads, buildings and parks. In order to have a better control on the style distribution, one could use density maps to make the styles dependent on the different city areas.

3.2 Procedural models parameters

This section shows how procedural model parameters are generated given an intermediate city model and a hand-made library of procedural models. For each entity (road, building, etc.) of the intermediate city model, our algorithm selects in this library one procedural model that fits well with the style of this entity. Then, it generates the input parameters for the selected procedural model. Each procedural model of the library is encapsulated into a VRML97 extern PROTO (see Section 4). The algorithm generates one VRML97 file per entity. Figure 8 shows an example of a building description example using a footprint, two floors and adjacency data. The procedural model used is named Build01. The 3D footprint is described by the coordinates of its vertices given in a clockwise order in the field *footprint*. The *floor* field indicates the number of floors the building prototype must generate, and the *adjacentHeights* field contain the heights of the adjacent buildings for each frontage. The adjacent building heights are used to prevent procedural model from generating geometric details on frontages that are occluded by adjacent buildings. The use of input parameters is explained in section 4. Note that the important point of this section is the fact that a very small amount of data is needed to describe a building or any other entity. The size of the file shown in the previous example is only 360 bytes if it is stored in the VRML97 UTF8 format. Consequently, the inputs parameters of procedural models generated for a city made with 100,000 buildings will require only 35Mb of memory. Furthermore this memory size will be drastically reduced using the compressed binary format presented in section 3.5.

3.3 Space subdivision

Our space subdivision method makes use of the generated intermediate city model. It subdivides the navigation space into 3D cells delimited by the roads and crossroads, once their associated footprints have been extruded. Figure 9 shows a space subdivision result. The extruded road footprints are subdivided into three new cells to reduce the size of the PVSs generated by the visibility computation process (see section 3.4). Each resulting cell is described with a new VRML97 built-in node named *ConvexCell* and saved in separate VRML97 files (Figure 10). The cell of the Figure 10 has two adjacent cells that are referred to using their relative urls in the field *cellUrl*. The Url notation used to refer to the cells is an extended Url like those used to refer to extern PROTOs. We have introduced it to allow nodes to refer to other nodes and specially to refer to cells using their DEF names even though they are not in the same file. Using this mechanism, we can distribute the cells descriptions among separated files in order to obtain a good network load granularity. That is to say, when a client needs a cell, it downloads only the description of this cell. The Urls of the adjacent cells are generated using the adjacency graph contained in the intermediate city model. The fields *coord* and *cellCoordIndex* are used to describe the cell convex hull. During navigation, the user is constrained to stay within these hulls. The union of all the convex cells is named the navigation space (see section 5). The fields *children* and *coverageHints* are used to manage visibility information and are filled by the visibility preprocessing step described in the next section.

```
#VRML V2.0 utf8

# Extern PROTO invocation i.e. building model

EXTERNPROTO Build01 [
  field MFVec3f footprint
  field SFInt32 floors
  field MFInt32 adjacentHeights
]
"Library/build01.wrl"

# model instanciation

Build01{
  footprint [12.2065,0,246.818, 32.1485,0,248.112,
             37.0815,0,228.059, 9.39851,0,229.303 ]
  floors  2
  adjacentHeights [ 0, 6, 0, 8 ]
}
```

Figure 8: VRML97 file generated for one building description using the EXTERNPROTO mechanism.

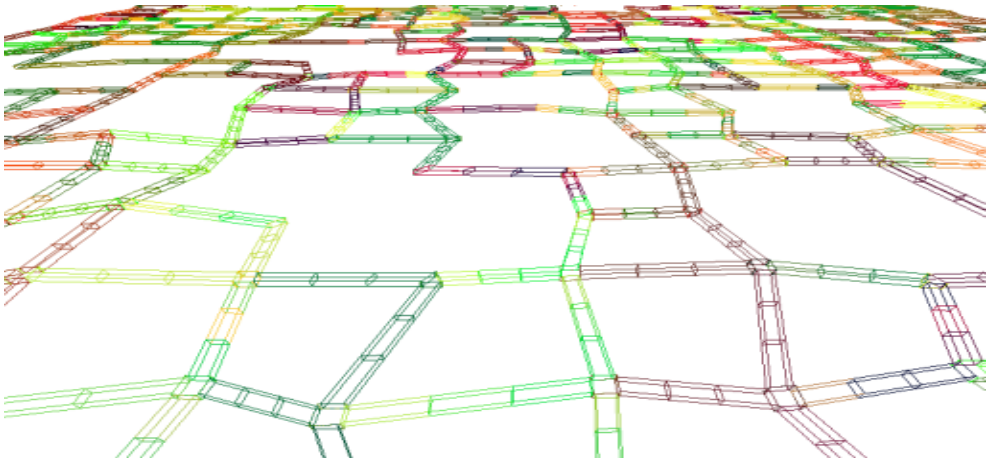

Figure 9: Generated space subdivision.

3.4 Visibility computation

Our visibility preprocessing consists in finding buildings, roads and crossroads potentially visible from each cell. In our algorithm, which is not conservative, we compute a PVS for

```
#VRML V2.0 utf8

DEF cell_1 ConvexCell {
   cellUrl [ "Cells/cell_100.wrl#cell_100" ,
             "Cells/cell_102.wrl#cell_102" ]
   children [ ]
   coverageHints [ ]
   coord Coordinate {
         point [
            45.0203 0 305.857 , 34.3379 0 305.329 ,
            41.9268 0 317.121 , 41.9268 4 317.121 ,
            34.3379 4 305.329 , 45.0203 4 305.857 ]
      }
   cellCoordIndex [
      0 , 1 , 2 , -1 , 3 , 4 , 5 , -1 ,
      1 , 0 , 5 , 4 , -1 , 2 , 1 , 4 ,
      3 , -1 , 0 , 2 , 3 , 2 , -1 ]
}
```

Figure 10: VRML97 file generated for a cell description using a *ConvexCell* built-in node.

each corner of the cell. The union of the obtained PVSs gives the PVS of the cell. In our system, the PVS of a cell corner is computed in screen space by rendering the scene for six cameras having the same COP (center of projection). As shown in figure 11a, the view direction of each camera is perpendicular to one face of a box. Such a box will be called rendering box from now on. The COP shared by the six cameras is the center of the rendering box and the FOV (field of view) of each camera is equal to 90 degrees. The projection plane of a camera is a face of the rendering box. Figure 11b shows the positions of the rendering boxes within a cell. The eight rendering boxes of a cell are not centered at the corners. Rather, a box center is placed close to a corner so that the box be inside the borders of the cell that are supported by the frontages of the buildings. In this way, the frontages of the buildings become occluders which reduces the size of the PVSs.

For each camera, all the geometric objects (generated using the method explained in section 3.2) are rendered, which gives 48 images. The geometric objects may be roads, crossroads or buildings. To compute rapidly these images, the rendering is performed using an OpenGL graphics hardware and the bounding box of each object is checked for intersection with the viewing frustum of the camera. If the bounding box intersects the frustum, the middle LOD (small number of polygons) is rendered for this camera. As these LODs respect the building hulls (see Figure 14) we do not introduce occlusion errors. The objects, individually saved in a separate VRML97 file, are loaded using a main root file that refer to them using *Inline* nodes. Figure 12 shows three 512x512 images generated for 3 cameras,

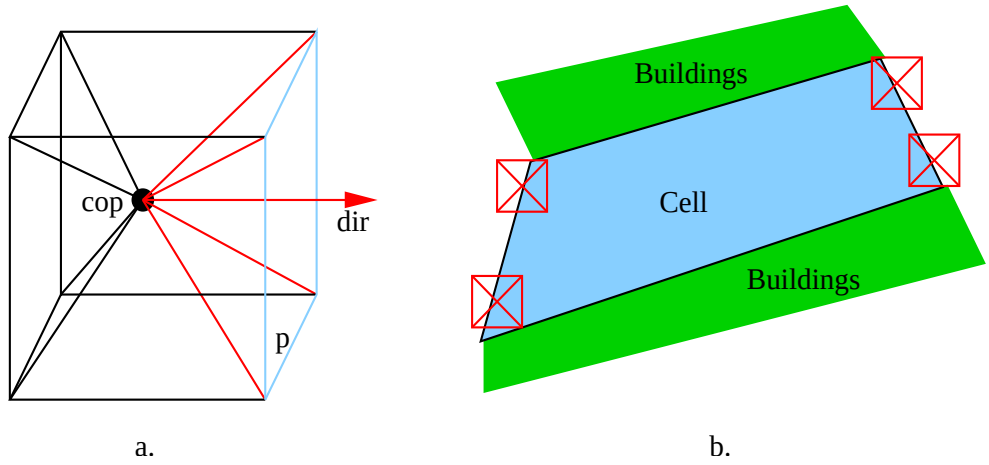

Figure 11: (fig. a) Rendering box: One of the six camera is in red, it is defined by its center of projection (cop), its direction (dir) and its projection plane p. (fig. b): Rendering box positions inside a cell (top view).

each one associated with a different rendering box. As shown in this figure, a procedural model, referred to with an *Inline* node, is displayed with a unique color which is assigned to the memory pointer pointing to this inline node. So, the contents of the image directly gives the memory pointers of all the *Inline* nodes visible from the COP of this camera. Consequently, the contents of all the 48 images gives the memory pointers of the *Inline* nodes that makes up the PVS of the cell. Note that one could use the OpenGL occlusion query extension to speed up this process.

In addition, for each camera C_i (of the 48 cameras associated with a cell), we count the number of pixels N_{ij} covered by each visible *Inline* node I_j. The total number of pixels N_j^{total} covered by a visible *Inline* node I_j is then:

$$N_j^{total} = \sum_{i \in [1,48]} N_{ij}$$

Let N_{nodes} be the number of *Inline* nodes visible for the 48 cameras. The total number of covered pixels N_{pixels}^{total} for the 48 cameras is then:

$$N_{pixels}^{total} = \sum_{j \in [1,N_{nodes}]} N_j^{total}$$

Figure 12: Three different rendered images. Each building is displayed with a unique color corresponding to a memory pointer that points to the *Inline* node describing the building. The background is displayed in white for a visual purpose but should be black since a NULL memory pointer has a value 0.

The ACH (Average Coverage Hint), denoted ACH_j, associated with each visible *Inline* node I_j is computed as:

$$ACH_j = \frac{N_j^{total}}{N_{pixels}^{total}}$$

The properties of the ACH values are the following:

$$\begin{cases} \forall j \in [1, N_{nodes}], ACH_j \in [0, 1] \\ \sum_{j \in [1, N_{nodes}]} ACH_j = 1 \end{cases}$$

As will be seen later (see section 5), to make walkthrough more efficient, the buildings closest to the current cell must be first downloaded. To this end, the *Inline* nodes, visible from a cell, are sorted according to their distance to the center of the cell before being copied into the cell's *children* field (see figure 10). The associated ACHs values are copied into the field *coverageHints* (sorted in the same order).

As mentioned in section 3.3 the extruded road footprints are subdivided into three cells. This allows to reduce the size of the cells' PVSs. Indeed, the number of visible objects is very high when the viewpoint is located at one of the street ends, so, using a single cell for a street would generate huge PVSs. Furthermore, as shown in figure 13a, when rendering the scene from a viewpoint, placed at one of the extremities of a cell and looking at the green buildings, many objects (red in the figure) located within the view frustum are needlessly sent to the graphics hardware whereas they are not visible. To cope with this problem, one could subdivide the street into many cells. The consequence is that visibility gets better for each cell but the PVSs of the cells contain redundant data, which drastically increase the memory size of the database. Actually, if we subdivide the street into four or more cells, the two center cells nearly have the same PVS. Furthermore, when using biggest cells we

can provide the prefetching algorithm with better results (see section 5). This is why we prefer to subdivide the streets into three cells (Figure 13b), which is in our opinion the best compromise.

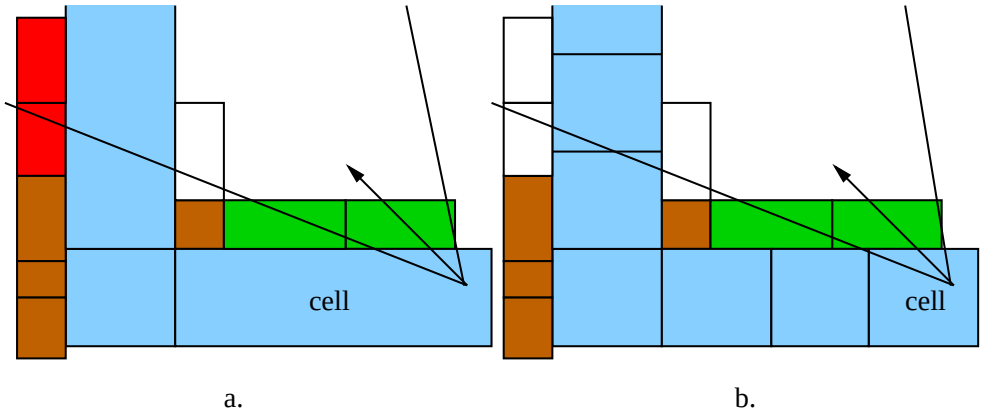

Figure 13: The cells are represented with a blue color. The white buildings are those that are not in the PVS of the cell that contains the viewpoint. The buildings in brown are in the PVS and frustum culled. The buildings in green are in the PVS and sent to the graphics hardware. The red ones are in the PVS and sent to the graphics hardware but are not visible. Fig.a: The PVS of the bottom rightmost cell contains the red upper leftmost buildings. Fig.b: After cell subdivision the new bottom rightmost cell is smaller and its PVS does not contain the red upper leftmost buildings anymore because they are hidden by the green buildings.

Our visibility solution is not conservative and the precision of the visibility computation is mainly related to the position of the rendering boxes inside the cells and to the pixel resolution of the cameras. However, as the topology of the cells is known, there is no problem to find out good positions for the rendering boxes (Figure 11b). Furthermore, as the rendered objects are buildings and roads (big objects), a 512x512 resolution seems enough to avoid visibility artifacts.

3.5 Database export

Once visibility has been computed for all the cells, our system generates a new VRML97 root file that contains a set of *Viewpoint* nodes that describes a certain number of viewpoints at which the user can start navigation. In our implementation we generate only one viewpoint that is placed in the middle of the first generated cell. The *Viewpoint* node we use contains an additional field named *cellUrl*. This field is set to an extended Url pointing to a valid cell node. A cell node referred to with one of such viewpoints is considered as linked. A

Viewpoint must lie inside the convex hull of the associated linked cell. If the two above constraints are respected, the linked cell is used as the current navigation cell when the *Viewpoint* gets bound (in the sense of VRML97). Then, navigation starts from the current cell using the visibility data contained in the cell description (see section 5). Once the new root file is generated, we export all the database files (root, cells and entity files) in a compressed VRML97 binary format.

The binary format we propose is a simple raw format that allows direct memory mapping on Linux and MS Windows systems. Since a client downloads files during navigation, we need very fast transmissions from the network card to the client's memory. To this end, the file format must allow to avoid any parsing on the client side. This is done by using direct memory mapping for all the fields of the VRML97 nodes. That is to say, each value of the VRML97 fields is memory dumped to the binary file and the fields are referred to using integers rather than UTF8 strings. Our binary format uses the file header `#VRML V2.0 mbinary`. Each VRML97 file contains a root scene graph that defines a scope which is used to refer to the user prototypes. Each user prototype also defines a scope that might contain other user prototypes. When exporting a root scene graph or a user prototype into a file according to our binary format, we first write the number of prototypes, referred to with the associated scope, using a 32bits integer. Next, each *PROTO* or *EXTERNPROTO* is in its turn written into the file. We write an 8bits integer value to distinguish *PROTOs* and *EXTERNPROTOs*. The interface of the written prototype is then encoded using UTF8 strings to describe the field types and the field names. The fields default values are written using raw memory format. Under Linux or MS Windows, integers, floats and strings are directly dumped from memory to file. If the field is a multiple field, the number of values is written using a 32bits integer just before writing the array of values that is dumped from memory. If the field is an *MFNode* or an *SFNode*, each node contained by the field is written using the writing mechanism explained hereafter. The prototype body is written using the same format that the one used to write nodes. As we can have different kinds of node due to the prototyping mechanism, when writing a node we first write its name in UTF8 text format followed by its number of non default fields (written fields). As the field names are invariant inside a node, we sort them in an alphabetic order and assign a number to each of them. The fields that are not default fields are then written into the file by first writing their associated number followed by their value. The field number is written using a raw integer while the field value is written using the same method as for PROTO fields default values. The binary file we obtain is then compressed using the zlib [14] lossless data-compression library and exported using the extension *wrz*.

4 Scripting language

In this section we briefly present the new VRML97 scripting language we have developed to describe our procedural models. As the VRML97 ECMAScript language provides a few facilities for generating 3D models and partial scene graph structures, we propose a solution which consists of an extension of the L-system [23] language that allows such operations.

A L-system is a context-sensitive parallel grammar that performs rewriting operations on symbols controlling a graphic turtle. In our language we keep the grammar functionality but we do not make use of the turtle paradigm. Nevertheless, we easily emulate a turtle to describe plant models. In addition to the classical L-system functionalities such as the stochastic or condition guided rules choices, we introduce the possibility to cut a parallel rewriting process by placing a "!" character before the rule to be rewritten at first. This point is very important since it allows for the instantiation of VRML97 nodes. As an example, let consider the case of a natural tree. A L-sytem rule associated with a branch of this tree may generate two other rules, one for each of its leaves. Let us call leaf rule these two new rules. The two leaf rules can be rewritten in parallel since they do not share any information. Now, if we consider a rule describing a building frontage that generates multiple window rules, the first one generates the window geometric model while all the subsequent ones instantiate this window model to repeat the window pattern lying on a frontage model. We must ensure that the generation of the first window model ends before other rules uses it by placing the character "!" just before the first rule. In this way, once the first rule has been rewritten, all the other rules can get rewritten in turn and in parallel.

In addition to this mechanism we have implemented built-in rules that allow dynamic allocation of VRML97 built-in or prototyped nodes through parameter passing and fields access. In order to allow for geometry generation we make use of mathematical operators using floats and integers, list manipulations as well as trigonometric functions. Using these mechanisms we can easily generate meshes and node hierarchies for pattern based geometric models using only one rewriting call.

The procedural models for generating buildings take as input the information data generated in section 3.2 to provide the geometric representation of each level of detail. In our implementation we generate three LODs per building. Figure 14 shows these levels for a given building using a certain set of input parameters. Each level is divided into two parts which are the frontages and the roofs. We use pattern repetition to describe each part. A ground floor is modeled using a row of windows, a door as well as another row of windows. The other floors (including roofs) are modeled using one row of windows. The model uses adjacent building heights to generate party walls. As a party wall does not contain any door or window, we make it start from the adjacent building height to reduce the number of polygons to be rendered. The frontage's wall is the same for level two and level three, so the geometry is shared by the two levels using instance sharing. Since the frontages of this model are nearly flat, all the geometry of each frontage is encapsulated in a new *OneSideGroup* node that uses the normal of the supporting plane to cull all the geometry in one go. The generated levels are then placed in the *children* field of an ALOD node (implementing our LOD selection mechanism using ACHs, see section 2).

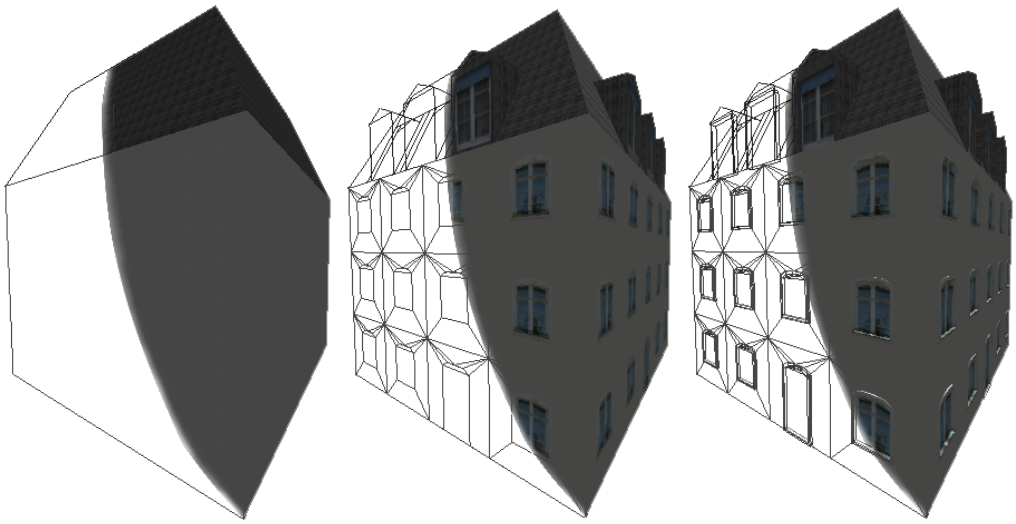

Figure 14: LODs generated for a building.

5 Real-time navigation

In order to implement and experiment with our algorithms, we use our framework, named Magellan, which is outlined in Figure 15. This framework allows the automatic generation of *client*, *server* and *builder* applications for Linux, Win32 and SunOS platforms. Client/Server applications manage automatically multi-server/multi-client connections.

Using C++ inheritance, these applications automatically manage *generic visual objects*. These *generic visual objects* are responsible for their read and write from/on files, their transmission across the network and their display. The framework provides a C++ message class for marshalling/unmarshalling the data transmitted over the network. These objects use OpenGL calls to generate their visual representation. In this way we can easily implement new visual objects that will be used by these applications. Furthermore, making the objects responsible for their display allows for hybrid rendering such as merging polygon-based and point-based renderings[22].

Besides, the framework allows to develop and use any new *rendering*, *motion* and *prefetching* modules. In addition, the user can integrate new modules into the framework such as the *rewriting* thread used for generating geometric models associated with procedural models. Each of these modules can be implemented with threads if needed. They can all access the scene graph handler to manipulate the generic scene graph encapsulated by this handler.

The generic scene graph data structure is composed of one root node and any needed extern root nodes. Each root node is associated with a file. A node can be either a *VisualNode*, a *TextureMapNode* or a *ConvexCellNode*. The *ConvexCellNode* nodes are used

if the scene is subdivided into cells [32]. With this kind of node, the *rendering module* takes advantage of cell-to-cell or/and cell-to-geometry visibility information. In addition, The *prefetching module* can prefetch cells using motion prediction [7, 2, 4], and the *motion module* can utilize the result of space subdivision to speed up the collision detection. The framework provides several other classes of nodes that are beyond the scope of this report.

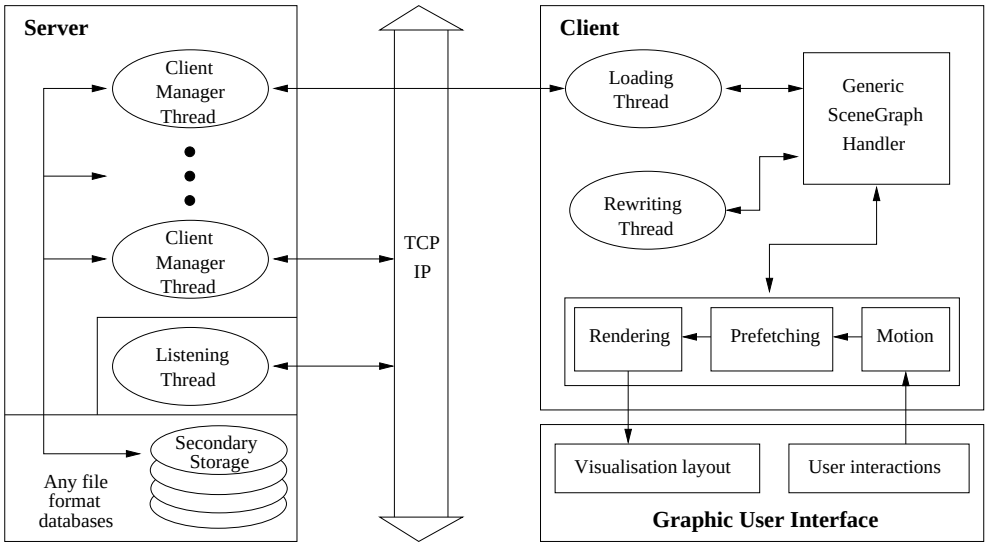

Figure 15: Magellan architecture. Client/Server communication uses TCP/IP protocol. The server side assigns a thread to each connected client. Each thread marshalls and sends data nodes on demand. The requested data can be complete or partial files to allow the transmission of progressive data. A client is composed of two main threads and any other user threads. The loading thread asks for nodes, unmarshalls nodes and adds nodes to the scene graph through the scene graph handler. The main thread performs three tasks consisting in generating the viewpoint motion, computing collisions and gravity, and updating the scene graph depending of the viewpoint position. The prefetching module uses motion predictions to download future visible cells. The rendering module selects the visible cells or objects and invokes their display methods. Finally, the added user module, named rewriting thread, performs the rewriting of parallel L-system scripts. All these tasks can access and modify the generic scene graph if needed through the scene graph handler.

5.1 Scene graph structure

Recall that the scene graph is composed of one root node and an array of extern root nodes. With each extern root node is associated a reference counter that is used to know how many

*shared loading*s have been performed for each node. When a user node needs to download another extern root node, it asks the scene graph handler to download this extern node using the *shared loading* mechanism. If this extern node is not in the scene graph, the scene graph handler sends a loading request to the server while setting the associated counter to 1, else it only increments this counter. In our implementation the VRML files referred to with *Inline* nodes as well as the textures are downloaded using this mechanism. Conversely, the cell nodes are downloaded by the client's modules using the *single loading* mechanism that always set to 1 their associated counters.

Since a database may be too huge to be completely stored into a client's memory, we need an heuristic to select the extern root nodes to be removed (because of the limited size of the free memory) when the loading thread tries to add a new node in the scene graph. To choose the root nodes to be removed we use the adjacency graph. We sort the cells by depth in the adjacency graph, starting from the current cell. Then, we remove the most far-off cells until the memory size needed is freed. If an extern root node is shared by two or more different cells and one of these cells ask for its removal, we only decrement the reference counter of the shared node.

In addition, the scene graph handler contains the current viewpoint which is one of the *Viewpoint* nodes generated in section 3.5. As said in section 3.5, if the current viewpoint is associated with a cell url it is considered as linked and the referred cell becomes the current navigation cell. All the modules access the current cell through the current viewpoint. When a viewpoint gets linked with a new cell, the scene graph handler asks for the *asynchronous* single loading of the new adjacent cells. If the linked cell is not already downloaded the scene graph handler asks for its single loading in a *synchronous* way before asking for the loading of its adjacent cells. In our implementation, when a cell arrives at the client, it automatically asks for the *asynchronous* shared loading of its children *Inline* nodes.

5.2 Motion module

The motion module obeys the user orders to generate the motion of the current viewpoint. It is also responsible for computing collisions and for simulating gravity. The collision detection is accelerated by using only the geometry contained in the current and the adjacent cells. The update of the current cell is performed as follows. The collision and gravity result is used to compute the new parameters of the current viewpoint. If the new position of the viewpoint is inside a cell adjacent to the current cell, this adjacent cell becomes the new current cell and the current viewpoint gets linked to this cell. In this way, the scene graph handler can refresh itself if needed (see. section 5.1). The new parameters of the viewpoint are added to the prefetching module's FIFO explained hereafter.

5.3 Prefetching module

The prefetching module uses a navigation history to determine the next cell that will be visited by the user. Once this next cell has been determined, its adjacent cells are prefetched. The module uses a FIFO of viewpoints to store the navigation history. In our implementation

we limit the FIFO size to handle the two most recent viewpoints. The navigation history can be used together with any motion prediction algorithm. For motion prediction, we use a simple ray casting algorithm that is performed only if the most recent viewpoints have two different positions regardless of their viewing directions. In addition, we maintain a list of adjacent cells that have not already been prefetched. This list is reset whenever there is a change of the current cell. The motion prediction algorithm works as follows. We trace a ray having as origin the position of the less recent viewpoint contained in the FIFO and passing through the position of the other viewpoint of the FIFO. Then, this ray is checked for intersection with each adjacent cell not already prefetched. The cells adjacent to the intersected cells are then prefetched asynchronously (if they are not already in the scene graph handler). Prefetching amounts to loading requests that have a priority lower than that of usual loading requests.

5.4 Rendering module

The goal of the rendering module is to render the scene at an interactive frame rate with a minimum number of visual artifacts. Since the geometric objects are responsible for their display, the rendering algorithm only computes the amount of polygons to be used for the next rendering to maintain a given frame rate (fps) given by the user. This amount of polygons will be called *polygon budget*. Using an average frame rate value estimated from the rendering time of some of the last computed frames and the last polygons budgets used for generating these frames, the algorithm determines the new polygon budget to be used to maintain the frame rate fps given by the user. Once this polygon budget has been computed, the algorithm passes it to *settleInPvs* which is one of the methods of the current cell, which is a visual node. This method is implemented by each visual node to perform frustum culling on its children nodes (using their bounding boxes) and to allow these nodes to share out the polygon budget associated with it. Once the execution of this method has ended, the algorithm invokes the *display* method of the current cell to compute the new frame according to the *settleInPvs* results encapsulated in each visible node.

The current cell rendering algorithm makes use of two methods. The *settleInPvs* method utilizes the polygon budget N^{poly} and the ACHs to make the visible children nodes of the cell share out the polygon budget. The *display* method simply invokes the display method of the visible children. The interesting point of the process is the polygon budget sharing which is performed by the *sellteInPvs* method of the cell. In this method, we first performs frustum culling using the bounding boxes of the *Inline* nodes that are stored in the *children* field of the cell. For each visible node, its ACH is normalized using the number of nodes that are visible. We compute the polygon budget N_i^{poly} to assign to a visible node i using its normalized ACH denoted ACH_i as follows: $N_i^{poly} = ACH_i * N^{poly}$. As each object referred to with *Inline* nodes is an ALOD, it uses this budget to select its best suitable LOD and returns N_{used}^{poly} which is the exact number of polygons of the selected LOD. The budget of polygons used for the rest of the children nodes is now $N^{poly} - N_{used}^{poly}$ polygons. This process is repeated for each child node, starting from the node having the highest ACH and ending

with the child node having the lowest one. Note that this budget assignment method is interesting for any system using level of details.

6 Results

In this section we provide some results showing the performances of our navigation system regarding the size of the generated databases, the quality of network transmissions and interactivity during navigation.

6.1 Database size

One important feature of any remote walkthrough system is that the smaller the database size, the faster the network transmissions. This is why, in our method, we use procedural models encoded using a compressed binary format.

To assess the quality of our compressed binary format, we have generated 10 databases of different dimensions and compared the sizes of the associated generated files for four different formats which are: UTF8 text, compressed UTF8 text, binary and compressed binary format. We have ascertained that, for a same database, the size of a binary file is always less than 98% of the size of a UTF8 text file. This shows that although a binary file is faster to parse (see section 3.5), it does not entail additional storage. Next, we have compared the average compression ratio obtained when compressing UTF8 text files with the one obtained by compressing binary files. The obtained results show that the size of a compressed UTF8 text file is equal to about 47.4% of the size of a UTF8 text file, while the size of a compressed binary file is equal to about 46.2% of the size of a UTF8 text file. These two ratios are then nearly similar, it show that using the binary format do not introduce lost within compression quality.

Size	Entities	BIN-ZIP	Model	Ratio
$122500m^2$	219	147.0KB	218684KB	0.067%
$176400m^2$	348	201.9KB	311748KB	0.064%
$240100m^2$	490	272.1KB	457088KB	0.060%
$313600m^2$	662	354.5KB	610400KB	0.058%
$396900m^2$	859	447.6KB	810380KB	0.055%
$490000m^2$	1077	541.9KB	1149452KB	0.047%

Table 1: Database sizes and compression ratios.

Although our binary file format offers fast parsing possibilities and good compression ratios, the main compression is achieved when using procedural model parameters to describe the geometry of the models. To compute the compression ratios, we have compared the size of the city models (used as test scenes) when stored in compressed binary format with the

size of these same models when their geometry is completely reconstructed using our L-system rewriting system. The table 1 shows the results for a set of databases. The columns respectively represent the ground surface of a city model, its number of entities, the size of the database in a compressed binary format, the size of the model when reconstructed and finally the ratio between the two last sizes. This table confirms the fact that procedural models are of high interest in remote navigation because of their small size which makes them well suited to low bandwidth networks.

6.2 Transmission quality

In order to analyze the transmission quality, we have recorded a walkthrough path called SE (on the client side). This path passes through the streets of a 700m square model that contains 860 entities and five parks (Figure 16). Then, we have performed two walkthroughs (with and without prefetching) using the recorded path and a simulated network connection with a bandwidth of 56Kb/s.

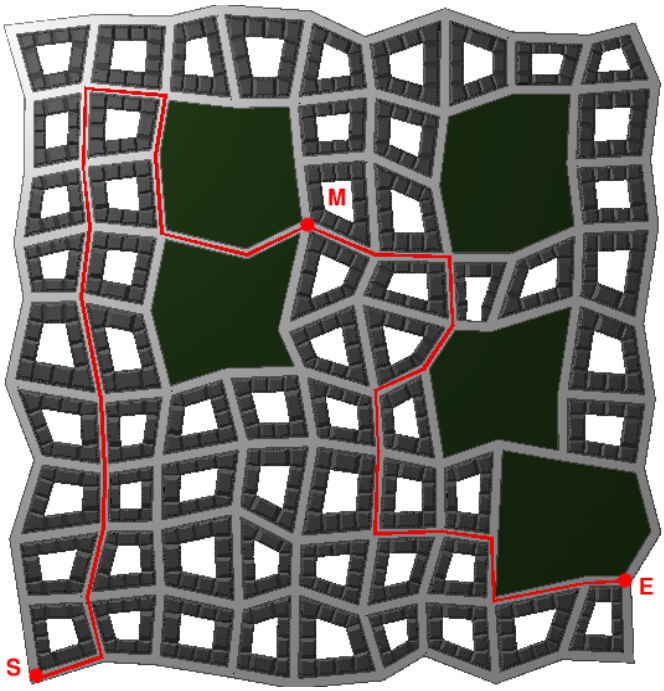

Figure 16: Top view of the test city model and path used for networked walkthrough analysis.

The navigation speed for these walkthroughs has been set to 15km/h. During these walkthroughs, we count, for each new frame, the number of entities that are in the PVS of

the current cell. Among these entities, we count those that have not been already downloaded and those whose geometry has not been already generated (rewritten) by the rewriting thread. This allows us to compute the percentage of needed entities that have been downloaded (called downloading quality) and those that have been rewritten (called rewrite quality). If both values are equal to 100%, the transmission quality is perfect for the new frame. Figure 17 shows how these two percentage values evolve over time and the size (expressed in KBytes) of each of the entities downloaded during the walkthrough, when using or not prefetching.

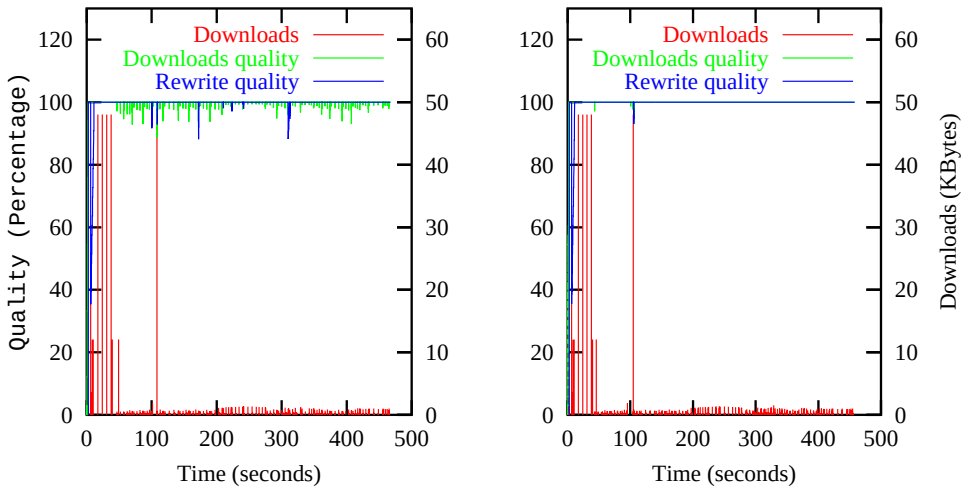

Figure 17: Left: results of downloading over time, downloading quality and rewriting quality for the walkthrough SE, without using prefetching. Right: same, using prefetching.

When looking at the downloading plot, we can observe higher values at the beginning of the walkthrough because of the nine uncompressed TGA texture maps (of size ranging form $12KB$ to $48KB$) used in our test scene. To overcome this problem, one could use compressed progressive image representation such as JPEG2000 or PTMs for which efficience when using ACHs is detailed in [?]. Nevertheless, downloading is low thanks to our procedural models and our compressed binary format. In addition, downloading is homogeneously distributed over time owing to our spatial subdivision and visibility computation results. Finally, the downloading and rewriting quality plots show that prefetching allows to obtain a perfect transmission quality (100%) most of the time.

6.3 Interactivity

Recall that our system relies on the frame rate history to compute the *polygon budget* to be used for each new frame construction (Section 5.4). To show that our system adapt to different kinds of computer to achieve interactivity, we have simulated a walkthrough using the path SM (Fig. 16) on two different computers. For each computer, we have performed this walkthrough twice, noting that for the second time all the downloaded data were kept into memory. Figure 18 shows the frame rate and the *polygon budget* over time, measured for a Pentium XEON 1.7Ghz (1GB RAM with a Nvidia Quadro2 Pro) and a Pentium III 800Mhz (512MB RAM with a Nvidia TNT2). For these two tests, we have used a minimum target frame rate of 25fps and a frame rate history based upon the four last frames. On the frame rate plot, we can see that the frame rate of the first walkthrough (range [0s,250s]) is less stable than the second one (range [250s,500s]). This is due to the fact that the rewriting thread works in parallel with the main thread to generate the geometry of the different entities downloaded over time. Nevertheless, we can see that the target frame rate is always reached with a very small latency. This is particularly visible with the less powerful computer. On the *ploygon budget* plot, we can see that for a same target frame rate, we obtain a very higher polygon budget with the more powerful computer. This is specially visible when walking in the park which presents large PVSs (range [175s,250s]). Note that the lower the target frame rate, the higher the visual quality (Figure 19). Consequently, our system adapts very easily to computers of different capabilities and to different user goals (interactivity, visual quality, in between, etc.).

7 Conclusion

In this report we have shown that procedural models are of high interest when the aim is networked walkthrough of large scenes such as cities. Indeed, their size is very small, so they can be transmitted quickly over a network. We have generated these procedural models with a modified version of the L-system language. The geometry of procedural models is generated on the fly at the client side. We are now capable of generating buildings, streets, crossroads as well as vegetation like trees. Our library of procedural models can be easily extended to large families of objects thanks to our extended L-system scripting language.

Remote real time navigation has been made possible thanks to the visibility preprocessing that have been performed using the graphics hardware and the use of LODs selected using our ACHs. Our ACH-based method is original and more efficient than the commonly used one consisting in comparing with a threshold the distance between the viewer and the objects within the scene. Moreover, this method allows on-line selection of visual quality and interactivity.

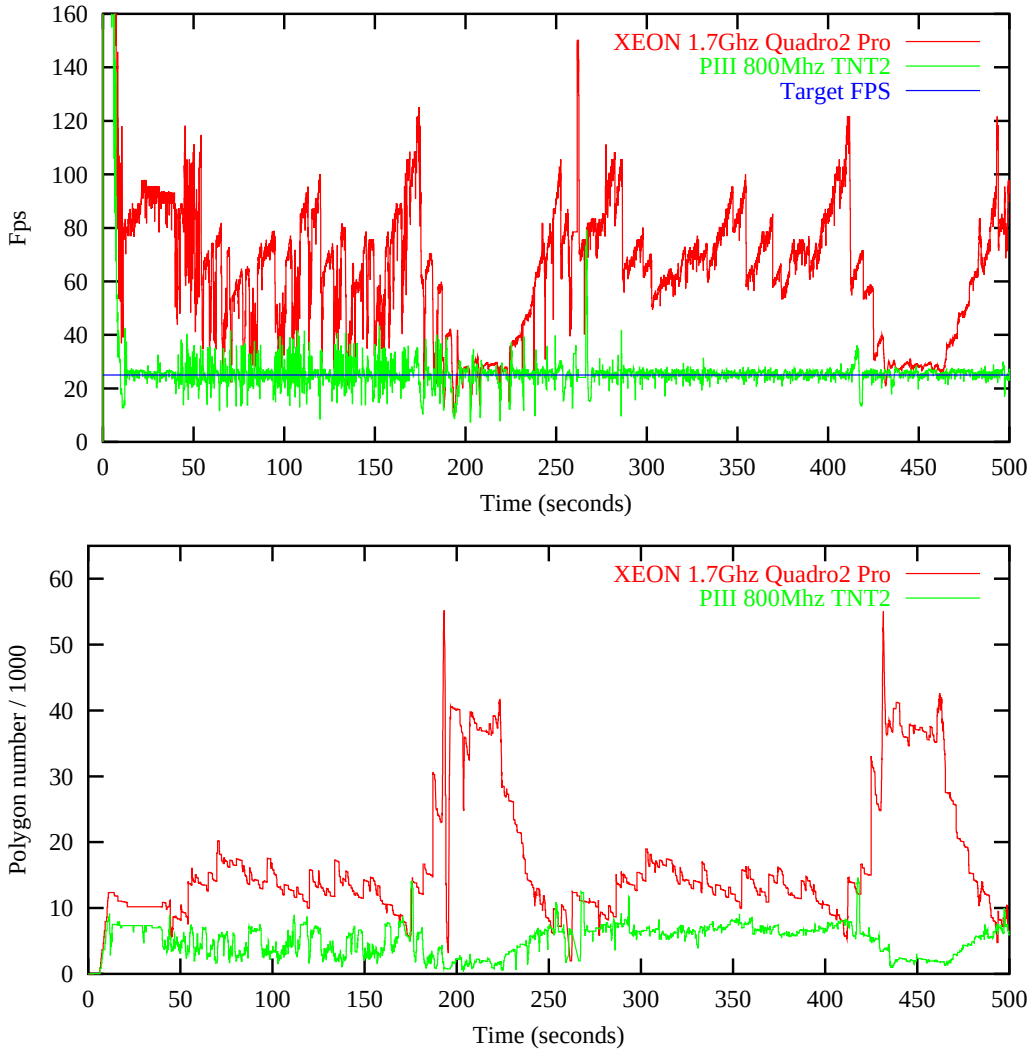

Figure 18: Top: Frame rate over time for walkthrough SM played twice on a Pentium XEON 1.7GHz, 1GB RAM, Nvidia Quadro2 Pro and a Pentium III 800MHz, 512MB RAM, Nvidia TNT2. Bottom: Polygon budget over time for the same tests.

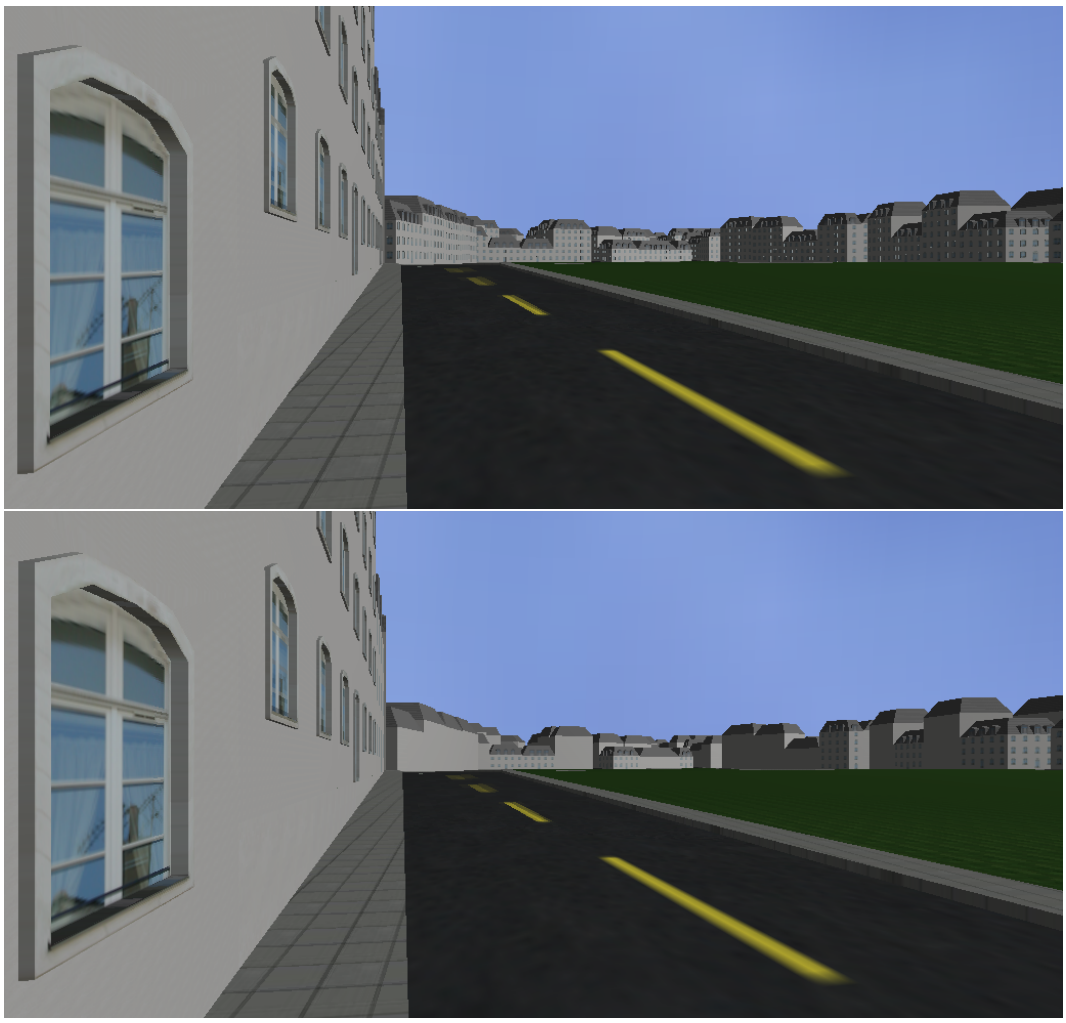

Figure 19: Walkthrough views obtained with a Pentium XEON 1.7GHz with a Nvidia Quadro2 Pro, screen resolution equal to 1024x480. Top: target fps set to 25fps, obtained 26.2fps, using 56194 polygons. Bottom: target fps set to 40fps, obtained 41.7fps, using 5703 polygons.

References

[1] J. M. Airey, J. H. Rohlf, and F. P. Brook. Toward image realism with interactive update rates in complex virtual building environements. In *Symposium on interactive 3D graphics*, pages 41–50, 1990.

[2] D. Aliaga, J. Cohen, H. Zhang, R. Bastos, T. Hudson, and C. Erikson. Power plant walkthrough: An integrated system for massive model rendering. Technical Report TR97-018, Department of Computer Science, University of North Carolina - Chapel Hill, Aug. 28 1997.

[3] C. Andujar, C. Saona-Vazquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of details through hardly-visible sets. *Computer Graphics Forum*, 19(3), 2000.

[4] R. Azuma and G. Bishop. A frequency-domain analysis of head-motion prediction. *Proceedings of SIGGRAPH'95*, pages 401–408, 1995.

[5] H. Biermann, A. Hertzmann, J. Meyer, and K. Perlin. Stateless remote environment with view compression. Technical Report 1999-784, Media Research Lab., Dept. of Computer Science, New York University, April 1999.

[6] R. Carey and G. Bell. *The Annotated VRML 2.0 Reference Manual.* Addison-Wesley Developers Press, 1997.

[7] A. Chan, R. W. H. Lau, and A. Si. A motion prediction method for Mouse-Based navigation. In H. H. S. Ip, N. Magnenat-Thalmann, and T.-S. C. R. W. H. Lau, editors, *Proceedings of the 19th Computer Graphics International Conference (CGI-01)*, pages 139–148, Los Alamitos, July 3–6 2001. IEEE Computer Society.

[8] D. Cohen-Or, G. Fibish, D. Halperin, and E. Zadichario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. In *Computer Graphics Forum*, pages 17(3):243–253, 1998.

[9] D. Cohen-Or, Noimark, and T. Zvi. A server-based interactive remote walkthrough. In *EGMM'2001*, 2001.

[10] X. Decoret, F. Sillion, G. Schaufler, and J. Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3):61–73, sep 1999. ISSN 1067-7055.

[11] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. *Proceedings of SIGGRAPH 2000*, July 2000. Held in New Orleans, Louisiana.

[12] T. Funkhouser. Database management for interactive display of large architectural models. In *Proceedings of Graphics Interface*, pages 1–8, May 1996.

[13] T. A. Funkhouser, C. H. Sequin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Symposium on Interactive 3D Graphics*, pages 11–20, 1992.

[14] J. Gailly and M. Adler. *A Massively Spiffy Yet Delicately Unobtrusive Compression Library*, 1996. http://www.gzip.org/zlib/.

[15] D. Hekmatzada, J. Meseth, and R. Klein. Non-photorealistic rendering of complex 3d models on mobile devices. In *Conference of the International Association for Mathematical Geology*, volume 2, pages 93–98, September 2002.

[16] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH*, pages 99–108, 1996.

[17] M. Levoy. Polygon-assisted jpeg and mpeg compression of synthetic images. In *ACM SIGGRAPH*, pages 21–28, 1995.

[18] Y. Mann and D. Cohen-Or. Selective pixel transmission for navigating in remote virtual environments. In *Computer Graphics Forum*, volume 16(3), pages 201–206, 1997.

[19] I. M. Martin. Adaptive rendering of 3d models over networks using multiple modalities. Technical Report RC21722, IBM T.J. Watson Research Center, april 2000.

[20] D. Meneveaux, E. Maisel, and K. Bouatouch. A new partitioning method for architectural environments. *Journal of Visualization and Computer Animation*, May 1997.

[21] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In ACM, editor, *SIGGRAPH 2001 Conference Proceedings, August 12–17, 2001, Los Angeles, CA*, pages 301–308, New York, NY 10036, USA, 2001. ACM Press.

[22] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 335–342. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[23] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer Verlag, 1991.

[24] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[25] S. Rusinkiewicz and M. Levoy. Streaming QSplat: A viewer for networked visualization of large, dense models. In *Symposium on Interactive 3D Graphics*, pages 63–68, 2001.

[26] C. Saona-Vasquez, I. Navazo, and P. Brunet. The visibility octree. a data structure for 3d navigation. Technical report, Universitat Politechnica de Catalunya, Spain, 1999.

[27] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion. Conservative volumetric visibility with occluder fusion. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 229–238. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[28] D. Schmalstieg and M. Gervautz. Demand-driven geometry transmission for distributed virtual environments. *Computer Graphics Forum*, 15(3):C421–C432, Sept. 1996.

[29] D. Schmalstieg and M. Gervautz. Modeling and rendering of outdoor scenes for distributed virtual environments. In D. Thalmann, editor, *ACM Symposium on Virtual Reality Software and Technology*, New York, NY, 1997. ACM Press.

[30] B. Schneider and I. Martin. And adaptive framework for 3d graphics over networks. *Computer and Graphics*, 23:867–874, 1999.

[31] E. Teler and D. Lischinski. Streaming of complex 3D scenes for remote walkthroughs. In *Computer Graphics Forum*, volume 20(3), pages 17–25, 2001.

[32] S. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, pages 61–69, 1991.

[33] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Eurographics Workshop on Rendering*, pages 71–82, June 2000.