A Vrml97-X3D Extension for Massive Scenery Management in Virtual Worlds

Jean-Eudes Marvie*
INSA of Rennes

Kadi Bouatouch†
University of Rennes

Abstract

In this paper we present a VRML97-X3D extension to describe precomputed visibility relationships in the context of progressive transmission as well as real time visualization of massive 3D sceneries. The extension we propose can be used to describe cell-to-cell, cell-to-objects as well as hybrid visibility relationships using a generic cell representation. Thanks to these relationships it is possible to represent indoor and outdoor sceneries using the same VRML97-X3D representation. We also present some new mechanisms that allow to perform instance sharing of objects described into separate files. These mechanisms allow to minimize the size of the set of VRML97-X3D files that are used for the database description as well as the amount of main memory required on the client side to visualize the scenery. In addition to these new mechanisms we also explain how we manage the memory on the client side and how we perform data pre-fetching according to the viewpoint movements. Finally we present several tests performed on different kinds of sceneries such as city and architectural models.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Remote systems; I.3.6 [Computer Graphics]: Methodology and Technics—Languages, Standards

Keywords: VRML97-X3D extension, massive sceneries, progressive transmission, visibility, data management, pre-fetching, interactivity, city models, architectural models.

1 Introduction

Thanks to VRML97 [Carey and Bell 1997], and more recently X3D [Web3D 2002], it is possible to describe very complex virtual worlds. Since the first purpose of such worlds is to be transmitted across networks, their high memory size can rapidly become a major limitation for the authors. Indeed, even when using `Inline` lazy download mechanism, `ProximitySensor` or `VisibilitySensor` nodes it is very difficult to distribute the transmission of such 3D models over the navigation time and to tune the client' main memory usage.

We propose the following classification of the different parts that make up a virtual world. In most virtual worlds only a small part of the 3D model is *globally dynamic*. That is to say, only few objects of the world, such as flying birds or cars, will move across the whole scene. Some other objects of the world, such as the doors of a building, are said to be *locally dynamic*. And finally most parts of the world, such as building walls and roofs or terrains, are *static* objects. In this paper, the part of the world made up of *static* and *locally dynamic* objects is called *scenery*.

In this classification we can notice that most of the *static* geometry is usually composed of large sets of opaque polygons. For example, in urban scenes the walls of a building generate massive occlusion when walking into a street. For indoor scenes, when walking into a room only few other rooms are still visible to the user because of the occlusion due to the walls of this room. Thus, as the geometry that induces this kind of occlusion is the *static* one, it is possible to pre-compute the subset of the *scenery* that will be potentially visible from any point of a given street or a given room. Such a set is called *potentially visible set* (PVS).

This kind of visibility relationship is, in our opinion, very interesting for web 3D applications. Indeed, if the client (i.e. the browser) knows in which street the viewpoint is located, it just needs to download the PVS of this street to construct the visual representation of the world. When the current street changes, the browser just has to download the missing geometry and to remove the geometry which is not potentially visible anymore. Thus, only the *globally dynamic* geometry has to be downloaded at the connection whereas the *scenery* downloading can be distributed over the navigation time. Using this mechanism, the browser is able to limit the size of the data to be transmitted across the network and to be rendered using the available graphics hardware. Finally, it is also able to minimize the amount of main memory (RAM) to be used during the remote navigation.

Because this mechanism introduces so many significant optimizations, we developed a VRML97 extension that allows to use such relationships to download and visualize the *scenery* of virtual worlds. The *globally dynamic* part of the world is managed using the classical VRML97 mechanisms. Since the purpose of our extension is visualization, we suppose the detection of the streets and the rooms as well as the visibility computation to be performed as a pre-processing using one of the many existing algorithms. Some of them are presented in the following section. In this paper, we will focus on the VRML97 representation and the management of such representation on the server and on the client side.

2 Related works

As we said in the introduction, the solution we propose uses the output of some algorithms that pre-compute conservative visibility relationships. A survey of visibility for walkthrough applications can be found in [Cohen-Or et al. 2002]. Among the existing algorithms, some are specially designed for architectural models [Airey et al. 1990; Teller and Séquin 1991; Teller 1992; Meneveaux et al. 1997]. In [Teller and Séquin 1991], indoor scenes are subdivided into cells using a constrained BSP [H.Fuchs et al. 1980]. The resulting cells correspond to the rooms of the scene. A visibility relationship is established between the cells using a set of portals that are extracted during the subdivision process. In [Wonka et al. 2000], which is specially designed for urban scenes, each street is a

*e-mail: jemarvie@irisa.fr
†e-mail:kadi@irisa.fr

cell and the potentially visible set of objects associated with a cell is computed using occluder fusion and is hardware accelerated. Finally, some more general solutions [Durand et al. July 2000; Schaufler et al. 2000] can be used in both cases. Many other solutions exist and most of them are described in [Cohen-Or et al. 2002]. In recent works, new solutions were presented to extract cells and portals from indoor scenes [Haumont et al. 2003] and from indoor and outdoor scenes [Lerner et al. 2003].

In most of these works, the result of the pre-computation is generally used for local visualization. Nevertheless, some optimization such as data pre-fetching or cache management techniques [Funkhouser 1996] can be easily transposed to remote environments. The difference between such systems and a web browser is the latency that is needed to fetch some data from the server. Indeed, in the local solution, the data have only to be transferred from the hard drive to the RAM. Whereas for remote systems the data have to be downloaded from a remote server, using most of time a low bandwidth connection. Some other solutions such as intermediate PVS representation [Koltun et al. 2001] or delta PVS representation [Durand et al. July 2000] can be used to minimize the memory size of the PVS descriptions. These optimizations are still interesting for remote connections since they reduce the amount of data to be transmitted through the network. Discussions about the PVS storage problem can be found in [Cohen-Or et al. 2002; Cohen-Or et al. 1998] and specially in [Cohen-Or and Zadicario 1998] that deals with visibility streaming. Finally, a VRML97 extension for the management of 3D scenes using cells and portals is proposed in [Grahn 2000]. We think this proposal to be very interesting but unfortunately limited due to the kind of visibility relationship used.

3 Basis concepts

In this section, we introduce the basis concepts and definitions of our extension. We first give the definition of a cell as well as the prototype of the node that is used to describe such a cell. We then explain how the cell descriptions should be distributed among separate files and the method we utilize to refer to these cells. Finally, we explain how we track the cell that contains the current viewpoint in order to determine the geometry that is potentially visible from this viewpoint.

3.1 Convex cell

As we said previously, we rely on algorithms that are able to compute visibility relationships for different parts of the virtual worlds such as the streets of a city or the rooms of a building. According to the nature of the algorithm, the spatial description of these places have to be done using either two dimensional polygons (called footprints) or three dimensional polyhedral volumes. In both cases, the spatial description is called a cell.

In our extension, a *cell* is a part of the whole world which is delimited by a convex volume whose boundaries are described using a set of convex polygons. A *cell* has to be convex in order to reduce the computation costs during the navigation. Such a *cell* is described using a new VRML97 built-in node named `ConvexCell`. The prototype of this new node is detailed in figure 1. The fields `coord` and `coordIndex` are used exactly the same way as for an `IndexedFaceSet` node. They are used to describe the convex hull of the cell. The polygons used for the description must be convex and have the same orientation, either clockwise or counterclockwise. This last constraint is used to determine rapidly if a point is located either inside or outside a cell. The other fields that are used will be specified in the following sections. Note that each field is of type `field`. Presently, we constrain the fields to be static because all our sceneries are pre-processed (subdivision, visibility) automatically. Therefore generating dynamic visibility relationships seems to be extremely difficult.

```
ConvexCell {
  field MFInt32  cadjIndex  []
  field MFString cellUrl    []
  field SFNode   coord      NULL
  field MFInt32  coordIndex []
  field MFInt32  cpvsIndex  []
  field MFNode   lpvs       []
  field MFNode   opvs       []
}
```

Figure 1: ConvexCell node prototype.

3.2 Linked viewpoint

During the navigation, the cell that contains the current viewpoint is used do determine the set of geometry to be downloaded and displayed. This cell will be called *current cell* for now on. In order to determine rapidly the *current cell*, we added a new field to the usual `Viewpoint` node that refers to the cell in which it is located. Thanks to this additional field, finding the *current cell* is immediate when the `Viewpoint` node gets bound. This field is named `cellUrl` and contains an extended URL pointing to the cell. The extended URL mechanism is explained in the following section. If the cell that is referred to exists and if its convex hull contains the viewpoint's position, the cell and the viewpoint are said to be *linked*.

3.3 Extended URL

Recall that we aim at distributing the downloading over the navigation time. For this reason, the cell descriptions should not be stored into a global file. In our extension, even if it is possible to do so when preparing a database for a local usage, we usually store each cell description into a separate file. Thanks to this, when the browser needs the description of a cell, it just has to download the associated file. However, it could be interesting to cluster the cell descriptions into a reduced set of files, each one containing some cell descriptions that will be needed at the same time. Doing so would reduce the amount of requests to be sent to the server and would consequently reduce the network utilization.

As the client must be able to access a given cell stored into an external file together with some other cells, we introduced the use of *extended URLs*. An *extended URL* is an URL with a syntax similar to the one used for the URLs that refers to external prototypes. The first part of an *extended URL* refers to the file that contains the cell and the second part is the name of the cell using the `DEF` mechanism. The two parts are separated using the `#` symbol. The *extended URL* `"cells_0.wrl#C2"` refers to the cell named `C2` which is defined into the file `cells_0.wrl`. In order to be accessed by the browser, the cell has to be described as a root node of its associated file. If several cells are using the same name, the last cell that is defined is used. Finally, if the *extended URL* only contains the `#` symbol as well as the second part, the referred cell is stored into the file where the *extended URL* is written.

3.4 Navigation space

As we saw in section 3.2 the current cell is determined using the extended URL that is stored in the bound `Viewpoint` node. Because the position of the viewpoint changes during the navigation, the current cell URL has to be updated whenever the viewpoint gets

into a different cell. Because we need that this update be fast, each cell contains a list of extended URLs that refers to its *adjacent cells*. The *adjacent cells* of a given cell are the cells that can be directly accessed from this cell.

Whenever the viewpoint position changes, we check if it lies in the current cell. If the new position is not inside the current cell, we check the position against each *adjacent cell* until a cell that contains the new position is found. If such a cell is found, it becomes the current cell and the `cellUrl` field of the viewpoint node is updated with the extended URL of the new cell. Otherwise, the viewpoint position is updated as if a collision with the convex hull of the current cell occurred. When using event routing to animate the viewpoint, the author must ensure that the position path passes through a set of successive *adjacent cells*. Otherwise, the animation of the viewpoint is not performed completely and the viewpoint remains blocked in the last current cell for which no *adjacent cell* can be found.

In a virtual world, the set of cells that can be accessed through a viewpoint binding or through an adjacency relation is called *navigation space*. In a virtual world where only linked viewpoints are used, the user navigation is limited to the *navigation space*. In an urban model, the *navigation space* can be made up of the street network of the city. In an indoor scene, the *navigation space* can be made up of the rooms of the building.

Finally, the collisions with the geometry are only tested against the geometry that is contained in the current and the adjacent cells. This reduction of collision tests gets significant when visualizing large sceneries.

4 Visibility relationships

Now that we have seen the basis concepts of our extension we can have a look at the different kinds of visibility relationships that can be used for scenery representation. In the two first following sections we present the cell-to-geometry as well as the cell-to-cell visibility relationships and we explain, through examples, how and when these two kinds of relationship should be used. Finally, in a third section, we explain how it is possible to merge these two kinds of relationship in order to minimize the size of the database, the amount of data to be transmitted over a network and the memory costs during the visualization.

4.1 Cell-to-geometry visibility

The *cell-to-geometry* visibility relationship is illustrated by the figure 2. With this kind of relationship, each cell contains a set of references to its potentially visible objects. For example, in an urban scenery, each street could refer to the buildings that are potentially visible. For now on, the set of potentially visible objects associated with a cell will be called *object PVS* (OPVS).

During the navigation, the OPVS of the current cell is first downloaded. It is then frustum culled before each frame construction, then the objects of the OPVS that are in the view frustum are rendered.

With this kind of relationship, one can notice a potential redundancy within the OPVSs of some cells. Imagine a public square, with a column at the middle of a urban model. Suppose the navigation space of the public square is composed of a set of cells placed around the column. In this case, the column will be potentially visible from each cell. For this reason, the description of the column must be done in a separate file that must be downloaded only once. Furthermore, the instantiation of the column model should also be done only once. And finally, each cell of the square should refer to this column as shared object. Thus, there is no more redundancy of objects description but only a redundancy of object references.

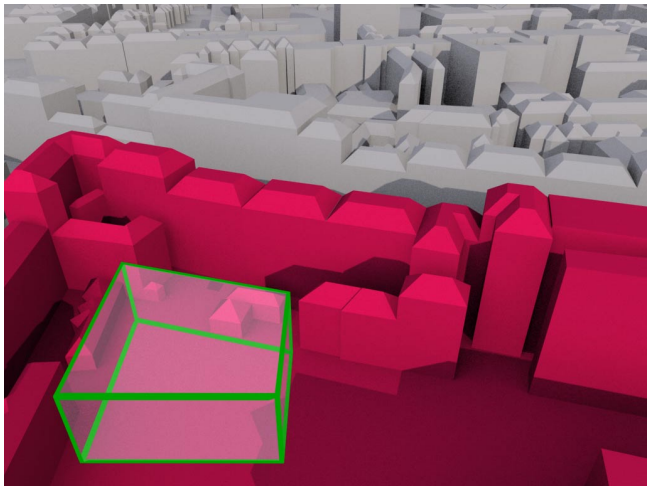

Figure 2: Cell-to-geometry visibility relationship. The outlined transparent volume is the convex hull of the cell for which the relation is depicted. The buildings and the ground that are in the foreground are the objects that are potentially visible from the cell (its OPVS). The geometry of the background is not visible from the cell.

In our extension, this kind of external reference is done using the shared inline mechanism which is detailed in section 5.1.

Note that the shared inline mechanism should be used for any object of the scenery. Thanks to this file distribution, the downloading of the scenery's objects can be distributed over the navigation time. Furthermore, the disk space required on the server side as well as the RAM and disk cache utilized by the browser are minimized.

In the extension, the OPVS references of each cell are depicted into its description. Even if the objects of the OPVS can be referred to using the shared inline mechanism, we let the possibility to create some databases for a purpose of local utilization. For this reason, the OPVS is stored into the `opvs` field of the `ConvexCell` node, which is of `MFNode` type.

When authoring a database for local usage, all the `ConvexCell` nodes can be stored into the root file and their `opvs` field can be filled with the object descriptions. Then, each object description can be shared by the cells using the classical `DEF/USE` mechanism.

For a remote usage, each cell and each object can be described into a separate file. In this case, the `opvs` field of each `ConvexCell` node is then filled using `SharedInline` nodes that implements the shared inline mechanism. Each `SharedInline` node refers to the shared objects described into a separate file. The specification of the `SharedInline` node is given in section 5.1.

4.2 Cell-to-cell visibility

The second visibility relationship is called *cell-to-cell* and is depicted in the figure 3. With this kind of relationship, each cell contains the description of its own geometry as well as a set of references to its potentially visible cells. For example, in an indoor scenery, each cell associated with a room contains the geometry of the given room and refers to a set of other cells (i.e. rooms) that are potentially visible. For now on, the set of potentially visible cells associated with a cell will be called *cell PVS* (CPVS) and the set of objects contained by a cell will be called *local PVS* (LPVS).

During the navigation, the CPVS of the current cell is first downloaded. Then, the following computations are performed for each frame construction. First, the cells of the CPVS are frustum culled using their convex hull description. Then, the objects of the LPVS

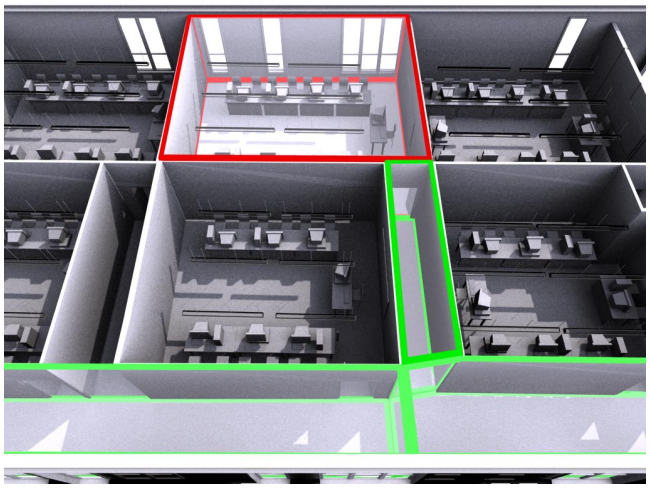

Figure 3: Cell-to-cell visibility relationship. The outlined transparent volume in the background is the convex hull of the cell C for which the relation is depicted. The outlined transparent volume in the corridors are the convex hulls of the cells that are potentially visible from the cell C (its CPVS). Each cell refers to the geometry (walls, furniture, etc.) contained in its convex hull as its local PVS (its LPVS).

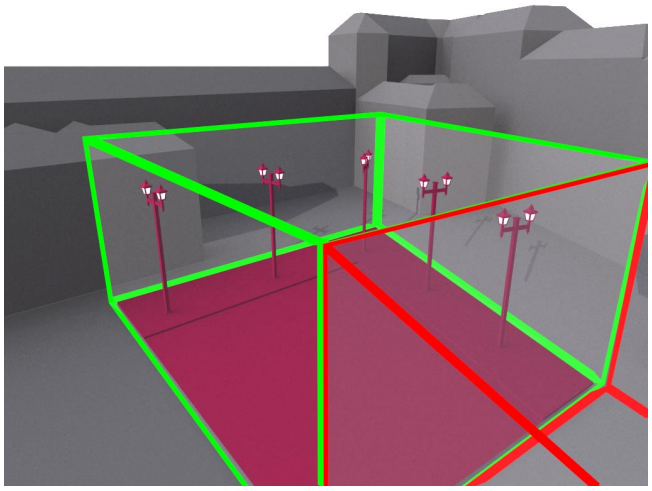

Figure 4: Hybrid visibility relationship. The outlined transparent volume in the foreground is the convex hull of the cell C for which the relation is depicted. The outlined transparent volume in the background is the convex hull of the cell C_i which composes the CPVS of the cell C. The street lamps and the pavements are in the LPVS of the cell C_i. The buildings are potentially visible from both cell and represent their OPVSs.

of the current cell as well as the objects of the LPVS of the cells that were found to be visible are then frustum culled. Finally, only the objects that are found to be in the view frustum are rendered. Because each cell of the CPVS is frustum culled before its LPVS is processed, the geometry of its LPVS has to be completely contained in its convex hull.

As we saw in the section 3.4, each cell contains a set of extended URLs that refer to its adjacent cells. The same mechanism is used to store the references to the cells of the CPVS of each cell. Note that some cells can be adjacent to a cell as well as potentially visible from this same cell. For this reason, the `ConvexCell` node contains three different fields to store the adjacent cells references as well as the cells references of the CPVS. The first field named `cellUrl` is used to store the list of all the extended URLs that are used. The second field named `cadjIndex` is a list of indices that refer to some extended URLs, stored in the `cellUrl` field, that represent the references to the adjacent cells. Finally, the field named `cpvsIndex` is a list of indices that refer to some extended URLs, stored in the `cellUrl` field, that represent the references to the cells of the CPVS.

In addition, the objects of the LPVS are described in the field `lpvs` which is of type `MFNode`. Since the LPVS of a cell is contained in the convex hull of the cell, the description of the objects that make up the LPVS can be put directly into the cell description. Nevertheless, we will see in section 5 that it is also possible to use instance sharing for some objects described into several LPVSs.

4.3 Hybrid visibility

As we can see, the two previous visibility relationships present some interesting advantages. The purpose of hybrid visibility relationship it to merge these advantages into a single representation. Let's have a look at the urban scenery depicted in Figure 4. This time, imagine that besides the roads, the street pavements and the building, we also use street lamps and some other objects that can be usually found in the streets of such a model. Thus, using the cell-to-object visibility relationship would introduce large sets of references to describe the OPVS of each cell. In contrast, since we want the navigation space to be made only of streets, it is not possible to use cell-to-cell visibility relationship because the geometry of buildings is not contained in the cells.

Thus, we propose the use of hybrid visibility to represent such a scenery. In this case, the buildings can be referred to by using the cell-to-object visibility relationship. Then, the geometry of each street can be described into the cell associated with this street. That is to say, if the convex hull of a cell contains the geometry of the street lamps, these latter can be described into the description of this cell as its LPVS. And finally, a cell-to-cell visibility relationship can be used. Thanks to this hybrid representation, if a cell contains 50 street lamps, the other cells would not have to refer to these 50 street lamps but only to the cell that contains the description of these street lamps. If we now imagine the high number of objects that can be placed into a street, we can easily imagine the gain obtained by using such a relationship. For now on, the union of the OPVS and the CPVS of a cell will be called *hybrid PVS* (HPVS). Recall that each cell can also contain a LPVS.

During the navigation, the OPVS as well as the CPVS of the current cell are first downloaded. Then, the following computations are performed for each frame construction. First, the cells of the CPVS are frustum culled using their convex hull description. Then, the objects of the LPVS of the current cell as well as the objects of the LPVS of the cells that were found to be visible are then frustum culled. At the same time, the objects of the OPVS of the current cell are also frustum culled. Finally, only the objects that are found to be in the view frustum are rendered. To sum up, the HPVS of the current cell is first downloaded and for each frame, only the objects of the HPVS that lie in the frustum are rendered.

5 Database optimizations

In this section we give some precisions about the shared inline and the shared transform mechanisms that are used to optimize the databases.

5.1 Shared inline mechanism

The *shared inline* mechanism, introduced in section 4.1, is motivated by the following problem. An object can belong to the OPVS of two different cells. The VRML97 solution that could be proposed is to use `Inline` nodes to refer to some geometry stored into a separate file. The problem is that the inline mechanism allows to refer to an extern file multiple times but produces one instance of the scene, described into the file, for each `Inline` node used. Therefore, the inline mechanism allows to download an extern file only once and to instantiate its content as many times it is used [Carey and Bell 1997, section 3.25].

In our extension, we want to refer several times to the same instance of a shared object. Consequently we introduced a new node named `SharedInline` whose prototype is exactly the same as for the VRML97 `Inline` node. With the `SharedInline` node, the referred file is downloaded once and its content is instantiated once too. Thus, if another `SharedInline` node refers to the same file, it uses the instance that was already created by the previous node.

Note that the `SharedInline` node can be used anywhere as the classical `Inline` node would be. But when using this kind of node, one should take care of the fact that only one instance of the content is referred to. Thus, it would certainly not be a good idea to refer to an extern file that contains some interactively animated objects. For example, suppose the doors of an indoor scene are animated through a *TouchSensor* node. If only one instance of that door is referred to using a `SharedInline` node, all the doors of the world would open or close when the user touches one of them.

In counterpart, we saw in section 4.1 that the static objects referred to within the OPVSs of the scenery should be stored into separate files and accessed through `SharedInline` nodes. Another interesting point of this mechanism is instance sharing among different LPVSs. In our urban scene example, the street lights might all be of the same model. Consequently, one could use only one description of the street lamp model and each cell can use `SharedInline` nodes to refer to the lamp several times within its own LPVS (Figure 5). The placement of the lamps is then performed through the use of `Transform` nodes, each one having a `SharedInline` node as child. In the same way, this mechanism can be used for indoor scenes where static objects, such as tables, are used several times in different rooms.

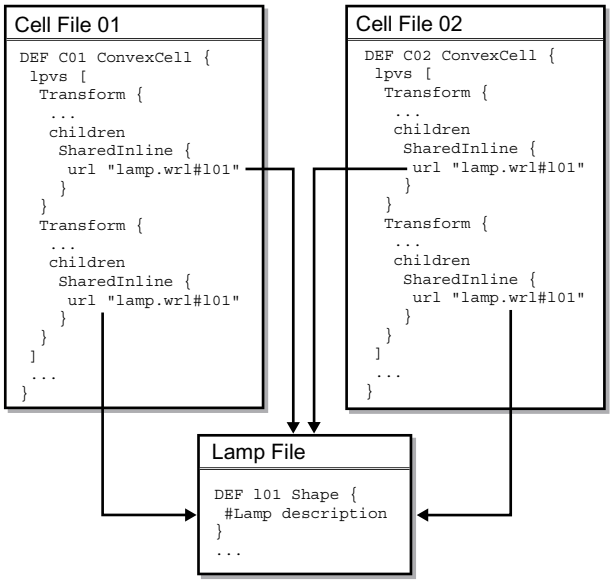

Figure 5: Instance sharing using shared inline mechanism. Each cell contains two lamps. Each lamp is described using a shared instance that is described into a separate file and referred to by SharedInline nodes. Each lamp instance is placed using a classical Transform node.

5.2 Shared transform mechanism

The *shared transform* is the last mechanism of our extension and is provided for a purpose of optimization. As it is described in section 3.1 the volume that is used to describe a cell has to be convex. Thus, the non-convex rooms have to be subdivided into convex cells. Recall that the geometry of the LPVS of a cell has to be contained in the convex hull of this cell. Consequently, if the cell-to-cell mechanism is used, which is the best solution for indoor scenes, the geometry of a non-convex room has to be cut and placed into the cells resulting from its subdivision. The amount of objects that have to be cut can be very high. For example it can be necessary to cut the geometry of the floor, the ceiling, the walls and some of the furniture that lying within the two cells. Cutting so much geometry would introduce many new vertices and polygons into the scenery, which can lead to a performance breakdown during the network transmission as well as during the rendering process. Furthermore, instance sharing cannot be utilized anymore because the objects might not be cut the same way.

To overcome this problem, the geometry of a non-convex room is not cut. Rather, each cell of the room refers to the shared geometry of the room as an OPVS, using the shared inline mechanism, and the geometry completely contained in a cell is described in its LPVS. Finally, a cell-to-cell visibility relationship is maintained. Even if this solution works properly, if a non-convex room R, that contains a shared object O, is split into two cells, C^1 and C^2, that can be seen from many other cells C_i, the reference to the object O must be stored in the OPVS of each cell C_i. Consequently, each cell C_i must refer to the object O as well as to the cells C^1 and C^2 that might contain some other objects into their LPVSs. Thus, if the scenery contains many shared objects, the amount of references to be used might increase rapidly. This overhead can be prevented by using the shared inline mechanism to make the LPVS of the two cells C^1 and C^2 share the object O. This solution is efficient in term of memory space but if the two cells are both parts of a CPVS, their respective LPVS will be displayed and the shared object O will be displayed twice.

To prevent multiple rendering of the same instance we introduced the *shared transform* mechanism. This mechanism is implemented with a new node named `SharedTransform`. This node has the same prototype as the `Transform` node. However, this node is used for rendering only once per frame construction. That is to say, if such a node is defined and used several times in the same file, only the first instance will be rendered but not the others. Furthermore, if such a node is defined into a file that is referred to by several `SharedInline` nodes, it is used only once, say for the first `SharedInline` node that is encountered during the scene graph traversal. Note that it is delicate to use this node and one should take care of the consistency between the `SharedInline` nodes when using this mechanism.

In the indoor example outlined in figure 6, a chair that is shared by two cells should be described into a separate file F_d. Then, each cell should contain a `SharedInline` node in its LPVS that refers to a shared file F_s. The file F_s contains a `SharedTransform` node whose child is a `SharedInline` node that refers to the file F_d in which the chair is described. Thanks to the intermediate file F_s, the shared model of the chair can be used in the LPVS of both cells and rendered only once per frame. Note that an intermediate file F_s has to be used for each chair model that is shared by these two cells. Finally if some other chairs are fully contained by one of these cells, they can be placed using `SharedInline` nodes that refers directly to the file F_d.

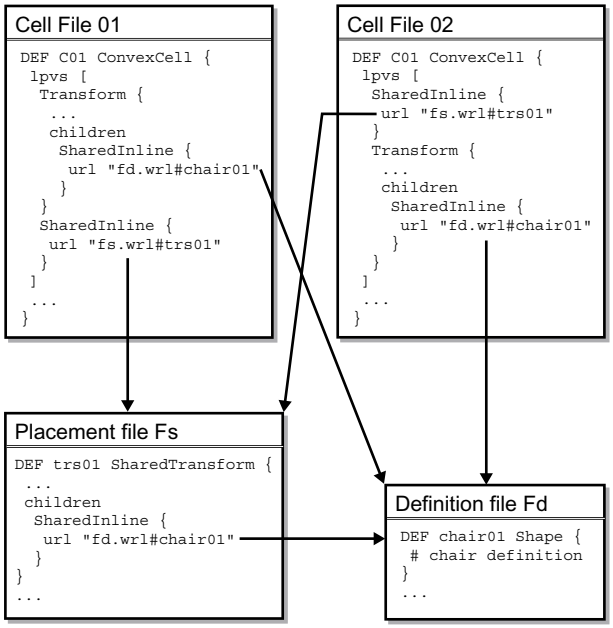

Figure 6: Instance sharing using shared inline and shared transform mechanisms. Each cell contains one instance of the chair described into the file F_d that is referred to by SharedInline nodes and placed using classical Transform nodes. Another chair instance is shared by the two cells by using the intermediate file F_s that places the chair into the world coordinate system using a SharedTransform node.

6 Database management

In the previous sections we presented solutions to describe the navigation space as well as the visibility relationships into a set of VRML97 files. Now that we have seen how to construct and optimize such a database on the server side we can have a look at the management of the data on the client side.

Thanks to the different kinds of relationship that are used in our extension, it is possible to perform data pre-fetching during the remote navigation as well as memory management. Some solutions are presented in [Funkhouser 1996] but for a purpose of local database access. The difference in our context is that the data to be pre-fetched are placed on a remote server and that the browser contains only a part of the navigation space at a given time. Indeed, since the navigation space is downloaded part by part over the navigation time, the browser can only access the cells and their relationships descriptions that have been already downloaded.

Another interesting point in our extension is the possibility to manage the client main memory (RAM). Indeed, if the client machine does not have enough memory space to store the whole scenery, it is possible to select and remove some unused cells or objects in order to download new cell or object descriptions. Some solutions are also presented in [Funkhouser 1996] but still use the whole navigation space description.

6.1 Pre-fetching

In a remote context, because downloading some data can take a long time (more than a minute), pre-fetching future required data while visualizing a part of the scenery is inevitable. In our extension, pre-fetching consists in finding and downloading the cell, as well as its associated HPVS, that will be visited after the current one. Unfortunately, this operation cannot always be done because the browser can only access the data that have been already downloaded. For this reason, the downloading operations consist of three different steps. Each of these steps can occur during navigation but only if its previous steps have been already completed. For example, the third step can be performed for ten consecutive frames but only if the first and the second steps are not required during these frames.

The first step, which is a fetching step, is performed when the navigation starts or when a different viewpoint gets bound. In this case, the current cell as well as its associated HPVS are downloaded *synchronously*. Then, once all the data are downloaded, the user can start the navigation through the current cell and visualize its associated HPVS. At this moment, only the current cell is stored in the client's memory and the browser has no information, but only the reference to the adjacent cells to perform pre-fetching.

In a second step, which is a pre-fetching step, the cells adjacent to the current cell as well as their HPVSs are downloaded asynchronously. The adjacent cells are first downloaded because they contain the references to the cells or objects that make up their own HPVSs. Since the cells and the objects of the scenery are all shared objects, their associated files are downloaded only if the browser does not already contain an instance of their content. Furthermore, once a cell is instantiated on the client side, the objects of its associated LPVS are immediately downloaded asynchronously.

Once the first adjacent cell is downloaded, the third step can start pre-fetching. This step, which is in our opinion the most important, relies on motion prediction to find the next adjacent cell that will be visited. If an adjacent cell is found to be a future visited cell, all its adjacent cells as well as their associated HPVSs are downloaded asynchronously. The algorithm in charge of the motion prediction 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. 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 selected. 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 selected and all the intersected cells are then selected as future visited cells.

6.2 Memory management

Because some sceneries can be too large to be entirely stored into the client's RAM, it is important for the browser to be able to free some memory before performing some new downloading. Indeed, because the client machine can benefit from different amounts of RAM, ranging for example from 1GB for a recent workstation to 64MB for a PDA, the memory releasing has to be performed automatically by the browser.

Before speaking about memory releasing in our extension, we should have a look at what can be removed from memory. As we said before, the shared object descriptions as well as the cell descriptions can be written into a set of files. When a cell or an object is requested, the file that contains its description is downloaded, the scene graph that is described into the file is instantiated and the requested node can be accessed by the browser through this instance. Such a scene graph will be called *shared scene graph*. Consequently, the part that can be removed from the client's memory is the instance of a *shared scene graph*. In our implementation, such an instance is assigned a reference counter that represents the number of references to this instance that are performed through the OPVS, the CPVS and the LPVS of the cells of the navigation space that are already downloaded. Such an instance is removed from

memory only if its reference counter is equal to zero. Otherwise, the reference counter is decremented and the instance is kept into the RAM because it is still referred to by some cells. For now on, this mechanism will be called *sharedRemove*. When an object is selected to be *sharedRemoved*, the reference counter of the *shared scene graph*, in which it is described, is decremented and the instance of the *shared scene graph* is removed from memory only if its reference counter is equal to zero.

In our implementation, the memory releasing mechanism is invoked when a memory allocation request fails. In this case, the releasing mechanism is asked to free a given amount of memory. The memory releasing is performed as follow. Starting from the current cell, all the cells stored on the client side are sorted arrording to their distance in the graph of adjacency. Then, the furthest cell as well as all the shared objects and cells that are referred to in its OPVS, its CPVS and its LPVS are selected to be *sharedRemoved*. This operation is repeated until the amount of requested memory is released or until the next furthest cell is one of the cells adjacent to the current cell. In this last case, the client machine does not have enough memory to visualize the database, the memory releasing fails and the scenery cannot be visualized anymore. As we can see, thanks to the adjacency relationship, the memory management is guided by the scenery topology.

In the case the client machine has a local disk with some free space at its disposal, the data can be swapped instead of completely discarded. More precisely, when a *shared scene graph* is selected to be removed, it is first swapped onto the hard drive before being effectively removed from the RAM. Because the available space on the hard drive is also limited, it is sometime necessary to remove some files in order to swap a new one. Because parts of the adjacency relationships are described into the file that are swapped, it is not possible to use the topology-based replacement anymore. Thus, in our extension, the file replacement system uses a classical LRU policy. When a shared object or a cell is requested, the file in which it is described is first searched into the disk cache. If the file is not found, a downloading request is sent to the server. Otherwise the local file is used to instantiate its *shared scene graph* and is removed from the disk cache.

In our implementation, the memory management as well as the use of disk swap can be set active, inactive or automatic through the interface of the browser. In the automatic mode, the user lets the browser choose the best solution according to the client machine capacities.

Note that when using memory management together with sceneries that contain locally dynamic objects, one should only use dynamic geometry whose state is not important (for example a door for which the author does not care if it is open or closed). Indeed, if some locally dynamic objects are removed from memory and downloaded again during the navigation, the state of the object before being removed and after being downloaded again might not be the same. In the case where the state is important, locally dynamic objects should be handled as globally dynamic objects. Another solution would be to keep these objects into the scenery, to ensure that the client machine has enough memory to store the whole scenery and to inactivate the memory management. This should only be done for certain targeted applications because in this case the database is not scalable anymore.

7 Results and future works

In this section we describe our two recent solutions that make an intensive use of our extension. We explain the structure and the mechanisms we used to represent the scenery for each solution and we present some tests concerning the progressive transmission of these sceneries across low bandwidth networks. At the moment these solutions were published the extension was not completely specified. Therefore some differences with the current specification can be found in [Marvie et al. 2003b; Marvie and Bouatouch 2003; Marvie et al. 2003a]. Nevertheless, we are now using the current specification in both solutions and we are currently enhancing our preprocessing algorithms in order to produce databases that make use of the optimization mechanisms (see section 5) that have since been implemented and validated.

7.1 Architectural sceneries

In [Marvie and Bouatouch 2003], we use the extension to visualize remote architectural scenes that contain many high resolution texture maps. In this solution, we use a space subdivision technique similar to Teller's one [Teller and Séquin 1991] for the scenery preprocessing. The result of the visibility pre-processing is then encoded using the cell-to-cell mechanism. Each cell is encoded into one file and its local geometry is encoded in the same file and described in the field `lpvs` of the cell. During the course of the subdivision the geometry that is lying in two cells is cut and the two resulting parts are placed into their respective cells.

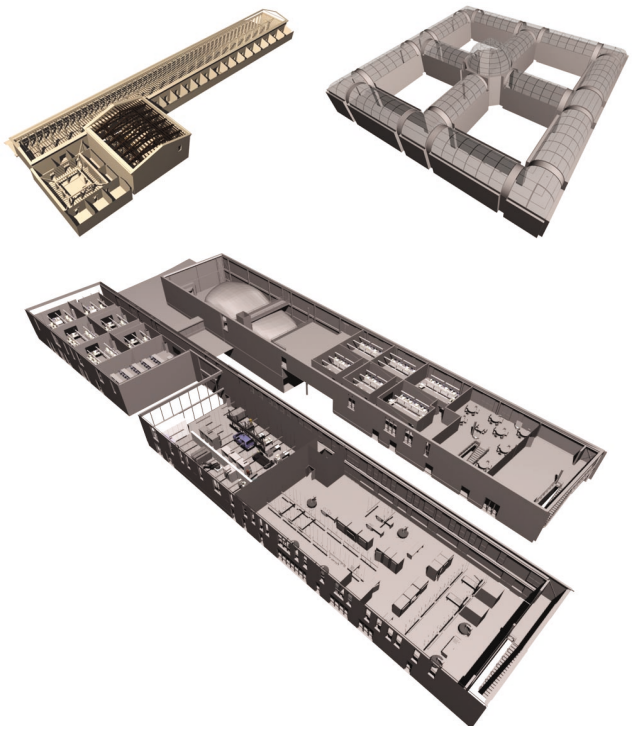

Figure 7: Top left: the Greek temple model (75000 polygons). Top right: the museum model (17000 polygons). Bottom: the Kerlan university (370000 polygons).

7.1.1 Progressive downloading

In order to analyze the downloading improvement induced by our extension we have performed the pre-processing described above on three different models : a museum model, a model of the university of Kerlan and a model of a Greek temple (Figure 7). We then performed a walk through each model. For the tests, we did not use any texture maps. Nevertheless, one can notice that our extension implicitly prevents the browser from downloading the texture maps that are not in the current PVS. Therefore, the texture map files are progressively downloaded in the same way as the cells and the geometry.

For each walkthrough we used a simulated bandwidth of 128Kb/s and a navigation speed of 3Km/h. During each walkthrough we compute, at each frame, the percentage of objects of the current PVS that are already downloaded. This value is called downloading quality. Table 1 sums up the following information for each walkthrough when using or not pre-fetching: the theoretical time required to download the original file (original time), the downloading time for the first cell and its PVS (convergence time), the minimum and average downloading qualities obtained during the walkthrough after convergence.

Model name	Orig. time	Conv. time	Pre-fetching	Min. quality	Avg. quality
Museum	9s	14.0s	disabled	74%	98.6%
			enabled	96%	$> 99.9\%$
Temple	57s	26.9s	disabled	36%	78.6%
			enabled	72%	94.1%
Kerlan	5m30s	45.5s	disabled	38%	80.0%
			enabled	61%	89.8%

Table 1: Downloading quality study, for three indoor sceneries using or not pre-fetching.

As we can see in this table, the extension is not useful for the transmission of the museum model. Indeed, this model is originally very small and the cell from which the navigation starts presents a very large PVS (nearly the full scenery). Therefore, the size of the model is enlarged with the sets of PVS references, which increases the convergence time. Note that if we have used texture maps, the convergence time would have been lower than the original time because the downloading of some texture maps would have been delayed. With regard to the two other models, the extension allows to start the navigation earlier than when using the original model. Furthermore, when using pre-fetching, the progressive transmission gives more than 89% of quality most of time. This value can easily be raised if we do not transfer the vertex normals and if we use the enhancements proposed in the following section. Indeed the two scenes contain a lot of similar objects such as chairs and tables in the Kerlan university or columns in the Greek temple.

7.1.2 Enhancement

In this solution, we did not use the *shared inline* mechanism to make the cells share the objects instances (such as the tables and the chairs). In order to enhance our existing solution, we are currently developing an automatic preprocessing step whose purpose is to extract the static DEF/USE nodes from the original scenery. Each extracted object is then described into a separate file and instantiated into each generated cell that contains an instance of this object in the original scenery. If the bounding box of the object is completely contained in the convex hull of the cell, the object is referred to using the *shared inline* mechanism described in section 5.1. Otherwise, if the object is shared by two different cells it is referred to using the *shared transform* mechanism described in section 5.2.

In addition to this optimization, we are also currently modifying our space subdivision algorithm so that the walls, the floors and the ceilings be shared by the cells that describe the volume of a non-convex room as explained in section 5.2.

7.2 Urban sceneries

In [Marvie et al. 2003b; Marvie et al. 2003a], we use the extension to visualize remote city models whose buildings are described and transmitted using procedural models. In this solution, the navigation space is restricted to the street network. Each street that is placed between two crossroads is subdivided into three cells and each crossroad is defined using a single cell. Each building, road or pavement is described as an object and a cell-to-object visibility relationship is established between these objects and each cell.

Figure 8: Bird eye view of the city model (1000000 polygons).

The model we used for the tests (Figure 8) is automatically generated using the city model generator described in [Marvie et al. 2003b]. In this report we emphasize the fact that in a city, many buildings are similar in style and shape. Therefore we propose the use of procedural models to describe each building. Each building style is described in an external prototype using an L-system based scripting language. Each building is then described into its associated VRML97 file by including the external prototype that encodes its style and an instance of the building is created by assigning the parameters, associated with this building, to the selected prototype. For example, the parameters can be the footprint of the building, its number of floors and the height of its adjacent buildings to correctly generate the party walls. With this method, we only need to transmit a very low amount of data (300Bytes) to describe a complex building. Furthermore, the visibility relationship that is used allows the progressive transmission of these low cost building descriptions. Finally, in order to accelerate the rendering process, the geometry of each building is generated using LODs whose levels are automatically selected according to the visual importance of the building as well as to the client machine performances.

7.2.1 Progressive downloading

Similarly to section 7.1.1 we performed a walk through a 700m square city model (Figure 8). The navigation speed was set to 15Km/h and the network bandwidth was limited to 56Kb/s. In this model the texture maps were encoded using non compressed TGA files. Thanks to the use of procedural models, the database size is equal to 541.9KB in a compressed format instead of 1.09GB when the geometry is completely reconstructed. During the walkthrough we compute, at each frame, the percentage of objects of the current PVS that are already downloaded and the percentage of objects of the current PVS for which geometry is reconstructed. These values are called downloading quality and rewriting quality. If both values are equal to 100%, the transmission quality is perfect for the new frame. Figure 9 shows how these values evolve over time for each walkthrough, when using or not pre-fetching.

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 used in our test scene. Nevertheless, downloading is low thanks to our procedural models and our compressed binary format. In addition, downloading is homogeneously distributed over time owing to the spatial subdivision and visibility computation results. Finally, the downloading and rewriting quality

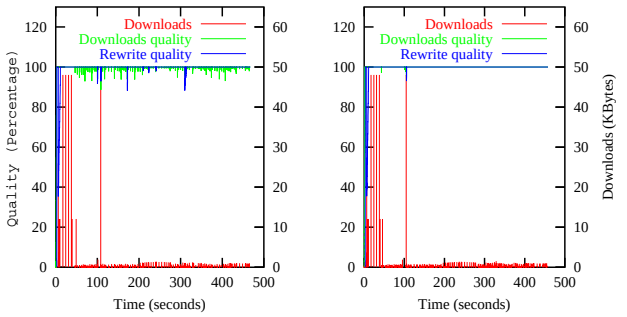

Figure 9: Left: downloading over time (expressed in KB), downloading quality and rewriting quality for the city walkthrough, without using pre-fetching. Right: same using pre-fetching.

plots show that pre-fetching allows to obtain a perfect transmission quality (100%) most of the time.

7.2.2 Enhancement

In this solution we only used the cell-to-object visibility relationship. Since our city models as well as the navigation space are automatically generated, we are currently enhancing our generator in order to use the hybrid visibility relationship. With the new generation process, the objects such as the street lamps will be encoded into their associated cells using the cell-to-cell mechanism. Furthermore, since many objects placed in the streets are similar it will be possible to use the shared inline and the shared transform mechanisms to refer to these objects as explained in section 5. We believe that we will soon be able to generate city models with a huge number of objects into their streets to increase their visual realism. Thanks to these mechanisms we will still be able to transmit and visualize the models using low bandwidth networks and clients with low performances. In addition, we intend to visualize these models using wireless connected PDAs.

8 Conclusion

In this paper we have presented a simple and efficient VRML97 extension that allows the progressive transmission and the interactive visualization of massive sceneries. Thanks to the three kinds of visibility relationships that are proposed, it is possible to generate databases for indoor and outdoor sceneries in a simple and compact way. Thanks to the general representation of the cell it could also be possible to merge indoor and outdoor sceneries in the same virtual world.

The shared inline and the shared transform mechanisms we introduced are new tools to share some objects located in separate files. Thanks to these mechanisms the author can minimize the data redundancy within the database and produce databases that are efficient in terms of network transmission and client RAM occupation.

Finally we believe that the cell-to-cell visibility relationship can also be used to speed up the rendering of globally dynamic objects. Indeed, if globally dynamic objects remain within the navigation space, it is possible to check if such objects lie in some of the cells that make up the current CPVS. Thus, if the object is in one of these cells, it needs to be rendered otherwise it is not visible. We are currently making some experiments with this case and we are looking for a solution to track the globally dynamic object in an efficient way. In our opinion, thanks to this optimization we will soon be able to visualize virtual worlds populated with a very large number of animated humanoids and vehicles.

References

AIREY, J. M., ROHLF, J. H., AND BROOK, F. P. 1990. Toward image realism with interactive update rates in complex virtual building environements. In *Symposium on interactive 3D graphics*, 41–50.

CAREY, R., AND BELL, G. 1997. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley Developers Press.

COHEN-OR, D., AND ZADICARIO, E. 1998. Visibility streaming for network-based walkthroughs. In *Graphics Interface*, 1–7.

COHEN-OR, D., FIBISH, G., HALPERIN, D., AND ZADICHARIO, E. 1998. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. In *Computer Graphics Forum*, vol. 17(3), 243–253.

COHEN-OR, D., CHRYSATHOU, Y., SILVA, C. T., AND DURAND, F. 2002. A survey of visibility for walkthrough applications. *TVCD*.

DURAND, F., DRETTAKIS, G., THOLLOT, J., AND PUECH, C. July 2000. Conservative visibility preprocessing using extended projections. *Proceedings of SIGGRAPH 2000*. Held in New Orleans, Louisiana.

FUNKHOUSER, T. 1996. Database management for interactive display of large architectural models. In *Proceedings of Graphics Interface*, 1–8.

GRAHN, H., 2000. Cells and Portal in VRML a proposal. Blaxxun interactive.

HAUMONT, D., DEBEIR, O., AND SILLON, F. 2003. Volumetric cell-and-portal generation. In *Eurographics*.

H.FUCHS, Z.KEDEM, AND NAYLOR, B. 1980. On visible surface generation by a priory tree structures. In *Computer Graphics*, 14(3):124–133.

KOLTUN, V., CHRYSANTHOU, Y., AND COHEN-OR, D. 2001. Virtual occluders : an efficient intermediate pvs representation. In *Rendering Techniques 2000 : 12th Eurographics Workshop on Rendering*.

LERNER, A., CHRYSANTHOU, Y., AND COHEN-OR, D. 2003. Breaking the walls: Scene partitioning and portal creation. In *Pacific Graphics*.

MARVIE, J.-E., AND BOUATOUCH, K. 2003. Remote rendering of massively textured 3D scenes through progressive texture maps. In *The 3rd IASTED conference on Visualisation, Imaging and Image Processing*, ACTA Press, vol. 2, 756–761.

MARVIE, J.-E., PERRET, J., AND BOUATOUCH, K. 2003. Remote interactive walkthrough of city models. In *proceedings of Pacific Graphics*, IEEE Computer Society, vol. 2, 389–393. Short Paper.

MARVIE, J.-E., PERRET, J., AND BOUATOUCH, K. 2003. Remote interactive walkthrough of city models using procedural geometry. Tech. Rep. PI-1546, IRISA, July. http://www.irisa.fr/bibli/publi/pi/2003/1546/1546.html.

MENEVEAUX, D., MAISEL, E., AND BOUATOUCH, K. 1997. A new partitioning method for architectural environments. *Journal of Visualization and Computer Animation* (May).

SCHAUFLER, G., DORSEY, J., DECORET, X., AND SILLION, F. X. 2000. Conservative volumetric visibility with occluder fusion. In *Siggraph 2000, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., Annual Conference Series, 229–238.

TELLER, S., AND SÉQUIN, C. H. 1991. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, 61–69.

TELLER, S. 1992. *Visibility Computations in Densely Occluded Environments*. PhD thesis, University of California, Berkeley.

WEB3D. 2002. Extensible 3d (X3d). Specification, Web 3D Consortium, http://www.web3d.org/fs_specifications.html.

WONKA, P., WIMMER, M., AND SCHMALSTIEG, D. 2000. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Eurographics Workshop on Rendering*, 71–82.