Interactive Walkthrough of Large 3D Models of Buildings on Mobile Devices

Alessandro Mulloni*
HCI Lab
Dept. of Math and Computer Science
University of Udine
via delle Scienze, 206
33100 Udine, Italy

Daniele Nadalutti†
HCI Lab
Dept. of Math and Computer Science
University of Udine
via delle Scienze, 206
33100 Udine, Italy

Luca Chittaro‡
HCI Lab
Dept. of Math and Computer Science
University of Udine
via delle Scienze, 206
33100 Udine, Italy

Abstract

Interactive visualization of large 3D architectural models on mobile devices such as PDAs would significantly benefit applications such as indoor navigators and mobile tourist guides, on-site monitoring and annotation of architectural designs at construction sites, evacuation training and evacuation guidance.
Although PDAs are becoming more powerful and a few are even equipped with 3D hardware accelerators, their performance does not yet allow to handle a large architectural model at an acceptable frame rate. To face this problem, we propose and experiment a system that exploits hierarchical view frustum culling and portal culling for interactively visualizing 3D architectural models on mobile devices. We also discuss the performance of the system and its integration with our mobile X3D player (MobiX3D). The performance of the system has been evaluated on a large three-floor building with 39 rooms, 42 stairs and 42 doors.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality; I.3.8 [Computer Graphics]: Applications

Keywords: mobile devices, architectural models, X3D, rendering, culling

1 Introduction

Interactive visualization of large 3D architectural models on mobile devices such as PDAs would significantly benefit applications such as indoor navigators and mobile tourist guides, on-site monitoring and annotation of architectural designs at construction sites, evacuation training and evacuation guidance. The visualization can also be location-aware, updating itself and providing information based on the users' position in the physical world.

The difficulty in rendering large 3D building models on mobile devices is related to their complexity in terms of polygons and in terms of topology. However, the topological structure of buildings can be exploited to improve rendering speed: only the room where the user currently is and what the user sees from its doors and windows need to be processed. Portal culling algorithms [Airey 1990; Teller and Séquin 1991; Luebke and Georges 1995] exploit this idea to speed up the rendering of 3D buildings.

Although PDAs are becoming more powerful and a few are even equipped with 3D hardware accelerators, their performance does not yet allow to handle a large architectural model at an acceptable frame rate. Mobile devices are indeed characterized by some serious limitations with respect to desktop systems:

- **limited CPU and memory**: PDAs use limited-power processors and have small RAM memories (e.g. 64 or 128 MB) that limit the size of the 3D model that can be loaded in the main memory.
- **limited bus speed**: the latest PDA internal and external buses have speeds that limit the dynamic loading and flushing of data from storage memory, needed for displaying large 3D models.
- **absence or limited performance of graphics accelerators**: most mobile devices on the market do not have a 3D graphics accelerator. A 3D graphics accelerator would also save CPU power and bus bandwidth while displaying large 3D models.
- **low resolution of the screen**: the latest PDAs support VGA resolution (480x640), but most PDAs are limited to lower resolutions such as 240x320. This limitation affects the quality of displayed images on mobile devices.
- **absence or limited performance of FPUs**: some mobile devices on the market are not equipped with a FPU, limiting the speed of floating-point operations. This limitation is usually tackled by using fixed-point arithmetics that is obviously more efficient, but also introduces a loss of accuracy.
- **energy consumption issues**: battery consumption is a critical factor in mobile devices. PDAs typically reduce clock frequency to save energy when the battery is low on power.

This paper describes the approach we followed for interactively visualizing large 3D buildings on mobile devices. We exploit portal culling as well as view frustum culling (polygons that are out of an approximation of the user's field of view do not need to be processed). The culling algorithm we implemented is conservative: it does not cull polygons that are visible, but it can sometimes process polygons that are not visible. To load and render X3D files, our system uses the MobiX3D player [Nadalutti et al. 2006] (freely

*e-mail: a.mulloni@gmail.com
†e-mail: nadalutti@dimi.uniud.it
‡e-mail: chittaro@dimi.uniud.it

available for download at [HCI Lab – University of Udine 2006]). MobiX3D was intended for playing generic X3D content on Pocket PCs, and the interactive rendering of a whole model of a large building could be impossible because of memory limitations and computational constraints of the target devices. To allow MobiX3D to interactively display X3D models of large buildings, it was necessary to come up with a dynamic method for:

- minimizing the amount of hidden polygons to be processed;
- handling the dynamic loading and flushing of data from the storage memory of the device in an efficient way, without interrupting the virtual exploration of the 3D model.

The paper is organized as follows. Section 2 surveys related work on culling algorithms and on systems for interactive visualization of architectural models. Section 3 describes in detail the algorithm we implemented to handle large 3D architectural models on mobile devices. Section 4 discusses implementation details. Section 5 analyzes system performance. Finally, Section 6 provides conclusions and outlines future research directions.

2 Related work

2.1 Culling algorithms

When handling large models, visibility culling can be used to perform a selective removal of part of the scene before it is sent through the graphics pipeline, thus reducing the number of polygons to be processed. Culling is based on computing visibility for the objects in the scene (i.e., determining whether a viewer can see an object from a given viewpoint) and can be done totally on-the-fly or it can exploit pre-computed data structures: the more static is the scene, the larger is the number of polygons whose visibility can be computed off-line.

An ideal visibility culling algorithm would send only the Exact Visible Set (EVS) of polygons through the pipeline. Unfortunately, the complexity for creating an EVS is too high and most visibility culling algorithms thus use a so-called Potential Visible Set (PVS). A PVS that fully includes the EVS is called *conservative PVS*. Conservative PVS are often considered more useful than non-conservative PVS because all visible polygons are rendered.

The first classical strategies of *visibility culling* are *back-face* and *view-frustum culling*. Back-face culling algorithms [Kumar et al. 1996] avoid the rendering of polygons that face away from the viewer, while view-frustum culling algorithms avoid processing polygons that are out of the *view volume* (i.e., a representation of the user's field of view); the most used volumes are boxes and frustums. Considering the view volume used to render a given frame, only the polygons falling inside such volume will eventually be rendered in the final frame. It is therefore possible to cull polygons laying outside of the view volume early on, before further processing takes place. Furthermore, it is also possible to approximate objects with their *bounding volume* to speed up the culling process. *Hierarchical view frustum culling* is based on this idea. A whole bounding volume hierarchy is built bottom-up starting from the bounding volume of each object and following the already available hierarchy of the scene. The bounding volume hierarchy is then compared with the view volume and nodes whose bounding volumes fall completely outside the view volume are culled together with their child nodes. Hierarchical view frustum culling was introduced in [Clark 1976] and is now a well-known method in 3D graphics [Slater and Chrysanthou 1997; Assarsson and Möller 2000].

While hierarchical view frustum culling helps reducing the processed polygons, there are certain situations where the produced PVS fits too loosely the actual EVS. One such situation is given by scenes with lots of occlusions, where many objects falling within the view volume are not really visible because they are occluded by other objects. Z-buffer implements this occlusion check, but in a late stage when polygons have already been partially processed. *Occlusion culling* methods exploit occlusions by performing an early calculation of the occluded objects and by culling them before they are processed. Architectural models lend themselves to occlusion culling, because the walls of a building are often good occluders.

Following the taxonomy of [Cohen-Or et al. 2003], occlusion culling methods fall in two categories: *point-based methods* and *from-region methods*. Point-based methods compute the visible set with respect to the location of the current viewpoint only; from-region methods compute a visible set that is valid in a given region of space. From-region methods have two main advantages in interactive walkthrough rendering: (i) the visible set calculated by these methods is generally valid for more than one frame also when the viewer moves, so their computational cost can be spread over time, and (ii) from-region methods have predictive capabilities: one can pre-fetch the adjacent regions to improve rendering performance.

From-region methods can be classified in two categories: (i) portal culling algorithms, and (ii) algorithms for generic scenes. Portal culling algorithms exploit the characteristic features of architectural interiors (their subdivision in floors and rooms) to optimize the rendering process. Portal culling was first introduced in [Airey 1990]. [Teller and Séquin 1991] worked on a more efficient and complex version of the original portal culling method. A simple method to implement real-time portal culling without too much pre-processing was proposed by Luebke and Georges [Luebke and Georges 1995].

Portal culling works on a representation of the scene made of *cells* (i.e., parts of the scene delimited by specific boundaries) connected to each other by *portals* (i.e., openings on the boundaries of cells). The fundamental idea is that the viewer is inside a cell, and objects belonging to other cells can only be seen through portals; culling can therefore be applied to all objects falling outside the portal areas. The cell structure can be organized to preserve the semantics of a building by representing each room as a cell and each door and window as a portal.

Cells and portals can be defined manually, but there are algorithms that automatically process the scene and create such data structures. Recent results are reported in [Haumont et al. 2003] for indoor scenes and in [Lerner et al. 2003; Lerner et al. 2006] for both outdoor and indoor scenes.

In architectural models subdivided into separate cells (or rooms), there are two types of visibility. Following [Marvie and Bouatouch 2004], they can be called *cell-to-geometry visibility* and *cell-to-cell visibility*. Cell-to-geometry visibility is the relation that binds each single cell to the objects it contains, i.e. it binds the room of a building to the set of objects it contains. Cell-to-cell visibility is the adjacency relation that binds cells connected to each other by a portal.

2.2 Systems for interactive visualization of large architectural models

2.2.1 Desktop systems

One of the first desktop systems for interactive visualization of large architectural models is described in [Funkhouser et al. 1992]. That system employs a hierarchical database containing objects at different levels of detail, cell-to-cell and cell-to-object visibility, a real-

time memory management algorithm for swapping objects in and out of memory as the observer moves through the model, and a real-time refresh algorithm for choosing which objects to render at which levels of detail in each frame. The system was used to interactively display a floor of the planned Computer Science building at the Berkeley University of California. Considering that current mobile developers have to face in a new context the problems that desktop developers faced twenty years ago, the ideas described in [Funkhouser et al. 1992] can be of interest.

Aila and Miettinen [Aila and Miettinen 2004] proposed an occlusion culling system, called dPVS, for massive dynamic environments. The dPVS system is intended for desktop platforms, but most of the ideas in [Aila and Miettinen 2004] can be also used in the mobile context. The main contribution of Aila and Miettinen is the definition of a framework that integrates different culling methods (such as view frustum culling, portal culling, occlusion culling algorithms based on visibility database) into a system that efficiently renders massive dynamic environments. The dPVS system uses a hierarchical spatial database to store the position of every object in the scene and to efficiently update the position of dynamic objects. The system has been tested with both outdoor and indoor complex scenes, achieving acceptable frame rates.

Marvie and Bouatouch [Marvie and Bouatouch 2004] proposed a VRML97-X3D extension to support portal culling. Their proposal is based on hybrid visibility relationships, exploiting both the cell-to-cell and cell-to-geometry visibility. Their focus is on the optimization of network bandwidth and rendering in client-server architectures, but their solution can be also used on a mobile device.

2.2.2 Mobile systems

The first attempts at interactively rendering indoor models on mobile devices come from videogames. In particular, some game engines, such as Quake III, have been ported to the Windows Mobile platform. There are two mobile implementations of the Quake III rendering engine: (i) Quake Mobile [Pulse Interactive 2005], and (ii) Quake 3 Arena CE [NoctemWare 2005]. These games exploit 3D hardware acceleration of recent mobile devices and allow the user to play at an interactive frame rate. The main limitation of mobile implementations of Quake III rendering engine (and of game engines in general) is the scarce flexibility: they are designed to support only the styles of interaction and the navigation modes that are needed by the game.

In the literature, Lipman's work [Lipman 2004] for visualizing steel structures on mobile devices and Nurminen's system [Nurminen 2006] for rendering 3D city models on mobile devices can be interesting for our work.

Lipman [Lipman 2004] uses PDAs to visualize VRML models of steel structures. These structures can be modeled by using CAD environments and then exported to VRML. Lipman's application allows the user to display and interactively navigate around little steel structure models. However, with large models the user cannot interactively navigate. The only way to move around in the VRML model is with predefined viewpoints. Lipman's focus is only on efficiently organizing the VRML files that contain the steel structure to display using the Pocket Cortona VRML browser [ParallelGraphics 2004] without any optimization of rendering algorithms.

Nurminen [Nurminen 2006] proposed m-LOMA, a client-server system that uses a mobile 3D city map to give location-based information in cities. A combination of server pre-processing, suitable modeling methodologies and real-time rendering optimizations allows the m-LOMA client to render 3D city models augmented with location-based information on smartphones and PDAs. The m-LOMA server uses pre-processed PVS for static structures, but not portals because urban scenes cannot be bundled into cells as easily as indoor scenes. Moreover, for dynamic entities, the m-LOMA server applies real-time virtual voxel based culling, which in essence is an implementation of a cell-to-cell visibility scheme. The m-LOMA client implements a real-time view frustum culling algorithm.

3 Proposed solution

3.1 System overview

Our solution for rendering large X3D building models on mobile devices is based on both cell-to-cell and cell-to-geometry visibility, exploiting a double level of culling to reduce the quantity of data to be held in the main memory of the device and the amount of polygons that need to be processed.

To apply a portal culling method, one has first to identify rooms and connections among rooms inside the architectural model. Thus, the model is physically subdivided into cells and portals, representing respectively rooms and connections. This can be done manually, maintaining the topological structure of the scene by having each room contained in a single cell and each cell containing exactly one room. An automatic method for scene subdivision can be used as well, provided that its output is a set of cells connected to each other by portals.

If the input model is a X3D model, subdivision into cells and portals can be carried out in two ways: (i) by including metadata to identify cells and portals in the X3D file, or (ii) by associating each cell with a distinct X3D file that contains the whole geometry of that cell and having an external file to list cells and portals and their boundaries. The first approach strictly follows the X3D standard and does not rely on external files, but forces one to load in memory and parse the whole X3D file to get the boundaries of all cells and portals of the scene and to build a visibility graph for the scene. This wastes precious memory space and CPU bandwidth and, considering the limitations of mobile devices, is not viable for large models. This problem can be solved by pre-processing the X3D file on a workstation, generating a compact list of cells and portals in an external file, and copying it on mobile device storage, but the solution does not follow anymore the X3D standard. Moreover, metadata about cells and portals in the X3D file become unnecessary after the pre-processing stage.

The second approach does not strictly follow the X3D standard and needs more work in case of changes in the displayed model (one has to modify one or more X3D files and the external file), but overcomes the disadvantage of the first without the need for a pre-processing stage, and allows one to include only the needed information in the X3D files. Moreover, there is no need for a client-server architecture and each employed X3D file defines the part of the cell-to-geometry relationship concerning the cell it represents. Marvie and Bouatouch [Marvie and Bouatouch 2004] proposed a solution that follows this approach because it stores each cell in a distinct file, but is slightly different because it defines new X3D nodes for cells and portals and encodes visibility relations in the X3D files where cells are described. There is no external file that contains only cells and their boundaries, so it is necessary to load the whole cell to know its properties.

Figure 1 illustrates the high-level architecture of our solution for rendering large X3D buildings on mobile devices. We follow the second above mentioned approach for describing cells and portals: each cell is stored in a separate X3D file and an external XML file, called *topological file*, is used to store the list of all cells and por-

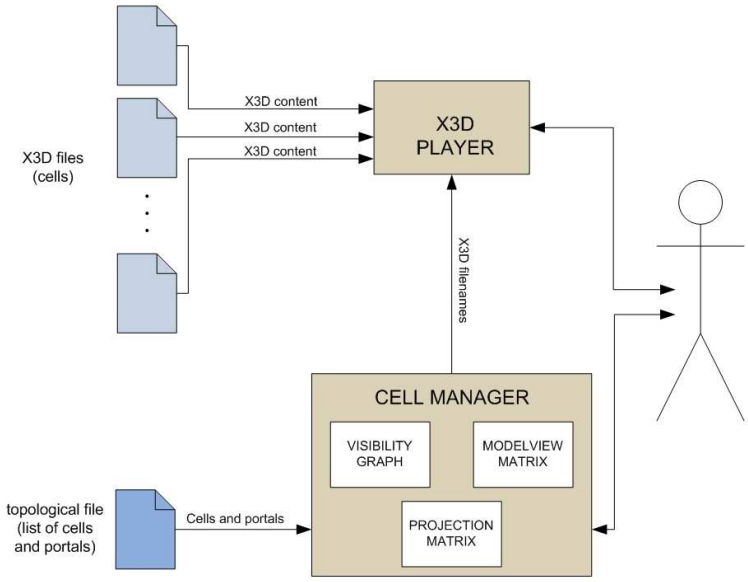

Figure 1: Architecture of the proposed solution

tals with their boundaries. The cells described by the X3D files are not overlapping to simplify inclusion tests. This simplification does not lead to less expressive power: all buildings can be represented because convex rooms can be described by single cells and non-convex rooms can be subdivided into sets of convex cells. Moreover, there is not a complexity growth of the scene in terms of polygons. In the topological file, cells are identified by X3D filenames, together with the specifications of their boundaries (stored as a set of planes defining a convex volume) representing the extension limits for the cells and the number of triangles of the cell; a portal is defined as an arbitrarily shaped polygon and the pair of rooms it connects. All the cell-to-cell visibility relationships are stored as pairs of cells connected by a portal inside the topological file; this allows us to build a visibility graph for the scene without loading any X3D file from storage memory. We chose the X3D format and not a compressed binary format for representation of buildings for two reasons: (i) prototyping and design of buildings is easier also for people who are not very skilled with 3D rendering (the most popular CAD and 3D modeling systems can export to VRML or X3D), and (ii) cells can be easily exchanged between different users.

The system we implemented is composed by two main modules: the *X3D player* and the *Cell Manager*. The X3D player loads in memory, flushes from it and renders one or more X3D files according to the commands received from the Cell Manager. The X3D player we use is the latest version of MobiX3D, a system that we originally proposed in [Nadalutti et al. 2006] and was later refined [HCI Lab – University of Udine 2006] with a basic view frustum culling algorithm. In this case, the view frustum culling of MobiX3D will be applied after the portal culling algorithm implemented by the Cell Manager. The user can navigate through the scene by using the interface of MobiX3D, described in [Nadalutti et al. 2006]. Three basic navigation modes are implemented (pan, walk and examine) and collision detection is not supported.
The Cell Manager is the module that manages the rendering of the cells. It initially builds the visibility graph of the scene starting from the topological file, then it determines the cell where the viewer is (*viewer's cell*) and the other cells that are visible from the viewer's position (*visible cells*). It indicates to the X3D player which X3D files to render, load or flush, according to the viewer's movements. Cells are dynamically loaded into main memory when they are visible and flushed from it when they have not been visible for a specified amount of time. Cells are not too large (each cell can be contained in the main memory) and if the size of visible cells is more than the main memory, the most distant visible cells are culled by using fog. This allows for the handling of massive models that could not possibly fit entirely into the main memory of the device. A multi-threaded approach was adopted to load cells so that loading is not interrupting.

To determine viewer's cell and visible cells, the Cell Manager uses:

- the visibility graph of the scene;
- the *modelview* and the *projection* matrices [Shreiner et al. 2005];
- the exact position of the viewer;
- the boundaries of all the cells in the model.

The viewer's cell is determined by a simple linear check of all the cells, but it can be optimized by exploiting more complex space division data structures. To check whether the viewer is inside or outside the boundaries of a cell, the Cell Manager checks her position with every plane defining the convex volume of the cell. To compute visible cells, a portal culling algorithm similar to the one of [Luebke and Georges 1995] is adopted. That algorithm exploits only cell-to-cell visibility, while our algorithm exploits also the distance of the cells from the viewer and cell-to-geometry visibility. In the following, we describe in detail our portal culling algorithm.

3.2 Portal culling algorithm

As in [Luebke and Georges 1995], our algorithm starts from the viewer's cell, considers all of its portals and checks whether they are visible or not: if a portal is visible, the cell connected to the viewer's cell by the portal will be visible as well. The same portal check method can be recursively applied to the adjacent cells connected by a visible portal.

More formally, we can define the *visible screen area* (VSA) of a cell as the part of the viewport where the rendered cell can be seen,

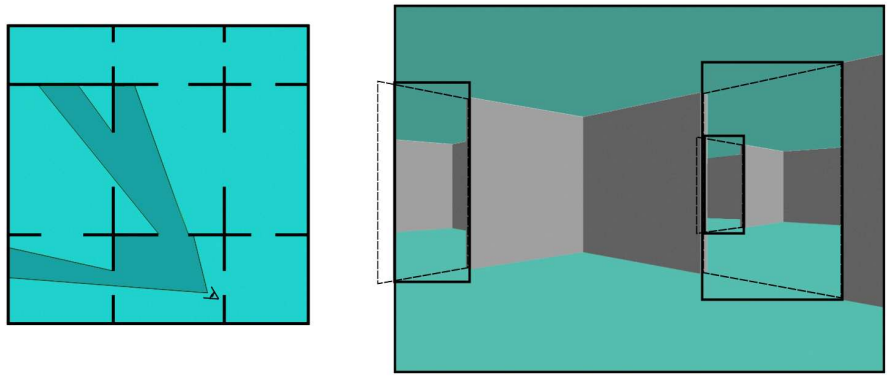

Figure 2: On the left, map of a set of cells. Viewpoint and visible parts of the cells are highlighted. On the right, the corresponding rendering. The original geometry for portals is highlighted by dashed lines, while actual VSAs are represented by thick lines.

i.e. the part of the viewport where no occluders have been rendered. We can notice that the VSA of a cell is usually reduced by passing through a portal. Thus, the portal check method described above can be refined using VSAs instead of the whole viewport of the portals. We represent a VSA using its axis-aligned bounding rectangle (Figure 2), or in an analogous way, indicating it with ranges on the two viewport axes. This simple representation has both advantages and disadvantages: it is conservative (by definition of the bounding rectangle) and can be handled in constant time (since we are using a fixed-size and axis-aligned representation instead of an arbitrary polygon) but there are situations where the bounding rectangle can be too loose thus leading to the computation of a VSA much bigger than the actual portal area. We chose this solution for its constant performance qualities.

The next four subsections illustrate the pseudocode of our algorithm. The first subsection deals with the culling algorithm that exploits cell-to-cell visibility, and is very similar to Luebke and Georges's algorithm, the other three discuss three optimizations of the algorithm: the first two have been introduced by us and respectively exploit distance from the viewer and cell-to-geometry visibility, the third is an improvement of the solution proposed in [Luebke and Georges 1995] for avoiding multiple renderings of the same cell.

3.2.1 Cell-to-cell visibility

Figure 3 presents the pseudocode of the culling algorithm. The *HandleCell* function has two parameters: the cell that has to be rendered (Cell) and its VSA (VSA). When *HandleCell* is called on a cell, it first renders the cell by calling the function *RenderCell*. This function descends the tree hierarchy of the X3D file that represents the cell (Cell.X3D) and renders the geometry found in all nodes of the X3D file. The *HandleCell* function then proceeds by considering every portal belonging to the given cell and tests for its visibility within the VSA of the cell. If the portal turns to be visible, the VSA is updated by computing the intersection between the VSA of the cell and the bounding rectangle for the projected portal vertices; the *Intersect* function computes an axis-aligned rectangle for the projected portal vertices and then performs an intersection of such rectangle with the VSA, resulting in another axis-aligned rectangle. Finally, the *HandleCell* function recursively calls itself by passing the cell on the other side of the portal (Portal.TargetCell) and the newly computed VSA as parameters. The visible cells are rendered by calling the *HandleCell* function with the viewer's cell and the full screen viewport as parameters.

```
HandleCell(Cell, VSA)
1.  RenderCell(Cell.X3D)
2.  for each Portal in Cell.Portals
3.    if ¬(IsPortalCulled(Portal, VSA))
4.      NewVSA := Intersect(Portal, VSA)
5.      HandleCell(Portal.TargetCell, NewVSA)
6.    endif
7.  endfor
```

Figure 3: Pseudocode for the culling method, showing the recursive call on visible adjacent cells.

To decide whether the portal polygon is inside or outside the cell VSA, we perform a number of tests on each of its vertices. These tests are performed by the *IsPortalCulled* function (Figure 4). This function has two parameters: a portal (Portal) and the VSA of a cell that is on one side of that portal (VSA). We consider five culling possibilities indicated in the pseudocode by five culling flags: *CullZ* means that the portal is completely behind the viewer, $CullX^-$ and $CullX^+$ mean that the portal is totally on the left or the right of the VSA, $CullY^-$ and $CullY^+$ mean that the portal is totally below or above the VSA. We start by initializing these culling flags to *true*, and subsequently set them to $false$ if we find a vertex (Vertex) of the portal polygon that fails the respective culling test. The *CullZ* test is performed in world coordinates: Vertex^T is Vertex transformed using the modelview matrix; in the pseudocode we assume that the viewer is on the origin $(0,0,0)$ looking towards the negative direction of the Z-axis: this means that a vertex that has a z value (Vertex_z^T) less than zero will be in front of the viewer and not behind. The *CullX* and *CullY* tests are performed in canonical view coordinates, therefore the vertex is further transformed (Vertex^P) using the projection matrix; x and y coordinates of the projected vertex (Vertex_x^P, Vertex_y^P) are compared to the boundaries of the VSA and culling flags are updated accordingly. The left, right, lower and upper boundary of the VSA are respectively defined as VSA.X^-, VSA.X^+, VSA.Y^-, VSA.Y^+. The algorithm finally returns *true* in the case that at least one of the culling flags is still *true*, which means that no vertex has failed the culling test associated to such flag.

3.2.2 Culling based on distance

We propose a further culling based on distance from the viewer: it is possible to cull objects that are more distant than a given thresh-

IsPortalCulled(Portal, VSA)

1. CullX$^-$:= CullX$^+$:= CullY$^-$:= CullY$^+$:= CullZ := **true**
2. **for each** (Vertex **in** Portal)
3. Vertex^T := Vertex * ModelviewMatrix
4. **if** (Vertex^T_z $<$ 0) CullZ = **false**
5. Vertex^P := Vertex^T * ProjectionMatrix
6. **if** (Vertex^P_x $\geq$ VSA.X$^-$) CullX$^-$:= **false**
7. **if** (Vertex^P_x $\leq$ VSA.X$^+$) CullX$^+$:= **false**
8. **if** (Vertex^P_y $\geq$ VSA.Y$^-$) CullY$^-$:= **false**
9. **if** (Vertex^P_y $\leq$ VSA.Y$^+$) CullY$^+$:= **false**
10. **endfor**
11. **return** CullX$^-$ **or** CullX$^+$ **or** CullY$^-$ **or** CullY$^+$ **or** CullZ

Figure 4: Pseudocode for the function that tests whether a portal is inside a VSA or not.

old. When properly tuned, this distance culling can help removing the processing of geometry that would end up to be too small to be appreciated on the display of a mobile device. Moreover, it is used in our solution to avoid that the size of visible cells becomes greater than the main memory of the device. This distance is tuned empirically, and its optimal value can vary based on the rendered model. This solution can be made seamless by having objects next to the threshold to vanish into fog. This culling test is inserted inside the *IsPortalCulled* function presented in Figure 4 by adding a new culling flag named *CullFog* in line 1 and inserting the line

> **if** (Vertex^T_z $\geq$ -FogThreshold) CullFog = **false**

between line 3 and 4 in the pseudocode because the test is performed in world coordinates. We use the opposite of the value of culling distance (FogThreshold) because we assume the viewer being in the origin $(0,0,0)$ and looking toward the negative direction of the Z-axis. The correct distance check would actually be

> **if** ($||$Vertex$^T||$ $\leq$ FogThreshold) CullFog = **false**

where $||\cdot||$ is the norm of a vector. The calculations involved to compute a vector norm include a costly square root and various multiplications, while the used method employs only a comparison, being at the same time quicker and conservative.

3.2.3 Cell-to-geometry visibility

To handle cell-to-geometry visibility, we augment the tree hierarchy already available in X3D files with a bounding box for each node. Figure 5 describes the function used to cull invisible branches of such a tree. This function has two parameters: an X3D node and the VSA of the cell it belongs to. The algorithm is quite similar to the one used for cell-to-cell visibility culling: a first test is performed to check whether the bounding box of the node (Node.BoundingBox) is totally outside the VSA or not. If it is not outside the VSA the node geometry is rendered (Node.Geometry) and the *CullAndRenderNode* is recursively called on all the children nodes (Node.Children).

The function *IsBoundingBoxVisible* can be implemented in a way similar to *IsPortalVisible*, because we need to consider each vertex of the bounding box to check if it is behind the viewer, to test its position in relation with the boundaries of the VSA and to check whether its distance from the viewer is less than the fog threshold. As with portals, if there is at least one test that does not fail with any vertex, then the bounding box and its content can be safely culled.

CullAndRenderNode(Node, VSA)

1. **if** *IsBoundingBoxVisible*(Node.BoundingBox, VSA)
2. *RenderGeometry*(Node.Geometry)
3. **for each** (Child **in** Node.Children)
4. *CullAndRenderNode*(Child, VSA)
5. **endfor**
6. **endif**

Figure 5: Pseudocode for the function that performs cell-to-geometry visibility culling and geometry rendering.

We can finally replace the line

> *RenderCell*(Cell.X3D)

in Figure 3 with the line

> *CullAndRenderNode*(Cell.X3D, VSA)

to exploit both cell-to-cell and cell-to-geometry visibility culling.

3.2.4 Avoiding multiple rendering of the same cell

The proposed method can lead to multiple renderings of the same objects. This can happen when a cell is visible through more than one portal. Since portals are not overlapping, different parts of the cell will be visible each time. An object might anyway be visible through more than one portal, thus leading to multiple renderings. This is not desirable when using particular effects, especially those using alpha blending. The solution proposed by Luebke and Georges [1995] is to mark each object as rendered so that it cannot be rendered more than once per frame; they report that this method has generally proved more efficient than rendering the objects more than once and culling them each time using the different VSAs, especially when rendering costs are much greater than culling costs. Our solution is similar. Instead of marking objects as rendered, which would require adding flags inside the inner data structures of the X3D player and the reset of such flags at every frame, we use a queue for all the cells to be rendered for a given frame. Cells are inserted into this queue at most once per frame, while a separate list is maintained for each cell, containing the VSAs through which the relative cell is visible. The synchronous rendering call represented by

> *CullAndRenderNode*(Cell.X3D, VSA)

in Figure 3 is therefore replaced by a call to

> *EnqueueCell*(Cell, VSA)

that enqueues cells and VSAs. The *CullAndRenderNode* call is performed when all the cells previously inserted in the queue are finally rendered. Cell-to-geometry culling is here performed by testing bounding boxes with the VSAs for the corresponding cell, until the test passes for one of them or fails for all of them.

4 Implementation

We implemented our system in C++ and OpenGL ES. We had to follow the OpenGL ES 1.0 Common Light (CL) specifications because that is the interface provided by the graphics processor of the accelerated device we used for testing (Dell Axim x51v with Intel 2700G GPU).

4.1 Some implementation issues

OpenGL ES 1.0 CL does not allow to query the driver for dynamic information, including the modelview and projection matrices. We thus had to store a local copy of these matrices and update their values each time an OpenGL function that changes them in driver is called.

We also tried to integrate stencil buffer culling into the algorithm. The idea was to associate a unique ID to each portal and render portals into the stencil buffer using the IDs. Then, cells that are visible through such portals are rendered onto the frame buffer only when corresponding pixels into the stencil buffer contain the given ID. This would reduce the amount of processed data by introducing a further occlusion culling step because the rendered portals would be z-culled when compared with occluding geometry in front of them. A simpler solution could consist in organizing the geometry in front-to-back order, achieving z-culling. However, this solution is less easy to extend for implementing transparency support than the solution based on stencil buffer because correct transparency support requires back-to-front rendering.
There is anyway a problem with the basic view frustum culling implemented by MobiX3D: when approaching a portal, the portal will be culled by the near plane of the view frustum and not rendered onto the stencil buffer, and consequently the whole room behind the portal would not be visible causing annoying popping artifacts. We plan to investigate ways to solve this problem as a future work.

5 Results

We tested the proposed approach on a Dell Axim X51v, equipped with a 624MHz Intel processor, 64Mb of RAM memory and the Intel 2700G for hardware 3D acceleration. Tests were carried out at 480x640 resolution.

The model used for the tests was a reproduction of the main building of our University, made of 28608 triangles. We started from a 3D model that was fully textured, subdivided into 39 cells and with 56 portals. The cells were too big and dynamic loading and flushing to and from memory was too slow, mainly because of file I/O operations. The lack of a texture caching system (which we are planning to investigate as a future work, together with the possibility of loading each texture on a separate thread) also led to loading time inefficiencies, because textures had to be loaded from storage as a cell became visible. With low-resolution textures, the mean loading time for a cell is about 18 seconds, while the mean loading time of a cell without textures is about 6 seconds (2 seconds for file I/O and 4 seconds for parsing). We therefore decided to substitute textures with colored materials; the model was also subdivided into smaller cells, leading to the final 3D model used for testing and consisting of 93 cells and 114 portals. The mean loading time for the smaller cells is about 2.5 seconds (1 second for file I/O and 1.5 seconds for parsing). The number of triangles per cell varies from a minimum of 102 up to a maximum of 452, and cells have between 2 and 4 portals. Figure 6 shows the subdivision in cells and portals of a part of the 3D model.

The achieved rendering speed using this final 3D model varies depending on the number of visible cells. We tried to display the whole model without using the portal culling algorithm, but the system crashed because there was not enough memory to load all the cells. Figure 7 illustrates a scene where 4 cells are visible, Figure 8 a scene where 2 cells are visible. Table 1 reports measured rendering speeds (in frames-per-second) both with and without the distance culling based on fog described in Section 3.2.2. Rendering speeds are slightly lower when using the fog because a further processing step is necessary to apply fog. This should not be misleading: enabling fog pays off when rendering models with long corridors because the farthest cells are culled by the fog and therefore the number of visible cells is smaller than it would be when fog is disabled.

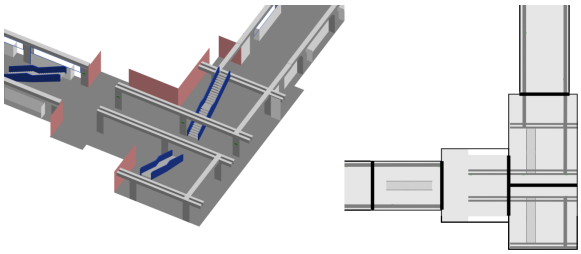

Figure 6: Part of the 3D model used for the tests with the corresponding subdivision in cells and portals. Portals are represented with black thick lines.

Cells	Triangles	Without fog	With fog
1	188	56.4 fps	42.7 fps
2	532	35.8 fps	34.7 fps
3	766	22.6 fps	20.7 fps
4	1156	19.4 fps	18.6 fps

Table 1: Rendering speeds (in frames-per-second) of the system both without and with the distance culling based on fog.

The results are encouraging and the large 3D model can be explored with satisfactory frame rates. The biggest problem encountered was the speed of loading and flushing cell data (both textures and geometry) between the storage and main memory. Moreover, it is not possible to keep more than 2000 triangles in memory. Anyway, these are not limitations of the proposed method, but optimization issues of X3D loading and resource handling code. We plan to solve these issues through the:

- optimization of X3D parsing code to obtain shorter loading times for the cells;
- design of a texture and geometry caching method to reduce the stream of data from storage to main memory;
- design of a predictive scheme to load cells in advance instead of loading them when they become visible.

6 Conclusions and future work

This paper proposed a system for interactive walkthrough of 3D models of large buildings on mobile devices. The system exploits a combination of view frustum and portal culling to minimize the non-visible geometry that is processed. The performance of our system in the visualization of a model composed by 28608 polygons shows the feasibility of using PDAs for interactively displaying large X3D building models.

Our research is now proceeding in two main directions. First, we will work at improving our visualization algorithms both in MobiX3D and in the portal culling system. MobiX3D improvements include incremental loading of X3D models, optimizations on the pre-processing stage during file loading (e.g. computation of bounding boxes) and integration of LOD (level of detail) support. Portal culling improvements include the implementation of the stencil culling optimization discussed in Section 4, a faster

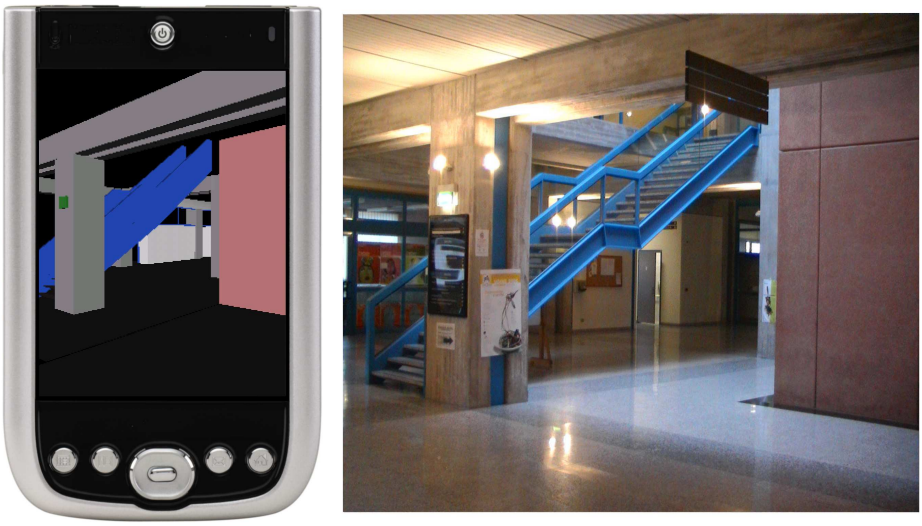

Figure 7: Part of the 3D model of the University of Udine (4 visible cells) with a photo of the corresponding area.

Figure 8: Part of the 3D model of the University of Udine (2 visible cells) with a photo of the corresponding area.

search method to calculate the cell where the viewer is and a method to sort cells in a back-to-front order before rendering them, as needed for correct transparency support.

Second, we will work at mobile 3D guidance for evacuation training. We will implement and test a system that generates evacuation paths based on the topology of the building and displays them on mobile devices. Then, we will employ RFID tags to make the system location-aware.

7 Acknowledgements

Our research has been partially supported by the Italian Ministry of Education, University and Research (MIUR) under a PRIN 2005 grant.

References

AILA, T., AND MIETTINEN, V. 2004. dPVS: An occlusion culling system for massive dynamic environments. *IEEE Computer Graphics and Applications 24*, 2, 86–97.

AIREY, J. 1990. *Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations*. PhD thesis, University of North Carolina at Chapel Hill.

ASSARSSON, U., AND MÖLLER, T. 2000. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools 5*, 1, 9–22.

CLARK, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*, 10, 547–554.

COHEN-OR, D., CHRYSANTHOU, Y. L., SILVA, C. T., AND DURAND, D. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 09*, 3, 412–431.

FUNKHOUSER, T. A., SÉQUIN, C. H., AND TELLER, S. J. 1992. Management of large amounts of data in interactive building walkthroughs. In *SI3D '92: Proceedings of the Symposium on Interactive 3D Graphics*, ACM Press, New York, NY, USA, 11–20.

HAUMONT, D., DEBEIR, O., AND SILLION, F. 2003. Volumetric cell-and-portal generation. *Computer Graphics Forum 22*, 3, 303–312.

HCI LAB – UNIVERSITY OF UDINE. 2006. MobiX3D website. http://hcilab.uniud.it/MobiX3D.

KUMAR, S., MANOCHA, D., GARRETT, W., AND LIN, M. 1996. Hierarchical back-face computation. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, Springer-Verlag, Berlin, Germany, 235–244.

LERNER, A., CHRYSANTHOU, Y., AND COHEN-OR, D. 2003. Breaking the walls: scene partitioning and portal creation. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, IEEE Computer Society, Los Alamitos, CA, USA, 303–312.

LERNER, A., CHRYSANTHOU, Y., AND COHEN-OR, D. 2006. Efficient cells-and-portals partitioning: Research articles. *Computer Animation and Virtual Worlds 17*, 1, 21–40.

LIPMAN, R. R. 2004. Mobile 3d visualization for steel structures. *Automation in Construction 13*, 119–125.

LUEBKE, D., AND GEORGES, C. 1995. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *SI3D '95: Proceedings of the Symposium on Interactive 3D graphics*, ACM Press, New York, NY, USA, 105–106.

MARVIE, J.-E., AND BOUATOUCH, K. 2004. A VRML97-X3D extension for massive scenery management in virtual worlds. In *Web3D '04: Proceedings of the 9th International Conference on 3D Web Technology*, ACM Press, New York, NY, USA, 145–153.

NADALUTTI, D., CHITTARO, L., AND BUTTUSSI, F. 2006. Rendering of X3D content on mobile devices with OpenGL ES. In *Web3D '06: Proceedings of the 11th International Conference on 3D Web Technology*, ACM Press, New York, NY, USA, 19–26.

NOCTEMWARE. 2005. Quake 3 Arena CE. http://www.noctemware.com/q3ce.html.

NURMINEN, A. 2006. m-LOMA - A mobile 3D city map. In *Web3D '06: Proceedings of the 11th International Conference on 3D Web Technology*, ACM Press, New York, NY, USA, 7–18.

PARALLELGRAPHICS. 2004. Pocket Cortona. http://www.parallelgraphics.com/products/cortonace/.

PULSE INTERACTIVE. 2005. Quake Mobile. http://www.pulsemobilegames.com/QuakeMobile.html.

SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2005. *OpenGL programming guide: the official guide to learning OpenGL, Version 2 (5th Edition)*. Addison-Wesley Professional.

SLATER, M., AND CHRYSANTHOU, Y. 1997. View volume culling using a probabilistic caching scheme. In *VRST '97: Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, ACM Press, New York, NY, USA, 71–77.

TELLER, S. J., AND SÉQUIN, C. H. 1991. Visibility preprocessing for interactive walkthroughs. In *SIGGRAPH '91: Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press, New York, NY, USA, 61–70.