EUROGRAPHICS'99 / P. Brunet and R. Scopigno (Guest Editors)

Volume 18 (1999), number 3

Occluder Shadows for Fast Walkthroughs of Urban Environments

Peter Wonka[*] and Dieter Schmalstieg[+]

[*]IRISA Rennes, France, and Vienna University of Technology, Austria, email: wonka@cg.tuwien.ac.at

[+]Vienna University of Technology, Austria, email: dieter@cg.tuwien.ac.at

Abstract

This paper describes a new algorithm that employs image-based rendering for fast occlusion culling in complex urban environments. It exploits graphics hardware to render and automatically combine a relatively large set of occluders. The algorithm is fast to calculate and therefore also useful for scenes of moderate complexity and walkthroughs with over 20 frames per second. Occlusion is calculated dynamically and does not rely on any visibility precalculation or occluder preselection. Speed-ups of one order of magnitude can be obtained.

1. Introduction

In walkthrough applications for urban environments, a user navigates through a city as a pedestrian or vehicle driver. Such a system is useful for example in traffic simulation, visual impact analysis of architectural projects and computer games. A desirable goal is real-time rendering with about 20-30 frames per second (fps), that are sustained even when the viewer's speed is high (e. g., 100km/h when driving in a fast car).

Scene complexity for large urban environments defeats any naive approach of trying to render everything in hardware. To remove larger scene parts before the final rendering traversal, two approaches seem to be promising:

- One way is to use impostors[20], mainly texture maps (textured depth meshes), to replace complex geometry. One texture map is valid only for a few frames and has to be updated frequently, which is only fast enough for a small number of impostors.
- The second method is to use occlusion culling[12]. A number of polygons are selected as occluders in each frame. The part of the scene that is occluded by one of these occluders is not visible and can be culled.

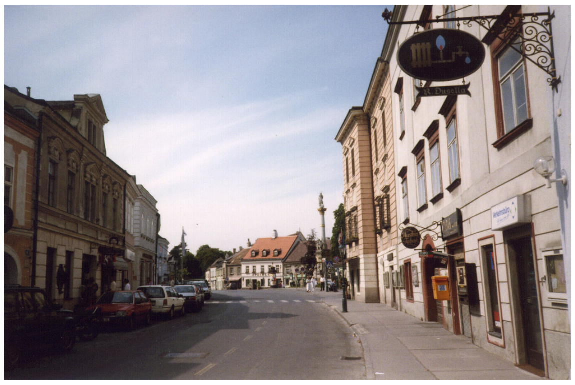

Figure 1: *Photograph from Klosterneuburg, Austria. Note how only a few buildings are visible because of occlusion.*

To understand optimization possibilities suitable for an urban model, we have to analyze how an urban environment looks like and examine its special properties which may be exploited by an acceleration algorithm. For the design of our occlusion algorithm, we tried to analyze the visibility and occlusion of real and virtual urban environments using data of European cities:

- Typically, an urban environment is modeled with a ground mesh or plane, with different objects placed on top of it. These are mostly buildings, trees and bushes, other smaller decorating objects like traffic lights, traffic signs, streetlights, mailboxes and the road network (if it is not directly a part of the ground). This type of environment is often referred to as 2½ dimensional. We can profit from this property for the design of our data structures and algorithm.

- For most parts of a walkthrough the view is rather restricted. Only from a few viewpoints can the observer see further than a few hundred meters. Figure 1 shows a typical view of a location in the city of Klosterneuburg, Austria, with a view between 100 and 200 meters. For such views, a small set of buildings or blocks (about 20) occludes the rest of the model.
- For more open parts of the scenes, however, like the place in front of a train station or the view along a broader straight street or a riverside, a few occluders are no longer sufficient. We still have very good occlusion, but the occlusion is made up with a larger number of occluders (up to a hundred). In the foreground of our scene overview (see Figure 10), there is a wide-open space where a larger number of occluders is necessary.

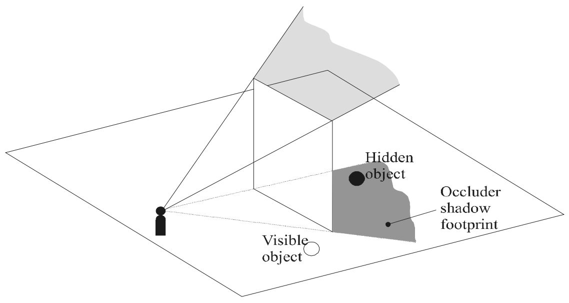

Figure 2: *An occluder shadow is used for fast rendering of urban environments. Note how the roof top of the shown building facade allows to determine whether an object is hidden from its occluder shadow footprint in the ground floor.*

A good occlusion algorithm should use all important occluders if possible, because any holes reduce the performance of the algorithm. However, identifying all important occluders analytically is a computationally costly process. Instead, a fast algorithm may simply use a larger set of potentially important occluders (according to some simple criterion such as size) to heuristically achieve the same degree of occlusion. Consequently, fast computation of occlusion is essential.

In this paper we introduce a new hybrid image-based and geometrical culling algorithm for urban environments exhibiting the properties discussed above. We compute *occluder shadows* (Figure 2) to determine which parts of the environment are invisible from a given viewpoint. Occluder shadows are shadow frusta cast from selected occluders such as building fronts. The 2½D property of an urban environment allows us to generate and combine occluder shadows in a 2D bitmap using graphics hardware.

Our algorithm has the following properties:

- It uses a relatively large set of occluders (up to 500) that are automatically combined by the graphics hardware.
- The algorithm is fast to calculate (about 13 ms on a mid-range workstation for our environment of 4 km^2) and therefore also useful for scenes of moderate complexity and walkthroughs with over 20fps.
- We calculate the occlusion dynamically and do not rely on any visibility precalculation or occluder preselection.

Furthermore, the algorithm is simple to understand and implement. These properties should make this algorithm suitable for many existing urban walkthrough implementations without introducing major drawbacks to existing systems.

After reviewing previous work, an overview of our algorithm is given and the various stages of our acceleration method are explained. Next we will describe our implementation and give results of different tests made with our example environment. Then we will discuss the applicability of our algorithm and compare it to the other existing solutions. Finally, we will give a preview of our future work in that field to achieve further optimizations.

2. Previous Work

Several methods were proposed to speedup the rendering of interactive walkthrough applications. General optimizations are implemented by rendering toolkits like Performer[16] that aim for an optimal usage of hardware resources. Level-of-detail (LOD) algorithms are very popular in urban simulation[14] because they do not require a lot of calculation during runtime. Heckbert[10] gives a good overview of recent LOD algorithms.

The idea of image-based simplification is to replace whole scene parts with an impostor. One impostor is usually only valid for a few frames and has to be updated frequently[17,19]. Other approaches use textured depth meshes[1,20] which incorporate depth information for efficient impostor update.

The idea of an efficient visibility culling algorithm is to calculate a conservative and fast estimation of those parts of the scene that are definitely invisible. The final hidden surface removal is done with the support of hardware, usually a z-buffer. A simple and general culling method is view frustum culling[3], which is applicable to almost any model. General algorithms for occlusion culling were proposed that calculate the occlusion in image space[9,14,11]. Green's hierarchical z-buffer[9] depends on special purpose hardware, which makes it impractical for real walkthrough systems. The hierarchical occlusion maps proposed by Zhang et al.[24] are a more practical approach, that was implemented and tested on existing graphics hardware. A set of occluders in the near field is rendered into the frame buffer and this image is used to calculate a hierarchy of occlusion maps, using mip mapping hardware. In a second pass, the scene graph is traversed, and the bounding boxes of the scene graph nodes are tested against the occlusion hierarchy.

A general method to accelerate visibility is to break down the viewspace into cells and precalculate for each cell a set of objects that are potentially visible[15] (potentially visible sets, PVS). For general scenes visibility precalculation can become quite costly with respect to time and memory, but for certain scenes like terrains, it is possible to use a priori knowledge about the scene structure for visibility calculation[4,21]. Cohen-Or[5] observes that in densely occluded scenes most objects are occluded by a single occluder near the viewpoint. He uses this observation for an algorithm to calculate PVS, which is demonstrated for an urban environment. In general, the proposed algorithms to precalculate visibility are too slow to be invoked for every frame.

For building interiors, most visibility algorithms decompose the model into cells connected by portals[22] and compute inter-cell visibility (potentially visible set, PVS). Luebke's algorithm[13] eliminates most precalculations and calculates the PVS during runtime. Urban environments were stated as a possible application, but we are not aware of any implementation. Unclear problems are how to render wide-open areas and the handling of the height structure.

For urban environments, occlusion culling with a few large occluders is a popular approach. Similar algorithms were proposed by Coorg[6,7], Bittner[2] and Hudson[12]. For each frame, a small set of occluders (about 5-30) likely to occlude a big part of the model is selected. Hudson[12] organizes the scene in a hierarchical data structure like a k-d tree or bounding-box tree and clips the individual nodes against the occluded region. Bittner uses a variation of the shadow volume BSP tree to merge occluder shadows. Coorg's latest algorithm calculates visibility events to make additional use of temporal coherence.

3. Overview of the Approach

3.1 Occluder shadows

The concept of occluder shadows is based on the following observation: Given a viewpoint O, an occluder polygon P casts an occluder shadow that occludes an area of the scene that lies behind the occluder, as seen from the viewpoint. This area can be seen as a shadow frustum that is determined by the occluder and the viewpoint. Objects fully in this shadow frustum are invisible.

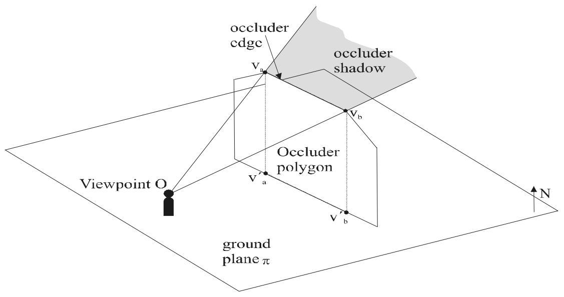

Figure 3: *This sketch shows the geometric relationship used for the occlusion culling. An occluder shadow is constructed from a given viewpoint and an occluder edge of an occluder polygon P.*

A more formal definition of an occluder shadow (see Figure 3) is given in the following:

- The environment is required to have a *ground plane*[1] π with a normal vector N (up direction).
- An *occluder polygon* P is a (not necessarily convex) polygon with its normal vector parallel to the ground plane (i.e., P is standing upright). P is constructed from a set of vertices $V=\{v_1,...,v_k\}$, connected by a set of edges $E=\{e_1,...,e_k\}$. An edge e from v_a to v_b is written as $e = (v_a, v_b)$. At least one edge of P must lie *in* the ground plane (i.e., $\exists e\in E : e = (v_a, v_b) \wedge v_a\in \pi \wedge v_b\in \pi$).
- An *occluder edge* is an edge $e\in E$ with $e = (v_a, v_b)$ of P that does not lie in the ground plane (i.e., $v_a\notin \pi \vee v_b\notin \pi$). Let v_a' and v_b' denote the normal projection of v_a and v_b onto π, respectively. Then all points along the two line segments (v_a, v_a') and (v_b, v_b') must be completely *inside* P, i.e. every point along each of the two line segments must also be contained in P.
- An *occluder shadow* is a quadrilateral with one edge at infinity, constructed from an occluder edge $e = (v_a, v_b)$ and two rays: $v_a +t (v_a - O)$ and $v_b +u (v_b - O)$. When looking in -N direction, the occluder shadow covers all scene objects that are hidden by the associated occluder polygon when viewed from O.

We exploit this observation by rendering the occluder shadow with orthogonal projection into a bitmap coincident with the ground plane. The height information (z-buffer) from this rendering allows to determine whether a point in 3D space is hidden by the occluder polygon or not.

[1] The definition is also valid if the urban environment is modeled on top of a height field, but this complicates the definition and is omitted here for brevity.

3.2 Algorithm outline

This section will explain how the concept of occluder shadows is used for rendering acceleration. We assume an urban model as described in the introduction. We rely on no special geometric features concerning the model. We need no expensive preprocessing and all needed data-structures are built on the fly at startup. For a completely arbitrary input scene, two tasks have to be solved that are not addressed in this paper: (1) The scene has to be decomposed into objects for the occlusion culling algorithm, which can be a semantic decomposition into buildings, trees and road segments or a plain geometric decomposition into polygon sets. (2) Useful occluders - mainly building fronts - must be identified. Details on preprocessing are given in section 4.1.

The scene is structured in a regular 2D grid coincident with the ground plane and all objects are entered into all grid cells with which they intersect. During runtime, we dynamically select a set of occluders for which occluder shadows are rendered into an auxiliary buffer - the *cull map* - using polygonal rendering hardware. Each pixel in the cull map (image space) corresponds to a cell of the scene-grid (object space). Therefore, the cull map is an image plane parallel to the ground plane of the scene (see section 4.2). Visibility can be determined for each cell (or object) of the grid according to the values in the cull map. To calculate the occluded parts of the scene, we can compare for each cell the z value in the cull map to the z-value of the bounding boxes of the objects entered into the scene-grid at this cell (see section 4.3).

When we render a shadow with the graphics hardware, we obtain z-values and color-buffer entries that more or less correspond to a sample in the center of the pixel. This is not satisfying for a conservative occlusion calculation. For a correct solution, we have to guarantee, that (1) only fully occluded cells are marked as invisible and (2) that the z-values in the z-buffer correspond to the lowest z-value of the occluder shadow in the grid cell and not a sample in the center. How this is achieved is detailed in section 4.4.

The advantage of our image-based approach over geometric occlusion computation is that the whole scene is not clipped against each shadow frustum separately. The calculation time depends only on the number of pixels and the number of occluders. For reasonable sizes of the cull map the overhead incurred by copying the frame buffer and visibility traversal is small and the algorithm runs always fast independent of the scene complexity.

Furthermore, we can use a larger number of occluder shadows, that are automatically combined by the graphics hardware. This *occluder fusion*[24] is very efficient, so that only a few invisible objects are overlooked by our culling algorithm.

To summarize, during runtime the following calculations have to be performed for each frame of the walkthrough:

- **Occluder selection**: For each frame a certain number of occluders has to be selected, that have a high chance to occlude the most parts of the invisible portion of the scene. The input to this selection is the viewpoint and the viewing direction.
- **Draw occluder shadows:** The selected occluders are used to calculate occluder shadows which are rendered into the cull map using graphics hardware.
- **Visibility calculation**: The cull map is traversed to collect all the potentially visible objects in the scene. Objects that are definitely not visible are omitted in the final rendering traversal.
- **Hidden surface removal:** The selected objects are passed to a hidden surface removal algorithm (z-buffer hardware).

4. Culling Algorithm

4.1 Preprocessing

At startup, we have to build two auxiliary data-structures: the scene grid and the occluder grid. To construct the scene grid we have to organize all objects in a regular grid covering the whole environment. All objects are entered into all grid cells[2] with wich they collide, so that we store a list of objects for each grid cell. As occlusion is computed on a cell basis, the complexity and size of the environment influence the choice of the grid size.

Among the objects in the scene, candidates for occluder polygons are identified. An occluder polygon should have good potential for occlusion - it should be fully opaque and large enough to possibly occlude other parts of the scene. An occluder polygon does not necessarily have to be a scene polygon. It could also be the combination of several coplanar polygons or the cross section of a solid object. Polygons extracted from objects like trees, bushes, vehicles, or roads violate one or more of these properties, which leaves mainly building fronts and polygons constructed from roof edges as useful occluders. Other possible candidates are walls, fences or monuments. The occluders are stored structure as connected occluder-chains in a separate grid data for fast retrieval at runtime. Typically, in large-scale urban modeling buildings are created as extrusions of their footprints. Therefore, it is usually easy to identify occluders together with connectivity information.

[2] Note that usually objects span more than one cell.

These tasks can be done very fast and take a time comparable to the loading of the scene.

4.2 Cull map creation

Once an occluder polygon is selected, the associated occluder shadow quadrilateral is rendered into the cull map. Since a quadrilateral with points at infinity cannot be rendered, we simply assume very large values for the t and u parameters from section 1.1, so that the edge in question lies fully outside the cull map. When the quadrilateral is rasterized, the z-values of the covered pixels in the cull map describe the minimal visible height of the corresponding cell in the scene-grid. The graphics hardware automatically fuses multiple occluder polygons rendered in sequence (Figure 4 and Figure 11).

When occluder shadows intersect, the z-buffer automatically stores the z-value, which provides the best occlusion.

Previous approaches selected a set of occluders according to a heuristic depending on the distance from the viewpoint, the area and the angle between the viewing direction and the occluder-normal[7,2]. This is a necessary step if only a small number of occluders can be considered, but it has the disadvantage that the selection has to be precalculated, which makes it more difficult to add occlusion culling to an existing visualization system. In our approach, the number of occluders is large enough, so we use only the distance from the viewpoint as a criterion for dynamic selection during runtime (Figure 12).

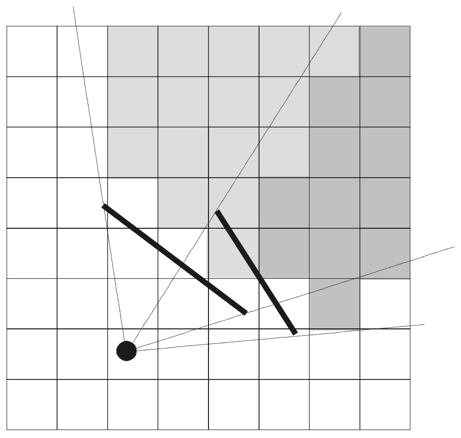

Figure 4: *The cull map is created by drawing occluder shadows as polygons using graphics hardware. Overlapping occluders are correctly fused by standard rasterization and z-buffer comparison.*

Our experiences have shown that a more sophisticated precomputed occluder selection does not improve the occlusion performance of our implementation. However, we perform a simple backface culling test on all occluder candidates. Building fronts we consider are generally closed loops of polygons, and it is sufficient to consider only front facing polygons.

For each of the occluders we render the occluder shadow in the cull map. The resulting cull map is passed to the next step, the actual visibility calculation.

4.3 Visibility calculation

To determine the visible objects that should be passed to the final rendering traversal, we have two options: We can either traverse the cull map or we can traverse the scene graph to determine which objects are visible. While traversing the scene graph would make a hierarchical traversal possible, we found that a fast scan through the cull map is not only simpler, but also faster.

Our algorithm therefore visits each cell that intersects the viewing frustum and checks the z-value in the cull map against the height of the objects stored in that cell. We use a two-level bounding box hierarchy to quickly perform that test: first, the z-value from the cull map is tested against the z-value of the bounding box enclosing all objects associated with the cell to quickly reject cells where all objects are occluded. If this test fails, the bounding boxes of all individual objects are tested against the cull map. Only those objects that pass the second test must be considered for rendering.

In pseudo code, the algorithm looks like this:

```
for each cell C(i,j) in the grid do
  if C(i,j).z > cullmap(i,j)
    for each object O(k) in C(i,j) do
      if O(k).z > cullmap(i,j).z
      and O(k) not already rendered
        render O(k)
```

4.4 Cull map sampling correction

As stated in the previous section, the simple rendering of the occluder shadows generally produces a non conservative estimation of occlusion in the cull map because of undersampling. For (1) correct z-values in the cull map and (2) to avoid the rendering of occluder shadows over pixels that are only partially occluded, we have to shrink the occluder shadow depending on the grid size and the rasterization rules of the graphics hardware.

Our method to overcome these sampling problems requires information about the exact position used for sampling within one pixel. Current image generators usually take a sample in the middle of a pixel. For example, the OpenGL[18] specification requires that (1) the computed z-value corresponds to the height in the middle of a pixel, (2) a pixel fragment is generated when the middle of a pixel is covered by the polygon and (3) For two triangles, that share a common edge (defined by the same endpoints), that crosses a pixel center, exactly one has to generate the fragment.

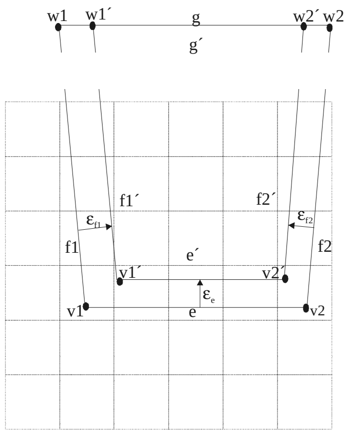

Figure 5: Sampling correction is performed by moving three edges of the occluder shadow inwards so that no invalid pixels can accidentally be covered.

Given these sampling constraints, a conservative solution for both stated problems is to shrink the occluder shadow geometrically (Figure 5). Consider an occluder shadow quadrilateral constructed from the vertices v_1, v_2, w_1, and w_2, defining the occluder edge $e = (v_1, v_2)$ and three other edges $f_1 = (v_1, w_2)$, $f_2 = (v_2, w_2)$ and $g = (w_1, w_2)$. By moving e, f_1, and f_2 towards the interior of the polygon along the normal vector by distances ε_e, ε_{f1}, and ε_{f2}, respectively, we obtain a new, smaller quadrilateral with edges e', f_1', f_2' and g', that does not suffer from the mentioned sampling problems. g is outside the viewing frustum and need not be corrected, it is only shortened.

To avoid sampling problems in x-y-direction, the correction terms ε_e, ε_{f1}, and ε_{f2} have to depend on the orientation of e, f_1, and f_2, respectively. An upper bound for them is $l\sqrt{2}$, where l is half the length of a grid cell. However, it is sufficient to take $(|n_x|+|n_y|) \cdot l$, where n is the normalized normal vector on an edge that should be corrected. This term describes the farthest distance from the pixel middle to a line with normal vector n that passes through a corner of the pixel.

If the occluder or the angle between the occluder and the viewing direction is small, f_1' and f_2' may intersect on the opposite side of e as seen from e'. In this case, the occluder shadow quadrilateral degenerates into a triangle, but this degeneration does not affect the validity of the occlusion.

To avoid sampling problems in z-direction, we render the new, smaller quadrilateral with a slope in z that is equal to or smaller than the slope of the original quadrilateral. For the case that the occluder edge e is parallel to the ground plane, the gradient (i.e., direction of steepest slope in z) of the occluder shadow is identical to the normal vector n of the occluder edge. Thus, the sampling problem for z-values is solved implicitly. If the occluder edge is not parallel to the ground plane, we have to consider an additional correction term that is dependent on the gradient of the occluder shadow. Note that only in this case the z-value of the vertices v1' and v2' is different from that of v1 and v2.

5. Implementation

Our implementation takes input scenes produced by the modeling system VUEMS[8] that was developed for traffic simulation. Occluder information and other semantic structures are created by the modeling system and written into a separate file. We reuse this information instead of extracting occluder polygons at startup.

We implemented our algorithm on an SGI platform using Open Inventor (OIV) as high-level toolkit for the navigation in and rendering of the urban environment. The cull map handling is done with native OpenGL (which also guarantees conservative occlusion) in an off-screen buffer (pbuffer). OIV allows good control over rendering and navigation and is available on different platforms. In our cull map implementation, we use only simple OpenGL calls, which should be well supported and optimized on almost any hardware platform. The most crucial hardware operation is fast copying of the frame buffer and the rasterization of large triangles with z-buffer comparison. Hardware accelerated geometry transformation is not an important factor for our algorithm.

We tested and analyzed our implementation on two different hardware architectures: an SGI Indigo2 Maximum Impact representing medium range workstations and an O2 as a low end machine. Whereas the implementation of most tasks met our estimated time frames, the copying of the cull map showed significantly different behavior on various hardware platforms and for different pixel transfer paths. Where the time to copy the red channel of a 250x250 pixel wide frame buffer area on the Maximum Impact takes about 3 ms, this step takes over 20 times as long on the O2, where only the copying of the whole frame buffer (red, green, blue and alpha channels) is fast. These differences have to be considered in the implementation. Furthermore, copying the z-buffer values on an O2 is more time-consuming and not efficient enough for our real-time algorithm.

Due to the fact that fast copying of the z-buffer is not possible (which is also stated by [24]), we had to resort to a variation of the algorithm that only needs to copy the frame buffer:

1. At the beginning of each frame, each value of the z-buffer is initialized with the maximum z-value from the objects in the corresponding grid cell. z-buffer writing is disabled, but not z-comparison. Next the color buffer is cleared and the viewing frustum is rendered with a key color meaning *visible*. The cells not containing any objects are initialized to have very

high z-values, so that they are not marked visible by rendering the viewing frustum.

2. Each occluder shadow is then written with a key color meaning *invisible* overwriting all pixels (= grid cells) that are fully occluded.
3. Finally, the resulting frame buffer is copied to memory for inspection by the occlusion algorithm.

The sampling correction for the occluder shadows makes it necessary to keep information of directly connected occluders. All connected occluders are stored as poly-lines. For our implementation we use an extension of the described sampling correction algorithm to a set of connected occluders, where most parts of the calculation are precalculated at startup.

Furthermore, all occluders that come from building fronts have an oriented normal-vector that is used for backface culling. This helps to reduce the number of occluders by about 50%.

6. Results

To evaluate the performance of our algorithm we performed several tests using a model of the city of Vienna. We started with the basic data of building footprints and building heights and used a procedural modeling program to create a three dimensional model of the environment, compatible with the output of the VUEMS[8] modeling system. This made it possible to control the size and scene complexity of the test scene. We modeled each building with a few hundred polygons. For the first test, we created a smaller model with 227355 polygons, which covers an area of about a square kilometer.

We recorded a camera path through the environment where we split the path in two parts. In the first part (until frame number 250) the camera moves along closed streets and places. This is a scenario typically seen by a car driver navigating through the city. In the second part the viewer moves in a wide-open area (in the real city of Vienna there is a river crossing the city).

For our walkthroughs, we configured our system with a grid size of 8 meter and we selected all occluders up to 800 meter (which is on the safe side). In most frames, we select between 400 and 1000 occluders and drop about half of them through backface culling. The construction of the auxiliary data structures and occluder preprocessing does not take longer than 10 seconds, even for larger scenes, while loading of the geometry takes sometimes over a minute. The cull map size for this test is 128x128.

Figure 6 shows the frame times for the walkthroughs on the Indigo2 in ms. The curve *frustum culling* shows the frame times for simple view frustum culling. The second curve *occlusion culling* shows the frame times for our algorithm. We see that we have good occlusion and that the algorithm is fast to compute. We have a speedup of 3.7 for the Indigo 2. Furthermore, the frame rate stayed over 20 fps. The frame rates in the second part are also high, because the model is structured in a way so that little additional geometry comes into sight.

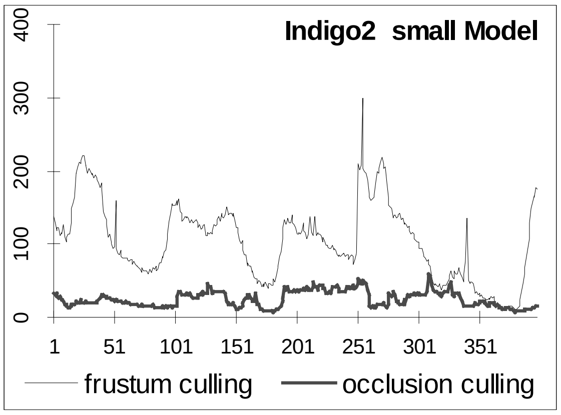

Figure 6: *The frame times for the walkthrough of a model consisting of 227355 polygons on an Indigo2 (frame times on y-axis in ms).*

For the second test, we used the same scene embedded in a larger environment to test (1) the robustness of the occlusion and (2) the behaviour of the algorithm when the viewpoint is located on wide-open places. The new city has about 1,300,000 polygons and covers a size of 4 km^2 (see Figure 10). The cull map size for this test is 256x256. To be able to compare the results, we used the same camera path in the larger environment. The results for the Indigo2 are shown in Figure 7 and a summary of all results is given in Figure 8.

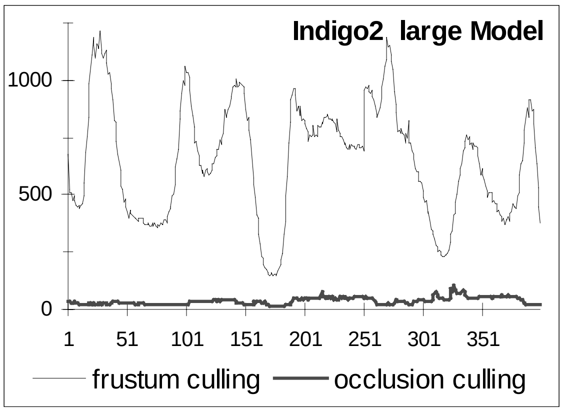

Figure 7: *The frame times for the walkthrough of a model consisting of 1,300,000 polygons on an Indigo2 (frame times on y-axis in ms).*

It can be seen that the frame rate in the first part is almost the same as in the smaller model. Almost exactly the same set of objects were reported visible for both city models. Longer frame times are only due to higher algorithm computation time and overhead from data structure management.

In the second part of the walkthrough, the field of view is not fully occluded for several hundred meters (sometimes up to 700) and a large number of buildings become visible. Still our algorithm works efficiently and selects only few invisible objects.

Indigo2	**Model 1**	**Model 2**
Frustum culling	103 ms	653 ms
Occlusion culling	25 ms	37 ms
Algorithm time	7 ms	13 ms
Part 1	26 ms	32 ms
Part 2	24 ms	44 ms
Speedup	4,2	17,8

O2	**Model 1**	**Model 2**
Frustum culling	312 ms	2029 ms
Occlusion culling	68 ms	108 ms
Algorithm time	13 ms	25 ms
Part 1	68 ms	84 ms
Part 2	67 ms	149 ms
Speedup	4,6	18,8

Figure 8: *Summary of all measurements for the two walthrough sequences. Speedup factors between 4 and 18 were be obtained. Frustum culling: average frame time using view frustum culling. Occlusion culling: average frame time using our occlusion culling algorithm. Algorithm time: average calculation time of our algorithm. Part1 (Part2): average frame time of the walkthrough in Part1 (Part2). Speedup: speedup factor compared to view frustum culling..*

This evaluation demonstrates that the occlusion is robust and that our algorithm generally does not leave unoccluded portions in the field of view if suitable occluders are present (see Part 1 of the walkthrough). Even the low-end O2 workstation was able to sustain a frame time of 108 ms.

In the second part of the walkthrough we demonstrated the algorithm behavior in a more challenging environment, where dense occlusion is no longer given. The performance drops as expected, but nevertheless the frame time of the Indigo2 does not exceed 85ms (the maximum frame time) and the average frame time for the second part also stays below 50ms.

The overall improvement for the whole path (compared to simple view frustum culling) equals a factor of about 18. However, results also indicate that performance depends on the presence of a large amount of occlusion. To unconditionally sustain a high frame rate, a combination with other acceleration methods (see proposed future work) is desirable.

A third experiment was conducted to examine the theoretical performance limits of the algorithm. The goal was to assess how the amount of detected occlusion is related to grid size. Completely accurate occlusion can be computed if all possible occluders are rendered into a cull map with infinitely small grid elements. We therefore performed a walkthrough for model 1 with cell sizes of 8, 6, 4, and 2 meters. All occluders in the view frustum were rendered into the cull map unconditionally. A breakdown of the times for individual parts of the algorithm is given in (see Figure 9).

The times reported for actual rendering of buildings indicate that a standard cell size of 8m has only about 20% overhead compared to the much smaller 2m cells. This overhead is mainly due to the fact that whole buildings are treated as one object. Despite its discrete nature, the algorithm estimates true visibility very accurately. We expect that better results can be achieved with more careful scene structuring while keeping cell size constant.

We also observed that the rasterization of occluder shadows and copying of the cull map becomes the bottleneck of the system for larger environments. cull map traversal is really fast for a typical setup (cull map < 256x256), so that it is not necessary to find optimizations through hierarchical traversals.

Indigo2				
Cell size	**8m**	**6m**	**4m**	**2m**
Cull map size	128 x128	174 x174	256 x256	512 x512
Render shadows	5.3	5.3	5.3	8
Copy cull map	1.3	1.7	2.3	6.3
Traverse cull map	0.5	0.7	0.9	2.2
Render buidlings	18.3	17.4	17	15.8
complete frame time	25.4	25.0	25.5	32.3

Figure 9: *This table shows the results of the walkthrough in the smaller model on the Indigo 2 for different cell sizes (cull map sizes). All occluders in the viewing frustum were selected for this test (time values are given in ms).*

7. Discussion

For the applicability of an occlusion culling algorithm we found that fast calculation times are an important feature. If we assume the goal of 20 fps for a fluent walkthrough, we have only 50ms per frame. If 30 ms are spent on the occlusion algorithm, little is left for other tasks like rendering, LOD selection or image-based simplifications. Such an algorithm may accelerate from 2 to 10 frames per second, but not from 4 to 20 fps, which is a fundamental difference. We therefore found it more important to have an algorithm that is robust and does not degenerate under worst case conditions rather than an algorithm that occludes as much as possible.

An expensive algorithm depends on strongly occluded scenarios to compensate for its calculation time, whereas a fast algorithm can also result in speedups for slightly occluded environments. Consider HOMs[24] used for the UNC walkthrough system, for which the calculation times of the occlusion algorithm itself on mid-range machines are relatively high. The pure construction time of the occlusion map hierarchy given the basic occlusion map on an SGI Indigo2 Maximum Impact is about 9 ms. This is about the time for one complete frame of our algorithm on the same machine. However, it must be stressed that HOMs provide a framework that is suitable for far more general scenes which cannot be handled by our algorithm.

The geometrical algorithms of Coorg[7], Hudson[12] and Bittner[2] operate on a similar idea. Coorg's algorithm makes additional use of temporal coherence, and Bittner merges occluder shadows. All algorithms have the advantages that they can be used as a general framework for three-dimensional occlusion culling, but we will only relate to their performance in urban environments. Geometrical algorithms have the following advantages:

- Independence from graphics hardware makes a multi-processor implementation simpler.
- The algorithm's scalability is not constrained by a fixed size cull map.

In contrast, our algorithm has the following advantages:

- It is faster to compute and we can handle an order of magnitude more occluders.
- We do not rely on the precalculation of suitable occluders for viewspace cells.
- The calculated occlusion is more robust and our algorithm also provides good occlusion when the viewpoint is located on open places.

8. Conclusions and Future Work

We have presented a new algorithm for fast walkthroughs of urban environments based on occluder shadows. The algorithm has proven to be fast, robust, and useful even for scenes of medium complexity and low-end graphics workstations. It is capable of accelerating up to one order of magnitude, depending mostly on support for fast frame buffer copying, which hopefully will be also available on low-cost hardware in the near future.

For very large-scale urban environments, occlusion culling alone is not sufficient if views with open view corridors into a far field with huge geometric complexity are possible. To overcome this restriction, we are currently working towards the integration of occluder shadows and a ray casting/image cache algorithm for far field rendering[23], which should eliminate the mentioned restriction. We hope to be able to present a true 20 fps walk through a very large-scale (10 million polygons) urban environment in the near future.

9. Acknowledgments

This research is supported in part by the Austrian Science Fund (FWF) contract no. P-11392-MAT and TMR Research Network PAVR. Special thanks to Stephane Donikian for organisational support, Michael Wimmer for fruitful discussions about rendering hardware, Michael Kofler from the TU Graz for the basic city model of Vienna and Erich Wonka for pictures of Klosterneuburg.

References

1. D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, D. Manocha. A Framework for the Real-Time Walkthrough of Massive Models. UNC, Technical Report #98-013, 1998.
2. J. Bittner, V. Havran, P. Slavík. Hierarchical Visibility Culling with Occlusion Trees. Computer Graphics International 1998 Proceedings, pp. 207-219, June, 1998.
3. J. Clark: Hierarchical geometric models for visible surface algorithms. Communications of the ACM, 19(10), pp. 547-554, October, 1976.
4. D. Cohen-Or, A. Shaked. Visibility and Dead-Zones in Digital Terrain Maps. Computer Graphics Forum, vol. 14, num. 3, pp. 171-180, September, 1995.
5. D. Cohen-Or, G. Fibich, D. Haperin, E. Zadicario. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. Proceedings of EUROGRAPHICS'98, 1998.
6. S. Coorg and S. Teller. Temporally Coherent Conservative Visibility. Proceedings of the Twelfth Annual Symposium on Computational Geometry 96, pp. 78-87, May, 1996.
7. S. Coorg, S. Teller. Real-Time Occlusion Culling for Models with Large Occluders. Proceedings of the Symposium on Interactive 3D Graphics, 1997.
8. S. Donikian. VUEMS: A Virtual Urban Environment Modeling System. In Proceedings of the Computer Graphics International '97, pp.84-92, 1997.
9. N. Greene, M. Kass: Hierarchical Z-Buffer Visibility, Proceedings of SIGGRAPH'91, pp. 231-240, 1993.
10. P. Heckbert, M. Garland. Survey of Polygonal Surface Simplification Algorithms. Technical Report, CS Dept., Carnegie Mellon University, 1997.
11. H. Hey, R. Tobler. Lazy Occlusion Grid Culling. Technical Report, Vienna University of Technology, Institute of Computer Graphics, TR-186-2-99-12, 1999.

12. T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, H. Zhang. Accelerated Occlusion Culling using Shadow Frusta. 13th International Annual Symposium on Computational Geometry (SCG-97), 1997.
13. D. Luebke, C Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. Proceedings of the Symposium on Interactive 3D Graphics, ACM Press, April, 1995.
14. W. Jepson, R. Liggett, S. Friedman. An Environment for Real-time Urban Simulation. Proceedings of the Symposium on Interactive 3D Graphics, 1995.
15. H. Plantinga. Conservative visibility preprocessing for efficient walkthroughs of 3D scenes. Proceedings of Graphics Interface '93, pp. 166-173, May, 1993.
16. J. Rohlf, J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. Proceedings of SIGGRAPH'94, pp. 381-395, 1994.
17. G. Schaufler, W. Stuerzlinger. A Three-Dimensional Image Cache for Virtual Reality. Proceedings of EUROGRAPHICS'96, 1996.
18. M. Segal and Kurt Akeley : The OpenGL Graphics System : A Specification (Version 1.2.2), 1998.
19. J. Shade, D. Lischinski, D. Salesin, T. DeRose, J. Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. Proceedings of SIGGRAPH'96, pp. 75-82, 1996.
20. F. Sillion, G. Drettakis, B. Bodelet. Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery. Computer Graphics Forum, 1997.
21. A. Stewart. Hierarchical Visibility in Terrains. Eurographics Rendering Workshop 1997, pp. 217-228, June, 1997.
22. S. Teller, C. Sequin. Visibility preprocessing for interactive walkthroughs. Proceedings of SIGGRAPH'91, pp. 61-69, 1991.
23. M. Wimmer, M. Giegl, D. Schmalstieg. Fast Walkthroughs with Image Caches and Ray Casting. Institut for Computergraphics, TU Vienna, Technical Report TR-186-2-98-30, to appear in EGVE'99, 1999.
24. H. Zhang, D. Manocha, T. Hudson, K. E. Hoff. Visibility Culling Using Hierarchical Occlusion Maps. SIGGRAPH 97 Conference Proceedings, 1997.

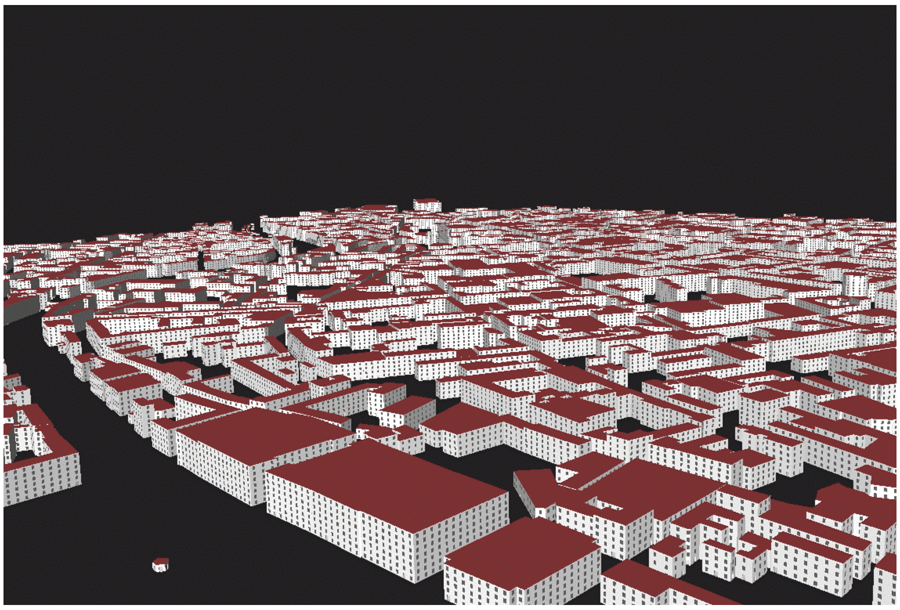

Figure 10: *This model of the city of Vienna with approximately 1.3M polygons was used for our experiments.*

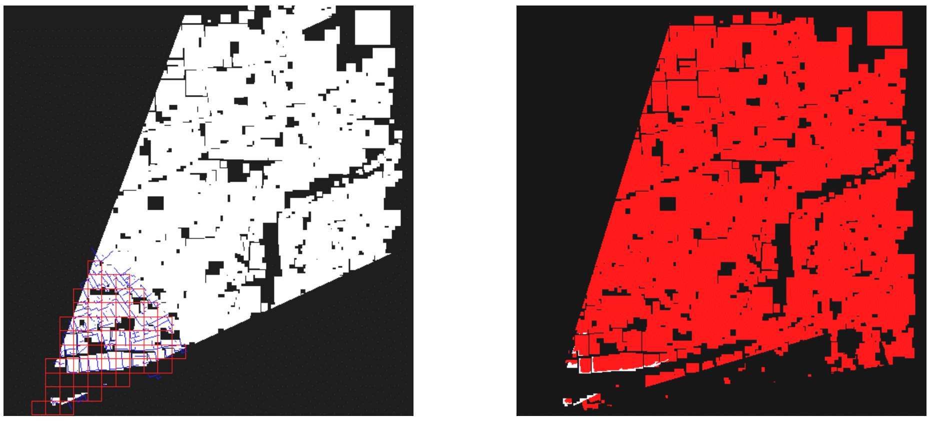

Figure 11: *Two views of the cull map used for occlusion culling. The left view shows the grid cells inspected for suitable occluders (in red) and selected occluders near the viewpoint (in blue). The right view shows the culled portion of the model (in red) and the remaining cells after culling (in white).*

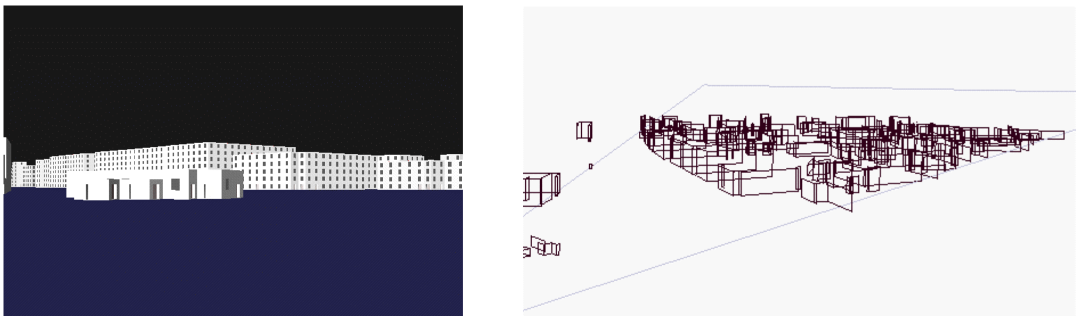

Figure 12: *For the viewpoint used in Figure* 11, *the resulting image is given on the left. The right view shows a wireframe rendering of the occluders to give an impression of occluder density.*