Institut für Computergraphik und Algorithmen
Technische Universität Wien

Karlsplatz 13/186/2
A-1040 Wien
AUSTRIA

Tel: +43 (1) 58801-18675
Fax: +43 (1) 5874932

Institute of Computer Graphics and Algorithms
Vienna University of Technology

email:
technical-report@cg.tuwien.ac.at

other services:
http://www.cg.tuwien.ac.at/
ftp://ftp.cg.tuwien.ac.at

Real-Time Occlusion Culling With A Lazy Occlusion Grid

Heinrich Hey Robert F. Tobler Werner Purgathofer

TR-186-2-01-02
January 2001

Abstract

We present a new conservative image-based occlusion culling method to increase the speed of hardware accelerated rendering of very complex general scenes which may consist of millions of polygons without time-expensive preprocessing. The method is based on a low-resolution grid upon a conventional z-buffer or an occlusion-buffer. This grid is updated in a lazy manner which reduces the number of expensive occlusion queries at pixel-level significantly compared to a busy update. It allows fast decisions if an object is occluded or potentially visible. The grid is used together with a bounding volume hierarchy that is traversed in a front to back order and which allows to cull large parts of the scene at once. We show that the method works efficiently on today´s available hardware and we compare lazy and busy updates.

Real-Time Occlusion Culling With A Lazy Occlusion Grid

Heinrich Hey Robert F. Tobler Werner Purgathofer

Abstract

We present a new conservative image-based occlusion culling method to increase the speed of hardware accelerated rendering of very complex general scenes which may consist of millions of polygons without time-expensive preprocessing. The method is based on a low-resolution grid upon a conventional z-buffer or an occlusion-buffer. This grid is updated in a lazy manner which reduces the number of expensive occlusion queries at pixel-level significantly compared to a busy update. It allows fast decisions if an object is occluded or potentially visible. The grid is used together with a bounding volume hierarchy that is traversed in a front to back order and which allows to cull large parts of the scene at once. We show that the method works efficiently on today´s available hardware and we compare lazy and busy updates.

1 Introduction

Complex scenes may consist of millions of polygons, much more than available graphics hardware can render at interactive frame-rates. Hierarchical view frustum culling and back face culling [20,22] reduce the number of drawn primitives to some extent but in many scenes this number will still be too high. Usually only a small part of such a scene is visible. Therefore occlusion culling methods try to determine those parts of the scene that are invisible due to occlusion by other parts so that these occluded objects do not have to be considered for drawing.

In this paper we present a new conservative image-based occlusion culling method for general scenes. It does its occlusion calculations on the fly during the display phase. It does not require time-expensive visibility-preprocessing which makes it suited for applications that need to diplay the scene instantly after the user has modified it, eg. for interactive changes in animations or virtual environments.

The image is subdivided into a low-resolution grid of cells with an occlusion state for each cell that shows if the complete area of the cell is occluded by already drawn objects or if the area is completely or partially free (unoccluded). This allows to determine if an object is occluded by querying the occlusion states of the few cells in the object´s image area instead of testing the content of the underlying occlusion-buffer or z-buffer for every pixel in this area.

The major feature that distinguishes our method from related methods like the hierarchical z-buffer [13] is that whe do not use a *busy update*, which means that every time after an object has been drawn the occlusion state of all cells in its image area has to be updated. Instead we use a *lazy update*, which means that the state of a cell is only updated if it is being queried and if it is currently marked as outdated because another object has been drawn into that area before. This has the advantage that significantly fewer updates and therefore fewer expensive occlusion queries at pixel-level are necessary. The lower number of updates has the following reasons:

- An object is potentially visible if the first unoccluded cell is found in its image area. The rest of the cells which may be outdated do not have to be queried. Up-to-date cells are queried before outdated cells to minimize the chance of needing an update even more.
- Often several objects draw into a cell´s area before the cell is queried and updated.

We have chosen a flat grid instead of a pyramid [13,15,37] because due to the lazy update it would be very likely that the higher level entries of such a pyramid are outdated because some of their lower level sub-entries are outdated. The update of such a higher level entry would require to update its outdated lower level sub-entries. Therefore most of the time the lower level entries would have to be queried and updated anyhow, similar as it is done with the simpler grid.

We use this image-based occlusion test on a bounding volume hierarchy that is traversed in a front to back order. This way a large part of the scene can be culled at once if its bounding volume is rated as occluded.

In section 2 we review existing techniques for occlusion culling with special emphasis on image-based methods. Section 3 describes the lazy occlusion grid with its occlusion-buffer and z-buffer variant. In section 4 we explain the usage of the bounding volume hierarchy and the front to back traversal of the scene. Section 5 describes how hardware acceleration can be done and section 6 presents ideas for future extensions. In section 7 we describe our implementation and present our results which includes a comparison of lazy and busy updates. Section 8 presents our conclusions.

2 Previous Work

The important role of occlusion culling for complex scenes has resulted in several different approaches. Exact global visibility methods try to solve the problem by representing all visibility events in a scene for all possible viewpoints [9,10,25]. The expense of exact visibility calculation can be avoided by overestimating the set of visible objects. In a static environment such potentially visible sets (PVS) can be precomputed by subdividing the scene into cells and calculating the PVS of each cell [2,6,11,19,26,27,30,32,33,35,36]. This precomputation usually requires between several minutes and several hours depending on the scene complexity. The advantage of precomputed PVSs is that the display-phase is usually very fast because the objects in the PVS of the viewpoint´s cell can be rendered without any further occlusion culling-overhead. Therefore these methods are often used eg. in games [1] where the frame-rate is the major criterion and the time-expensive precomputation does not hurt so much.

Methods that, like our new method, do their occlusion calculations on the fly during the display-phase [21,23,24,34] have the advantage that they do not need a time-expensive precomputation but of course the occlusion calculation during the display-phase produces some overhead.

The hierarchical z-buffer [13,31] is an image-based on-the-fly method that uses a pyramid of z-values to cull objects in large already completely occluded parts of the image with only a few z-comparisons. The scene is subdivided into an octree to realize a hierarchy of bounding volumes which are tested against the hierarchical z-buffer. If a bounding volume is completely occluded then its sub-objects and sub-bounding volumes are also occluded and can be culled. This method can be extended to error-bounded antialiasing [14].

Hierarchical coverage masks [15] also use a pyramid, but this one only contains the occlusion state instead of a z-value. Each entry in the pyramid has 8x8 instead of 2x2 subentries, therefore the pyramid has fewer levels. The major feature of this method is that it uses fast table-lookups and bit-operations instead of traditional scanline-rasterization. Geometry is traversed in exact front to back order. Traditional graphics-hardware can be used for texturing and shading.

Hierarchical occlusion maps [37] work with a pyramid of averaged occlusion-values which is generated with bilinear filtering using graphics hardware. This occlusion-pyramid is build initially for a few heuristically chosen occluder-polygons. Several other methods also use a small set of occluders [5,7,8,17]. This is based on the assumption that these polygons occlude large parts of the scene. Hierarchical occlusion maps and several other methods [4,12,18] support non-conservative culling which speeds up the computation but which does not guarantee that all visible objects are drawn.

A simple method to test the occlusion of a bounding volume with available hardware accelerated occlusion queries [16,28] is to rasterize the whole bounding volume without modifying any buffer and querying whether any fragment passed the z-test. This can also be done for several parts of the image in parallel [3] to increase performance.

3 Lazy Occlusion Grid

The grid allows to determine if a bounding volume is occluded by already drawn objects or if it is potentially visible. Each cell of the grid stores a state that represents if the cell´s image area is occluded or not (see section 3.1 and 3.2). The grid is based upon a conventional z-buffer or an occlusion-buffer where a single bit per pixel shows if that pixel is free or occluded. On systems where the graphics hardware supports occlusion queries at pixel-level or fast reading access to the z-buffer (see section 5) the hardware z-buffer will be used for that. On systems without these graphics hardware capabilities a conventional software z-buffer or occlusion-buffer must be used in parallel to the hardware z-buffer to do the pixel-level occlusion queries.

Occlusion culling of the scene´s objects with the lazy occlusion grid works as follows:

- Test if the bounding volume of an object is occluded or potentially visible (see section 3.1 and 3.2).
- If this test decides that the bounding volume is potentially visible then the object is drawn conventionally with the z-buffer hardware, otherwise it is culled. If a software occlusion-buffer or z-buffer is used for the pixel-level occlusion queries then the object is also drawn (per software) into this software buffer.
- After an object has been drawn this way all cells in its bounding volume´s image area are being marked as outdated so that the next occlusion test that queries the occlusion state of one of these cells knows that it must update the occlusion state of this cell with a pixel-level occlusion query (see section 3.1 and 3.2).

In a software implementation the occlusion-buffer has the advantage that it is faster than a software z-buffer because it only has to set and test a boolean value per pixel instead of calculating and comparing the pixels´ z-values.

The z-buffer variant of the grid has the advantage that the objects in the scene can be processed in an arbitrary approximative front to back order. The occlusion-buffer variant is used with a special approximative front to back sorting which guarantees that the occlusion test of a bounding volume is done before objects are drawn that are not completely in front of the bounding volume (see section 4). It does not need an exact front to back sorting of the primitives because exact visibility is solved by drawing the primitives with the z-buffer hardware. On systems where graphics hardware supports the pixel-level query, which is required for the occlusion-buffer variant, this variant can be implemented solely with the hardware z-buffer and does not need a software occlusion-buffer, because the z-comparison can be used instead of testing the single bit per pixel in the occlusion-buffer (see section 5).

The optimal number of pixels per cell that gives the best overall-performance is system-dependent and can easily be determined by testing typical scenes of the desired application with different numbers of pixels per cell.

3.1 Occlusion-Buffer Variant

In the occlusion-buffer variant of the grid the state which is stored in each cell of the grid can be

- completely free (completely unoccluded)
- partially free (some of its pixels are free and some are occluded)
- full (completely occluded, represented by a flag that overrides the other states to avoid that a full cell is being set to outdated)
- outdated (something has been drawn into the cell´s area and it has not been tested yet if the cell is occluded now)

Initially (before any object is drawn or any bounding volume is tested) the occlusion-buffer is cleared and all cells are in completely free-state. The occlusion test which is outlined in fig. 2 is applied for occlusion culling of the scene´s objects as described above. In this occlusion test we distinguish the cells that intersect with the tested bounding volume (see fig. 1):

- A cell which is completely covered by the bounding volume is called an *internal cell* of the bounding volume.
- A cell which is only partially intersected by the bounding volume is called a *border cell* of the bounding volume.

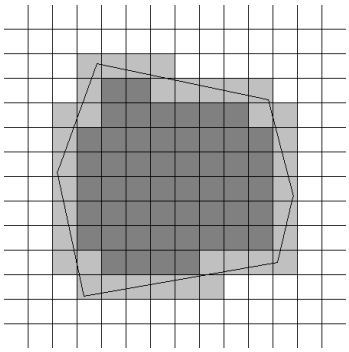

Fig. 1. Border cells (light grey) and internal cells (dark grey) of the tested bounding volume

This distinction is necessary to be able to recognize that a bounding volume is occluded behind a horizon (silhouette of another object). The problem herein is that in general a horizon does not completely occlude the cells it intersects, therefore these cells are partially free. If the tested bounding volume has such cells as border cells and if they would be handled like internal cells then the bounding volume would be rated as potentially visible because these cells are partially free. To avoid this the occlusion test has to make a pixel-level occlusion query in the intersection area of the bounding volume and its partially free border cell.

In the occlusion test we first try to determine potential visibility by using solely the cells´ states (in the …WithoutPixelquery functions). Only if this does not result in potential visibility we have to use the more expensive pixel-level

```
Bool isOccluded(bvol)
    if internalCellIsFreeWithoutPixelquery(bvol,internalPixelqueryList)
        return false
    if borderCellIsFreeWithoutPixelquery(bvol,borderPixelqueryList)
        return false
    if internalCellIsFreeWithPixelquery(internalPixelqueryList)
        return false
    if borderCellIsFreeWithPixelquery(bvol,borderPixelqueryList)
        return false
    return true

Bool internalCellIsFreeWithoutPixelquery(bvol,internalPixelqueryList)
    internalPixelqueryList=empty
    for all internal cells of bvol
        if not cell.full
            if cell.state=outdated
                add cell to internalPixelqueryList
            else    //completelyfree or partiallyfree
                return true
    return false

Bool borderCellIsFreeWithoutPixelquery(bvol,borderPixelqueryList)
    borderPixelqueryList=empty
    for all border cells of bvol
        if not cell.full
            if cell.state=completelyfree
                return true
            else    //partiallyfree or outdated
                add cell to borderPixelqueryList
    return false

Bool internalCellIsFreeWithPixelquery(internalPixelqueryList)
    for all cells in internalPixelqueryList
        if a pixel in the cell´s area is marked
                as not-occluded in the occlusion-buffer
            cell.state=partiallyfree
            return true
        else
            cell.full=true
    return false

Bool borderCellIsFreeWithPixelquery(bvol,borderPixelqueryList)
    for all cells in borderPixelqueryList
        if a pixel in the area of (cell ∩ bvol) is marked
                as not-occluded in the occlusion-buffer
            cell.state=partiallyfree
            return true
    return false
```

Fig. 2. Pseudocode of test if a bounding volume is potentially visible or occluded by already drawn objects inclusive lazy update of the cells´ states (occlusion-buffer variant of the grid)

```
Bool isOccluded(bvol)
    if internalCellIsFreeWithoutPixelquery(bvol,internalPixelqueryList)
        return false
    if borderCellIsFreeWithoutPixelquery(bvol,borderPixelqueryList)
        return false
    if internalCellIsFreeWithPixelquery(bvol,internalPixelqueryList)
        return false
    if borderCellIsFreeWithPixelquery(bvol,borderPixelqueryList)
        return false
    return true

Bool internalCellIsFreeWithoutPixelquery(bvol,internalPixelqueryList)
    internalPixelqueryList=empty
    for all internal cells of bvol
        if bvol.z_near≤cell.z_far
            if cell.outdated
                add cell to internalPixelqueryList
            else
                return true
    return false

Bool borderCellIsFreeWithoutPixelquery(bvol,borderPixelqueryList)
    borderPixelqueryList=empty
    for all border cells of bvol
        if cell.z_far=z_max
            return true
        else if bvol.z_near≤cell.z_far
            add cell to borderPixelqueryList
    return false

Bool internalCellIsFreeWithPixelquery(bvol,internalPixelqueryList)
    for all cells in internalPixelqueryList
        cell.outdated=false
        cell.z_far=max z of all pixels in cell´s area
        if bvol.z_near≤cell.z_far
            return true
    return false

Bool borderCellIsFreeWithPixelquery(bvol,borderPixelqueryList)
    for all cells in borderPixelqueryList
        if bvol.z_near≤max z of all pixels in the area of (cell ∩ bvol)
            return true
    return false
```

Fig. 3. Pseudocode of test if a bounding volume is potentially visible or occluded by already drawn objects inclusive lazy update of the cells´ states (z-buffer variant of the grid)

occlusion queries (in the …WithPixelquery functions). The lazy update of outdated cells is done in the …WithPixelquery functions.

3.2 Z-Buffer Variant

The z-buffer variant (fig. 3) is similar to the occlusion-buffer variant. The differences are that in the z-buffer variant each cell stores the farest z-value of its pixels (z_{far}) and a state-flag if the cell is outdated. Initially (before any object is drawn or any bounding volume is tested) all cells' z_{far} are set to z_{max} (which corresponds to completely free) and their outdated-flags are cleared. The z-buffer variant compares if the nearest z-value (z_{near}) of the bounding volume is greater than the cell´s z_{far} to determine if the bounding volume is occluded or potentially visible in the cell´s area. A outdated cell´s z_{far} is being updated with a pixel-level query that returns the farest z-value of all pixels in the cell´s area. This lazy update is done in the internalCellIsFreeWithPixelquery function.

4 Front to Back Traversal of a Bounding Volume Hierarchy

In the previous section we have described how to do occlusion culling for single objects (bounding volumes). What we need is to ensure that objects in the front are usually drawn first so that they can occlude objects behind them. This is accomplished by sorting the objects in an approximative front to back order. In a scene that contains a large number of objects it would be inefficient to do this sorting and the occlusion test for each object separately. To avoid this we use a bounding volume hierarchy for the scene that is traversed recursively. If a bounding volume is rated as occluded it can be culled without having to do the occlusion test for its sub-bounding volumes. Therefore a large occluded part of the scene can be culled at once with a single occlusion test. Any kind of bounding volume can be used, eg. octree, kd-tree, polyhedra or sphere. In our implementation which is described in section 7 we have used axis-aligned bounding boxes.

4.1 Z-Buffer Variant

The z-buffer variant of the lazy occlusion grid can be used with an arbitrary front to back sorting because this variant of the grid allows to draw objects before the occlusion test of bounding volumes which are (partially) in front of them is done. A simple front to back traversal for the z-buffer variant is outlined in fig. 4. It utilizes a list of bounding volumes which are sorted by their respective nearest z-value (z_{near}). Note that a heap could be used as well instead of the list. Note also that we distinguish between sub-objects (no bounding volumes) and sub-bounding volumes.

```
initialize lazy occlusion grid
initialize list with root bounding volume
while list is not empty
    bvol=frontmost bounding volume in list, remove it from list
    if bvol intersects view frustum
        if isOccluded(bvol)=false
            sort bvol´s sub-bounding volumes (if any) into list
            if bvol has objects as direct children
                draw objects that are direct children of bvol
                mark cells in bvol´s image area as outdated
```

Fig. 4. Pseudocode of a simple front to back traversal that incorporates occlusion culling with the z-buffer variant of the grid and hierarchical view frustum culling

4.2 Occlusion-Buffer Variant

The occlusion-buffer variant of the lazy occlusion grid is used with a special approximative front to back sorting which does the occlusion test of a bounding volume before objects are drawn that are not completely in front of the bounding volume. Otherwise these objects could falsely occlude the bounding volume or its sub-bounding volumes. The occlusion-buffer variant does not need an exact front to back sorting of the primitives because exact visibility is solved by drawing the primitives with the z-buffer hardware. This approximative front to back sorting is illustrated in fig. 5. The front to back traversal that realizes this sorting is outlined in fig. 6. It draws the frontmost object if it is completely in front of all bounding volumes that are not occlusion-tested yet and it tests the frontmost bounding volume for occlusion if no object is completely in front of it. To do this the traversal uses two lists (note that two heaps could be used instead of these lists, in our implementation we have used lists because usually these lists contain only a few elements):

- Bounding volumes that are not tested yet are sorted by their respective nearest z-value (z_{near}). Initially this test-list contains the root bounding volume.
- Bounding volumes that are already rated as potentially visible and that have objects as direct children are sorted by their respective farest z-value (z_{far}). Initially this draw-list is empty. In fig. 5 the objects themselves are shown instead of these bounding volumes for sake of simplicity of the illustration.

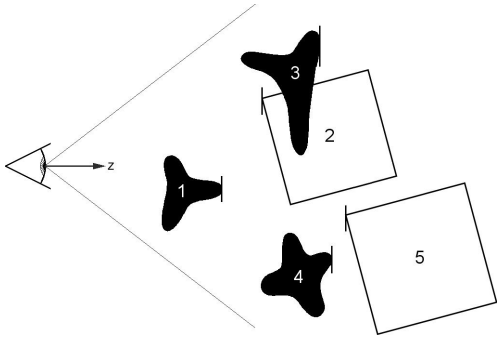

Fig. 5. Front to back sorting for occlusion buffer variant of the grid: z_{near}-sorted bounding volumes (white) are occlusion-tested before z_{far}-sorted objects (black) that are not completely in front of them are drawn. z_{near}/z_{far} is marked at each bounding volume/object, sub-objects and sub-bounding volumes of the bounding volumes are not shown.

```
initialize lazy occlusion grid
initialize test-list with root bounding volume if it
        intersects view frustum, else test-list=empty
draw-list=empty
while test-list or draw-list is not empty
    bvol=bounding volume with smallest z_near from test-list or
          bounding volume with smallest z_far from draw-list
          (depends on if z_near or z_far is smaller), remove it from its list
    if bvol is from test-list
        if isOccluded(bvol)=false
            for each of bvol´s sub-bounding volumes
                if sub-bounding volume intersects view frustum
                    sort sub-bounding volume into test-list
            if bvol has objects as direct children
                sort bvol into draw-list
    else   //bvol is from draw-list
        draw objects that are direct children of bvol
        mark cells in bvol´s image area as outdated
```

Fig. 6. Pseudocode of front to back traversal that incorporates occlusion culling with the occlusion-buffer variant of the grid and hierarchical view frustum culling

5 Hardware Acceleration

The most suited hardware-support is given on systems where pixel-level occlusion queries are implemented in hardware. The occlusion-buffer variant of the grid requires a query that tests if all pixels in the requested area are occluded (see section 3.1). For a hardware implementation of this query [16,28] it means comparing if the pixels´ z-values in the hardware z-buffer are less than z_{max} (which corresponds to an unoccluded pixel). The cost of this query compared to the cost of drawing triangles varies between different hardware [29].

The z-buffer variant of the grid needs a query that returns the farest z-value of all pixels in the cell´s area [3] (see section 3.2). On systems where this kind of query is not implemented in

hardware (unfortunately this is commonly the case today) it is possible to use another kind of query with a modified version of the z-buffer variant. In this modified version the original query for the farest z-value is replaced with a function that uses a query whether any pixel of the bounding volume passes the z-test (is visible) [16,28] in the cell´s area. If this is true the function returns the cell´s old z_{far}-value otherwise it returns the bounding box´s z_{near}-value.

On systems where neither kind of query is implemented in hardware it is still possible to use the hardware z-buffer by reading it and doing the query in software. The speed of reading the hardware z-buffer varies significantly between different systems.

A software z-buffer or occlusion-buffer parallel to the hardware z-buffer is only necessary on systems where the hardware does not support pixel-level queries and reading of the hardware z-buffer is too slow.

6 Future Extensions

The possible future extensions of our basic method which are sketched in this section use different occlusion tests for special cases to increase overall-performance. Some of the extensions use heuristics to decide if these tests shall be used or not. Note that these extensions do not violate conservatism.

A possible extension is to quickly rate a cell as free (and therefore the bounding volume as potentially visible) instead of using a pixel-level query whenever the probability that the cell is occluded is smaller than a given threshold. This probability can be approximated by adding up the size of the image-areas of the primitives that were drawn into the cell which for reason of speed can be done without considering if these areas are intersecting.

Another possibility is to test whether a bounding volume's border cell is full whenever the area of intersection of the bounding volume with the cell is larger than a given threshold. In this case the overhead of querying the whole cell-area instead of querying only the smaller intersection-area is not so big and we have the the chance to detect that the cell is full.

Border cells could also be treated completely as internal cells if the image area of the bounding volume is larger than a given threshold, because the image position of the border of such a large bounding volume is often quite different to the image position of the real object´s border. In such a case occlusion of the object behind a horizon has to be determined with tighter sub-bounding volumes.

7 Implementation and Results

We have implemented and tested occlusion culling with the lazy occlusion grid on a PC with a 900 MHz Thunderbird CPU and a GeForce2 GTS graphics board under OpenGL. We had no access to graphics hardware that supports pixel-level occlusion queries therefore we used the occlusion-buffer variant of the grid as described in section 3.1 in combination with reading the hardware z-buffer and doing the pixel-level queries in software for the generation of our results. Reading the hardware z-buffer is done with the glReadPixels function. The size of the grid´s cells is 32x32 pixels per cell and has been determined heuristically as described in section 3. Of course on other systems the optimal cell-size may be different.

For our scenes we have used a hierarchy of axis-aligned bounding boxes. The image area of a bounding box is approximated by its bounding rectangle in the image. The bounding boxes hierarchy is traversed with the front to back traversal as described in section 4.2. Note that this traversal also incorporates hierarchical view frustum culling of the bounding boxes [22] which uses simple clipping of the bounding boxes´ polygons in software. The hierarchy of bounding boxes is built initially for the given set of primitive objects of the scene. In the forest scene (fig. 10-12) each tree is a primitive object with an own bounding box. In the city scene each triangle is a primitive object and the scene is hierarchically subdivided into bounding boxes until each bounding box contains no more than 1500 triangles. This generation of the bounding boxes hierarchy takes less than one second for our scenes.

The forest scene contains 1,694,426 triangles and the city scene contains 34,034,176 triangles. We tested both scenes with walkthroughs that were rendered

- with occlusion culling with the lazy occlusion grid (lazy update).
- with occlusion culling with the occlusion grid but with a busy update where everytime after an object has been drawn the occlusion states of all cells in its image area are being updated (similar to a hierarchical z-buffer with two pyramid-levels).
- without occlusion culling (but also with hierarchical view frustum culling).

The rendering times of these walkthroughs, the number of drawn triangles per frame (this means that they are sent to OpenGL, backface culling is done by OpenGL) and the number of pixel-level occlusion queries (glReadPixels) per frame are shown in fig. 7-9. The scenes were rendered at 640x480 as well as 1280x960 pixels to show to what extent the rendering time is affected by image resolution. Note that depending on scene and image resolution the average frame-rate with the lazy occlusion grid is 1.9 to 5.8 times higher than with the busy occlusion grid and 1.9 to 39.1 times higher than without occlusion culling. In the forest scene rendering with the busy occlusion grid is slower than without occlusion culling. We have measured that with our hardware 38-52% of the total rendering time is spent for the glReadPixels calls when we use the lazy occlusion grid and 54-69% when we use the busy occlusion grid.

		lazy occlusion grid/ busy occlusion grid	no occlusion culling
forest	640x480	7,009	147,969
forest	1280x960	7,510	147,969
city	640x480	11,981	1,964,918
city	1280x960	11,775	1,964,918

Fig. 7. Average number of drawn triangles (sent to OpenGL) per frame of walkthrough

		lazy occlusion grid	busy occlusion grid
forest	640x480	498	2,394
forest	1280x960	650	6,781
city	640x480	565	1,332
city	1280x960	937	4,227

Fig. 8. Average number of pixel-level occlusion queries per frame of walkthrough

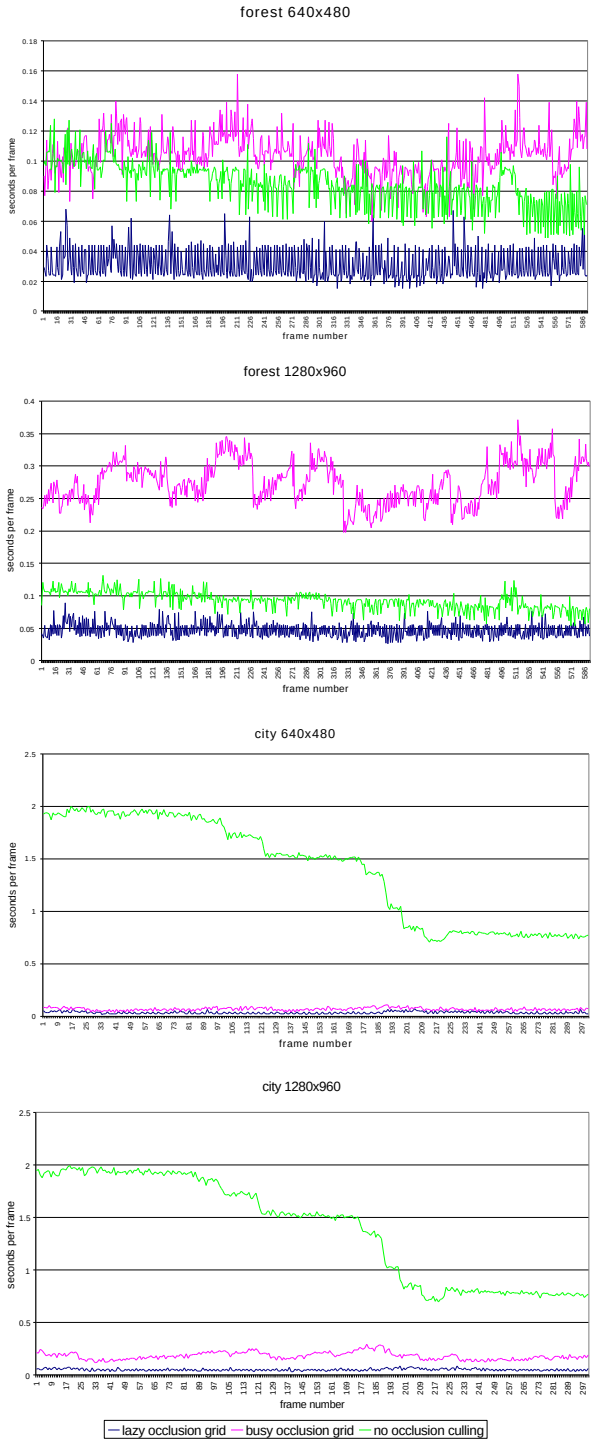

Fig. 9. Rendering time per frame of walkthrough

8 Conclusion

We have presented a conservative image-based occlusion culling technique which is capable of handling very complex general scenes at interactive frame-rates and that requires no time-expensive preprocessing. Our results show that significantly higher frame-rates can be achieved with the lazy update which is used in our method than with a busy update.

Future work includes utilization of temporal coherence and support of hardware that provides parallel pixel-level occlusion queries for improved performance.

Acknowledgments

This work is supported by the Austrian Science Fund (FWF) contract no. P13600-INF. Thanks to Michael Wimmer and Peter Wonka for the original version of the city model.

References

[1] M. Abrash. Inside Quake: Visible Surface Determination. *Dr. Dobb's Sourcebook* January/February 1996 pp. 41-45

[2] J. Airey, J. Rohlf and F. Brooks Jr. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *Symposium on Interactive 3D Graphics 90* pp. 41-50

[3] D. Bartz, M. Meißner and T. Hüttner. Extending Graphics Hardware For Occlusion Queries In OpenGL. *EUROGRAPHICS/SIGGRAPH workshop on graphics hardware 98* pp. 97-103

[4] D. Bartz, M. Meißner and T. Hüttner. OpenGL-assisted Occlusion Culling for Large Polygonal Models. *Computers & Graphics* 23 (1999) pp. 667-679

[5] J. Bittner, V. Havran and P. Slavík. Hierarchical Visibility Culling with Occlusion Trees. *Computer Graphics International 98* pp. 207-219

[6] D. Cohen-Or, G. Fibich, D. Halperin and E. Zadicario. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. *EUROGRAPHICS 98* pp. 243-253

[7] S. Coorg and S. Teller. A Spatially and Temporally Coherent Object Space Visibility Algorithm. MIT LCS Technical Report 546, 1996

[8] S. Coorg and S. Teller. Real-Time Occlusion Culling for Models with Large Occluders. *ACM symposium on interactive 3D graphics 97* pp. 83-90

[9] F. Durand, G. Drettakis and C. Puech. The 3D visibility complex: a new approach to the problems of accurate visibility. *EUROGRAPHICS workshop on rendering 96* pp. 245-256

[10] F. Durand, G. Drettakis and C. Puech. The Visibility Skeleton: A Powerful And Efficient Multi-Purpose Global Visibility Tool. *SIGGRAPH 97* pp. 89-100

[11] F. Durand, G. Drettakis, J. Thollot and C. Puech. Conservative Visibility Preprocessing using Extended Projections. *SIGGRAPH 2000* pp. 239-248

[12] C. Gotsman, O. Sudarsky and J. Fayman. Optimized occlusion culling using five-dimensional subdivision. *Computers & Graphics* 23 (1999) pp. 645-654

[13] N. Greene, M. Kass and G. Miller. Hierarchical Z-Buffer Visibility. *SIGGRAPH 93* pp. 231-238

[14] N. Greene and M. Kass. Error-Bounded Antialiased Rendering of Complex Environments. *SIGGRAPH 94* pp. 59-66

[15] N. Greene. Hierarchical Polygon Tiling with Coverage Masks. *SIGGRAPH 96* pp. 65-74

[16] Hewlett-Packard. OpenGL Implementation Guide. *www.hp.com/workstations/support/documentation/manuals/user_guides/graphics/opengl/ImpGuide/01_Overview.html#OcclusionExtension*, 2000

[17] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff and H. Zhang. Accelerated Occlusion Culling using Shadow Frusta. *ACM Symposium on Computational Geometry (SCG) ´97*

[18] J. Klosowski and C. Silva. The Prioritized-Layered Projection Algorithm for Visible Set Estimation. *IEEE transactions on visualization and computer graphics* vol. 6 no. 2 pp. 108-123, 2000

[19] V. Koltun, Y. Chrysanthou and D. Cohen-Or. Virtual Occluders: An Efficient Intermediate PVS representation. *EUROGRAPHICS workshop on rendering 2000* pp. 59-70

[20] S. Kumar, D. Manocha, W. Garrett and M. Lin. Hierarchical Back-Face Computation. *EUROGRAPHICS workshop on rendering 96* pp. 235-244

[21] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. *Symposium on Interactive 3D Graphics 95* pp. 105-106

[22] T. Möller and E. Haines. *Real-Time Rendering* pp. 192-200, 1999

[23] B. Naylor. Partitioning tree image representation and generation from 3d geometric models. *Graphics Interface ´92* pp. 201-212

[24] B. Naylor. Interactive Playing with Large Synthetic Environments. *Symposium on Interactive 3D Graphics 95* pp.107-108

[25] H. Plantinga and C. Dyer. Visibility, Occlusion and the Aspect Graph. *International Journal of Computer Vision* 5(2) 1990 pp. 137-160

[26] C. Saona-Vázquez, I. Navazo and P. Brunet. The Visibility Octree. A Data Structure for 3D Navigation. *Computers & Graphics* 23 (1999) pp. 635-643

[27] G. Schaufler, J. Dorsey, X. Decoret and F. Sillion. Conservative Volumetric Visibility with Occluder Fusion. *SIGGRAPH 2000* pp. 229-238

[28] N. Scott, D. Olsen and E. Gannett. An Overview of the VISUALIZE fx Graphics Accelerator Hardware. *Hewlett-Packard Journal* May 1998 pp. 28-34

[29] K. Severson. *VISUALIZE Workstation Graphics for Windows NT*. Hewlett-Packard product literature, 1999

[30] J. Stewart. Hierarchical Visibility in Terrains. *EUROGRAPHICS workshop on rendering 97* pp. 217-228

[31] O. Sudarsky and C. Gotsman. Dynamic Scene Occlusion Culling. *IEEE transactions on visualization & computer graphics* vol. 5 no. 1 pp. 217-223, 1999

[32] S. Teller and C. Séquin. Visibility Preprocessing For Interactive Walkthroughs. *SIGGRAPH 91* pp. 61-69

[33] Y. Wang, H. Bao and Q. Peng. Accelerated Walkthroughs of Virtual Environments Based on Visibility Preprocessing and Simplification. *EUROGRAPHICS 98* pp. 187-194

[34] M. Wimmer, M. Giegl and D. Schmalstieg. Fast Walkthroughs with Image Caches and Ray Casting. *EUROGRAPHICS workshop on virtual environments 99* pp. 73-84

[35] P. Wonka, M. Wimmer and D. Schmalstieg. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. *EUROGRAPHICS workshop on rendering 2000* pp. 71-82

[36] R. Yagel and W. Ray. Visibility Computation for Efficient Walkthrough of Complex Environments. *Presence* vol. 5 no. 1 pp. 45-60, 1996

[37] H. Zhang, D. Manocha, T. Hudson and K. Hoff III. Visibility Culling using Hierarchical Occlusion Maps. *SIGGRAPH 97* pp. 77-88

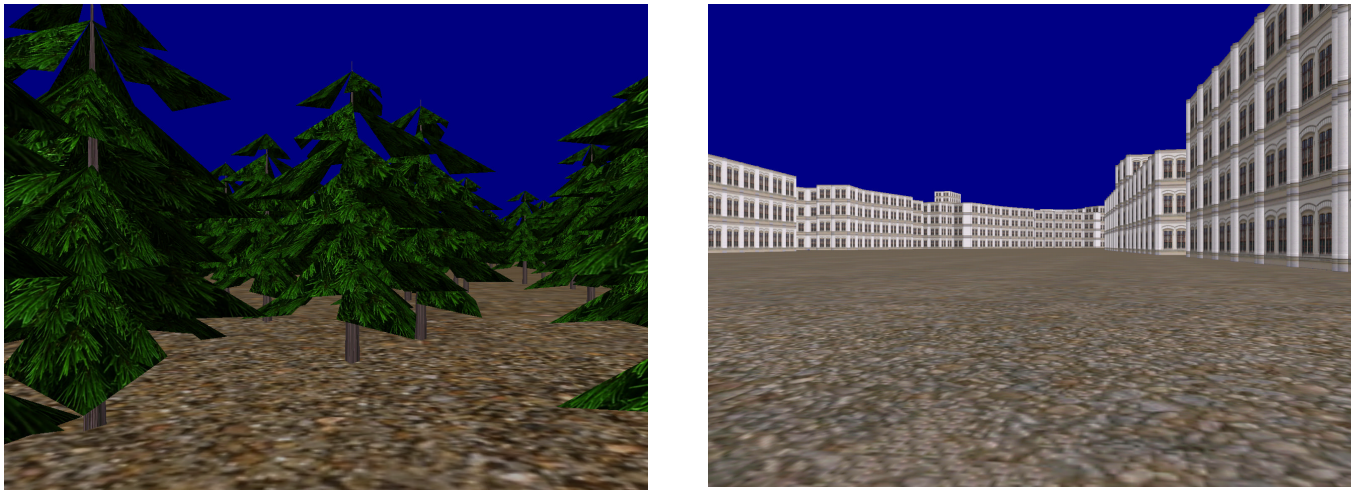

Fig. 10. Actually rendered image

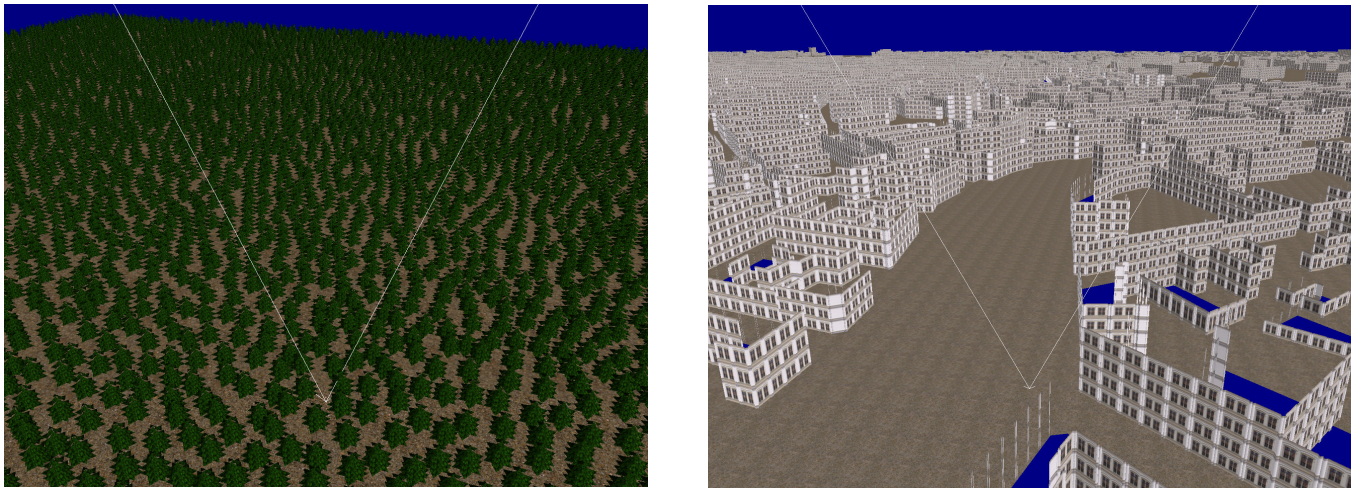

Fig. 11. View from above with view frustum from fig. 10

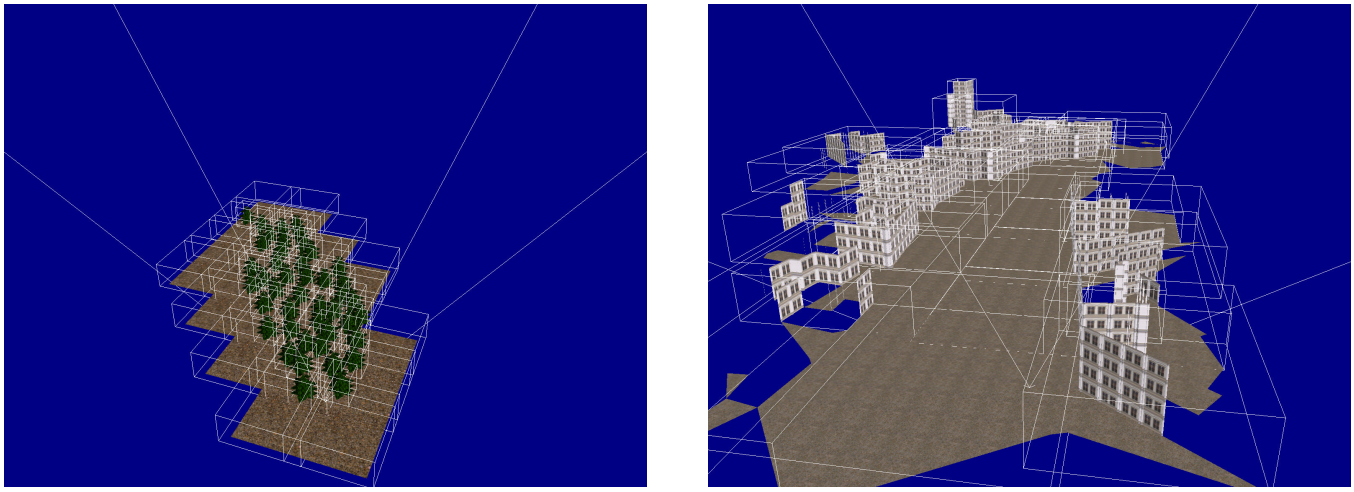

Fig. 12. Only the objects that are drawn (sent to OpenGL) in fig. 10, their leaf-bounding boxes and the view frustum