Level of Detail Based Occlusion Culling for Dynamic Scenes

Anselm Grundhöfer
Bauhaus University Weimar
Weimar, Germany
anselm.grundhoefer@medien.uni-weimar.de

Benjamin Brombach
Bauhaus University Weimar
Weimar, Germany
benjamin.brombach@medien.uni-weimar.de

Robert Scheibe
Bauhaus University Weimar
Weimar, Germany
robert.scheibe@medien.uni-weimar.de

Bernd Fröhlich
Bauhaus University Weimar
Weimar, Germany
bernd.froehlich@medien.uni-weimar.de

Abstract

This paper presents a non-conservative occlusion culling technique for dynamic scenes with animated or user-manipulated objects. We use a multi-pass algorithm, which decides the visibility based on low level of detail representations of the geometric models. Our approach makes efficient use of hardware support for occlusion queries and avoids stalling the graphics pipeline. We have tested our approach for large real-world models from different areas. Our results show that the algorithm performs well for medium complex scenes with 5 to 20 million triangles. with a very low number of hardly noticeable pixel errors, typically in the range of 0.02 percent of the total number of visible pixels.

Keywords

CR Categories and Subject Descriptors: I.3.3 [Picture/Image Generation]: Viewing Algorithms, Occlusion Culling; I.3.5 [Computational Geometry and Object Modelling]: Object Hierarchies; I.3.7 [Three-Dimensional Graphics and Realism]: Hidden Line/Surface Removal

Additional Keywords: Visibility and occlusion culling, large-scale data visualization

.

1. INTRODUCTION

Occlusion culling is an important acceleration technique for scenes with medium to high depth complexity. It may significantly improve the rendering performance for geometry- or fill-limited scenes. In particular the second case has become increasingly important due to the widespread use of complex pixel-based shading techniques. Most occlusion culling techniques were developed for static scenes and do not work well for dynamic environments. In addition, the combination of level of detail techniques and occlusion culling has not been explored much.

We present a non-conservative multi-pass approach, which uses low level of detail representations of the geometric models to decide the visibility for the higher resolution versions. Our approach computes the visibility on a per object basis using the hardware occlusion flags available with most recent graphics cards. In a pre-process large objects or objects with large bounding volumes are sliced into smaller sub-objects to provide a reasonable granularity for the occlusion decision. During run time, there are no additional data structures needed, except those for handling the level of detail rendering.

Figure 1: The algorithm allows manipulations of arbitrary objects in the scene without losing the occlusion culling ability. In the upper images you can see how the marked roof is translated by the user. The lower image shows that interior parts of the car are missing, since they would be occluded by the roof (not displayed). (Note: The original model does not have a second front seat!)

Our approach was developed for rendering medium to large geometric models in the range of tens of millions of triangles, such as the ones appearing in the automotive industry (Figure 1). In such application domains, users need to be able to look at the car from the inside and outside, open and close doors, or assemble an engine. These models are often organized in a scene graph structure and use level of detail representations to achieve reasonable frame rates. Our approach integrates well into such environments, since it does not require much infrastructure.

The main contribution of this paper is the development of an occlusion culling technique for dynamic scenes, which is based on level of detail representations of geometric models. Our approach makes efficient use of hardware support for occlusion queries and avoids stalling the graphics pipeline. We have tested our implementation for real-world models in the range of 5 to 20 million polygons and show that it performs better than alternative techniques. Our approach is non-conservative, but for our test scenes the fraction of incorrect pixels is typically around 0.02 percent, which is hardly noticeable for interactive applications.

2. RELATED WORK

We provide a short overview of occlusion culling approaches relevant to our work and the generation of occlusion preserving level of detail representations.

2.1. Occlusion culling and visibility culling

In the ideal case, only visible objects should be sent down the graphics pipeline to minimize the required bandwidth, the number of vertex operations, and the fill rate requirements. Generally, invisible objects can be classified into three categories:

1. Objects outside the viewing frustum.
2. Objects which are projected to an area less than one pixel in image space.
3. Objects occluded by other objects.

Removing occluded objects is the most challenging task. It requires knowledge about the occlusion relationships of objects in the scene and needs to identify the potentially visible set (PVS) of objects in the scene.

A lot of research has been done in this field, leading to many different occlusion culling approaches. [Cohen-Or et al. 2003; Hey and Purgathofer 2001; Bittner and Wonka 2003] provide an overview and a detailed discussion of these methods.

Occlusion culling algorithms can be classified into different categories: off-line and on-line, image and object space, and conservative or non-conservative. Another classification decides if the visibility is computed from a certain point or for a region. While off-line approaches need complex data structures to keep information about the occlusion relationships in the scene, on-line approaches need some additional computation to determine occlusion relationships for each frame. Object space algorithms calculate in most cases the PVS of all objects using spatial bounding volume hierarchies. Many algorithms are developed for region based visibility calculations, which estimates a very conservative PVS [Durand et al. 2000].

A lot of work has been done for pre-processed visibility calculations as well as for on the fly occlusion culling. Some algorithms, which use pre-processing, subdivide the static scene into grid elements and calculate the PVS for every grid cell [Saona-Vázquez et al. 1999; Wang et al. 1998]. During rendering only the objects in the current PVS have to be rendered, so no additional per-frame overhead is needed.

On the fly occlusion culling methods have the advantage that the PVS can be estimated for the current viewpoint [El-Sana et al. 01; Greene et al. 1993; Plate et al. 2004; Zhang et al. 97]. Some approaches have the ability to handle dynamic scenes, but an overhead is incurred due to the additional calculations for each frame. Some algorithms also employ a hybrid approach using pre-processed data structures and per frame occlusion culling [Hillesland et al. 2002; Bittner et al. 2004].

A lot of algorithms which use the hardware acceleration of modern graphic cards have been developed. While earlier algorithms use the depth and stencil buffer, nowadays graphic cards offer a hardware accelerated occlusion query. This feature allows us to render objects and to query if the object is visible. The original `HP_occlusion_test`, was able to send one occlusion query and the result had to be retrieved immediately. The latest extension is able to send many queries before retrieving the results, which avoids stalling the graphics pipeline. In addition the query returns the number of visible pixels in the view port instead of just a Boolean value. This extension was introduced by NVIDIA, called `NV_occlusion_query` and is now an official OpenGL ARB extension.

2.2. Occlusion preserving level of detail

Many occlusion culling approaches use simplified versions of the original geometry for calculating occlusions. A decimated model must lie "inside" the original model to guarantee correct and conservative occlusion.

[Law and Tan 1999] use virtual occluders, which are simplified versions of the original model. Their edge correction moves all the vertices and edges of the reduced model inside the boundary of the original model. With their approach it is possible to generate highly decimated occlusion preserving models because the virtual occluders never occlude other objects than the original model would. Thus the virtual models may be geometrically very different from the original but the decimated models are only used for visibility testing. For the final rendering different level of details are used. This approach works well for solid models, but CAD models are often sets of surface patches.

[Germs and Jansen 2001] present an algorithm for generating simplified facades from complex models for urban walkthrough applications. Their approach is based on the observation that buildings are often 2.5D structure. Most buildings can be described by a 2D contour, called footprint, and an associated height. A building itself consists of stacked blocks. The goal of this approach is to find parts of the building that have the same height and encapsulate them into one block. Every block has its own footprint and height. While going along the z-axis (vertical), different heights can be found. This is called z-level identification. For every z-level a footprint is created. With the set of footprints and height levels the facade of the high detailed building is generated. The facade is completely inside the original model and thus the technique is conservative.

We have not worked on building good and correct occluders for our occlusion culling approach, since it is a wide field of research and was not the focus of this paper. We use an existing edge collapse algorithm, similar to [Hoppe et al. 1993], to generate the levels of detail for our models. However, our approach would directly benefit from occlusion preserving level of detail representations.

3. ALGORITHM

Our approach combines LOD-based rendering with an efficient multi-pass occlusion culling algorithm. The LODs are generated in an off-line pre-processing step as well as the slicing of the high polygon count objects. The actual rendering is done by a three pass algorithm which efficiently identifies occluded objects during the first two passes. The third pass renders the potentially visible set.

3.1. Pre-processing

Similar to standard LOD-based rendering systems we need to create level of detail representations of our objects. Low level of detail representations are used to determine the occlusion relationships in the scene. For the final rendering pass more detailed representations are selected based on a given target frame rate. The decimation approach affects the quality of our algorithm directly, since simplified models should keep the occlusion relationship as much as possible.

Our occlusion culling approach determines visibility on a per object basis. If the scene consists only of a small number of large objects, our algorithm would not perform well. In general for scenes with very large objects we apply a geometry splitting step, which divides the model into parts of approximately equal size. This object split has to be applied to each level of detail of a model. This step improves the granularity of our occlusion culling approach significantly. It also adds a small number of vertices, but the improved culling efficiency outweighs this overhead in general.

3.2. Multi-pass rendering

Our algorithm consist of three rendering passes, Listing 1 provides an overview in pseudo code notation. An initial view frustum culling pass removes all objects outside the viewing frustum. This can be done in software or hardware or skipped if it is efficiently implemented on the graphics card. The LOD pre-processing step computes the required LODs for each object. The LODs for the first two rendering passes are lower than the ones used for the final rendering pass if the target frame rate is appropriately selected. All three passes make use of the split object representations computed in the pre-processing step to achieve a good granularity for the occlusion culling approach.

The next passes make use of the OpenGL occlusion culling extension which counts the number of visible pixels of geometric primitives. Occlusion query objects have to be generated and activated for each geometric object. After the geometry is drawn, the occlusion query has to be disabled and the number of visible pixels can be queried. Since this query requires that the object has been rasterized, it is useful to send multiple queries before asking for the results.

In the first rendering pass the depth buffer is cleared and the objects are rendered with low levels of detail into the z-buffer with activated depth testing. During this pass, colour buffer writes and rendering features like lighting, shading,

```
// view frustum culling
do view frustum culling and label all objects lying
in view frustum
(all further passes treat only these marked objects)

// LOD pre-processing:
calculate distances of each object to camera
determine LODs for final rendering and occlusion
culling
store both LODs for each object

// 1st pass:
Disable colour mask, texturing, lighting and frag-
ment/vertex programs;
clear z-buffer;
for each object
  render occlusion culling LOD into depth buffer;

// 2nd pass:
disable Depth Mask;
set glDepthFunc to GL_LEQUAL
for each object
  begin occlusion query;
  render occlusion culling LOD
  end occlusion query;

// 3rd pass:
enable colour mask, texturing, lighting and frag-
ment/vertex programs;
enable depth mask;
clear depth and colour buffer
for each object
  get occlusion query result
  if(number of visible pixels > threshold)
    render in final rendering LOD
```

Listing 1. LOD-based multi-pass occlusion culling.

Texturing, and vertex/fragment shaders are disabled. After this pass the depth buffer contains the depth information for the whole scene.

In the second pass all the objects are drawn again but now depth buffer writes are disabled as well. The depth testing function has to be set to GL_LEQUAL (instead of the default GL_LESS state) to count only the pixels of visible objects. With each object now an occlusion query is send. Only finally visible pixels are counted, since the z-buffer of the first pass is used.

In the last pass colour and depth buffer writes are enabled and the depth function is turned back to its default state. Other states like texturing and lighting or vertex and fragment shaders can be enabled if needed.

Before rendering an object with a higher LOD, which is selected based on a distance calculation, we retrieve the result of the corresponding occlusion query. If the query result is zero, no pixels of this object were visible during the second rendering pass and the object can be skipped in any case. Instead of skipping only objects with zero visible pixels, we use a threshold value typically in the range of zero to 100 pixels. The threshold affects the "conservatism" of our approach.

Translucent objects cannot be treated in the described way because they do not occlude the objects lying behind them. Therefore these objects have to be excluded from the first (depth buffer filling) pass. While they are not able to occlude other objects they still can be occluded by opaque

objects. This means we can still include them in the second pass. For correct alpha blending, the translucent objects are rendered after the opaque objects in back to front order.

3.3. Alternative approaches

For comparisons, we implemented two more algorithms. The first algorithm (Listing 2) is very similar to the one described in [Fernando and Pharr 2005], but it makes use of our low LODs to support more efficient culling. This approach is highly non-conservative, since it uses the occlusions from frame n-1 to determine the visibility for frame n. We refer to this approach as "multi-frame occlusion culling. For the very first frame all objects are labelled as visible and they are drawn with high resolution LODs. Then all objects' bounding boxes are drawn into the depth buffer and occlusion queries are sent for each object. The results are queried during the next frame and only the visible objects are rendered using the correct LOD. Afterwards all the other objects are rendered only into the depth buffer with low LODs. Now the depth buffer is filled with the z-values from all objects - some objects in high resolution, some in low resolution. Then the bounding boxes of all objects including occlusion queries are rendered, which is very similar to the second pass of our algorithm. In this case we cannot use the "occlusion culling" LOD because of the fact that the depth buffer contains some objects rendered with high LODs, so we cannot guarantee the visibility of the decimated objects. Please note that the results of the occlusion queries for the bounding boxes are only evaluated during the next frame which explains the speedup compared to our algorithm.

```
// LOD pre-processing:
calculate distances of each marked object to camera
determine LODs for final rendering and occlusion
culling
store both LODs for each object

// 1st pass:
enable colour mask, texturing, lighting and frag-
ment/vertex programs;
enable Depth Mask;
clear depth and colour buffer
for each object
  get occlusion query result from last frame
  if(number of visible pixels > threshold)
    render in final rendering LOD
  else
    label object as not rendered

// 2nd pass:
disable colour mask, texturing, lighting and frag-
ment/vertex programs;
for each labeled object (which was not rendered in
the first pass)
  render occlusion culling LOD in depth buffer;

// 3rd pass:
disable depth mask;
for each object
  begin occlusion query;
  render bounding box into depth buffer;
  end occlusion query;
```

Listing 2. Multi-frame occlusion culling.

We also implemented a simple "stop and wait" algorithm, which renders the scene in front to back order. For each object – except for the front most object – a bounding box is drawn with an occlusion query. The result is fetched immediately afterwards and if the box was visible the corresponding object is drawn. See Listing 3 for the pseudo code of the algorithm. This algorithm needs almost no overhead (only the bounding boxes) and generates conservative results, but due to the GPU stalling with each occlusion query the performance increase can be quite limited. We did not implement a hierarchical version, since it is difficult to deal efficiently with dynamic scenarios.

```
// LOD pre-processing:
calculate distances of each object to camera
determine LODs for final rendering
store LOD for each object
sort all objects in front to back order

// rendering:
clear depth and colour buffer
draw the nearest object in final rendering LOD

for all other objects:
  disable colour/depth mask, texturing, lighting and
  fragment/vertex programs;
  begin occlusion query;
    render bounding box into depth buffer;
  end occlusion query;

  get occlusion query result
  if(number of visible pixels > threshold)
    enable Colour/Depth Mask, Texturing, lighting
    and fragment/vertex programs;
    render for the final rendering LOD
```

Listing 3. Simple stop-and-wait occlusion culling.

4. RESULTS

We have evaluated the three different occlusion culling implementations for three different scenarios.

4.1. Scenarios

Our first scene consists of a large number of Utah Teapots (see Figure 9). The second scene consists of four Volkswagen Beetle models, which serve as our real world example (Figure 1, 2 and 3). The third test object was the power plant model from the University of North Carolina. The models were decimated into three levels of detail in addition to the full resolution model. The two lowest LODs were used for occlusion culling, whereas the three highest LODs were used for the third rendering pass. The Beetle and the power plant model were split into tiles to improve the occlusion culling granularity. See table 1 for the detailed information about the scene complexity.

	Teapots	Powerplant	Beetle
Triangles per LOD	1. 11.848.815 2. 1.213.440 3. 568.800 4. 310.470	1. 8.133.064 2. 2.273.369 3. 1.437.965 4. 958.695	1. 3.403.818 2. 1.283.723 3. 454.654 4. 190.985
# of objects	1.185	854	337

Table 1. Scenarios overview

4.2. Object splitting

We split objects with large bounding boxes and objects with a large number of triangles into smaller parts to provide a reasonable granularity for our approach. Large objects are split such that each tile's edge length is shorter than 10 percent of the corresponding edge of the scene bounding box. Objects with more than 300.000 vertices are split even if they are smaller in size than the 10 percent of the scene bounding box. There is only a single split step, which subdivides the objects' bounding boxes along all dimensions resulting in approximately cube shaped parts. These parameters were empirically chosen and worked well for our scenarios.

4.3. Pixel error measurement

For the comparison of the different rendering algorithms we created key frame animations for our test scenes and measured the frame rates with and without occlusion culling. For determining the pixel error rate of our algorithm, each frame is rendered twice – with and without occlusion culling. These two frames are copied into textures and they are compared by a fragment program to count the errors efficiently. Each object is tagged by a unique colour (see Figure 2) and the rendering is done without lighting or shading to guarantee an accurate comparison of the pixel colour values.

A full screen quad with texture coordinates is rendered for including all the pixels into the comparison. The fragment shader does a texture lookup into both textures which contain the two previously generated images. It discards all fragments which are equal in both textures and keeps only those with different colours. Using the occlusion culling extension we can efficiently count all the rendered pixels which is equivalent to the pixel error rate of our algorithm.

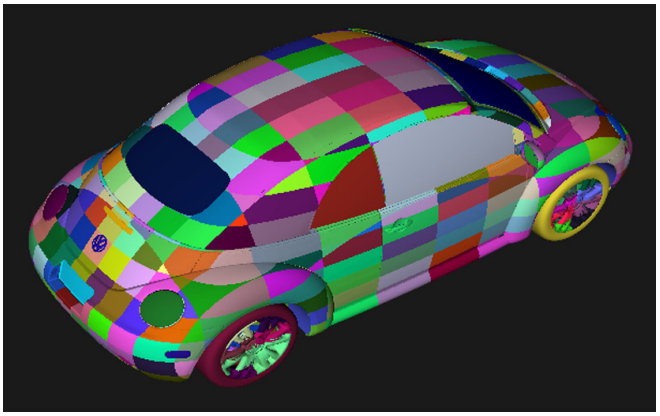


Figure 2: Object colour ID rendering. All objects are labeled by different colours. This mode was chosen to identify the pixel errors of our algorithm.

4.4. Low level of detail representations

The use of low level of detail representations for the occlusion tests is only an approximation of the actual visibility of the high resolution objects. We have measured the number of actually rendered faces using our approach compared to the number of faces if the highest resolution LODs would be used for the occlusion test. We did not measure the actual number of visible triangles, just the number of triangles drawn. The results indicate that the low resolution version tend to a conservative behaviour since the number of drawn triangles for the final pass is always larger than the number required if the high resolution LODs would be used for the occlusion test. These results are heavily dependent on the quality of the LODs, the pixel threshold as well as the granularity of the (split) objects in the scene. However, the results suggest that the use of low level representations for the occlusion tests, which are generated by a standard decimation algorithm, behave good-natured.

35	33	23	41
32	28	22	38

Figure 3: The upper line shows the percentage of faces drawn for the final pass for four different views using low resolution LODs for the occlusion test. The lower line shows the percentage of drawn faces using only the highest LOD for the occlusion test.

4.6. Fixed frame rate rendering

We have also experimented with constant frame rate rendering by selecting the LODs for all the objects such that a given number of triangles was not exceeded. This resulted in constant frame rates for geometry limited scenes. If pixel fill during the third rendering pass is the limitation, our algorithm will perform as fast as the hardware allows, but it cannot be adjusted to a certain frame rate.

When comparing our algorithm to a standard LOD algorithm without occlusion culling, we can see that it renders with slightly lower frame rates but with much increased rendering quality (see Figure 4).

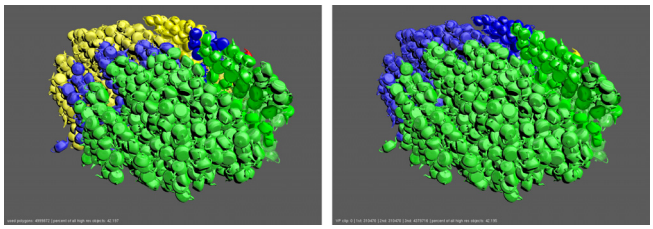


Figure 4: Fixed frame rate scene quality comparison. Different LODs are coloured from green (high res), blue, red to yellow (lowest LOD). The overall scene quality increases if we use our multi-pass algorithm (right) compared to standard fixed frame-rate rendering (left).

4.5. Implementation

Our implementation uses a QT/C++ framework for the interface and the data structure implementation. The rendering API is OpenGL. The software runs under Linux and Microsoft Windows operating systems. All tests were done under Fedora Core 3 Linux on an Intel dual Xeon 3.2 GHz

workstation with 4 GBs of RAM and a NVIDIA Quadro FX 3400 graphics card with driver version 76.67.

Our algorithm was compared to standard LOD based rendering without occlusion culling (No OC), but also to the multi-frame approach and the stop-and-wait implementation. In addition, we compare two versions for each scenario – simple colour-per-vertex rendering (geometry limited) and high quality per-pixel shading (fill-limited). Figures 5, 6 and 7 show a comparison of the averaged frames rates of a fly through for our three models at 1024x768 resolution and for a pixel threshold of 30 pixels.

For most cases our algorithm performs notably faster than the stop-and-wait implementation, while the multi-frame approach outperforms our approach slightly for the colour scenario. If fragment shaders are used, the multi-frame approach performs worse than our approach because of the bounding boxes which are used for the occlusion test. If we compare the generated pixel errors between our multi-pass algorithm and the multi-frame algorithm (Figure 8 and 9) we see many more pixel errors for the multi-frame algorithm due to the one frame delay for the application of the occlusion query results.

In general our algorithm renders more triangles than the stop-and-wait approach, because of the triangles required for rendering the first two passes. Nevertheless our approach performs better due to the overhead produced by extensive GL-state switching and the GPU stalling with the stop-and-wait implementation.

Figure 10 shows one frame of our animation. The left image is generated with standard LOD rendering while the right image is generated with our 3-pass algorithm and a pixel threshold of 20 pixels. The black and white image below shows the differences of the two images. The errors are very small and mostly not noticeable.

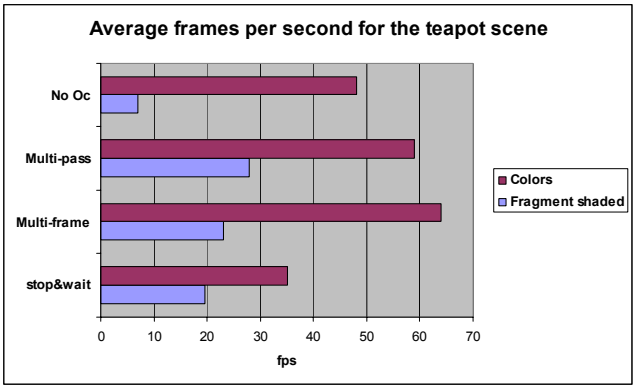


Figure 5: Teapot scene: If only vertex colours are rendered, the speedup is much smaller and the stop-and-wait algorithm is even slower than rendering without occlusion culling because of the GPU stalling and GL-state switching problems. Our algorithm performs even better than the multi-frame algorithm when using fragment shaders for illumination due to the more precise detection of culled objects.

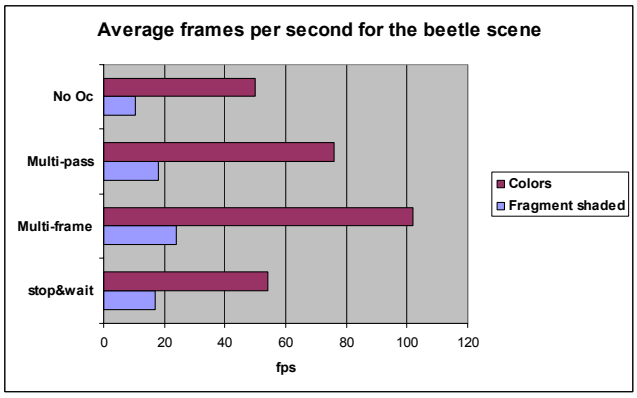


Figure 6: Single Beetle scene: Since the Beetle model has only around 4 million triangles, the stop and wait algorithm performs almost as well as our algorithm because of the large number of triangles used for the two initial passes.

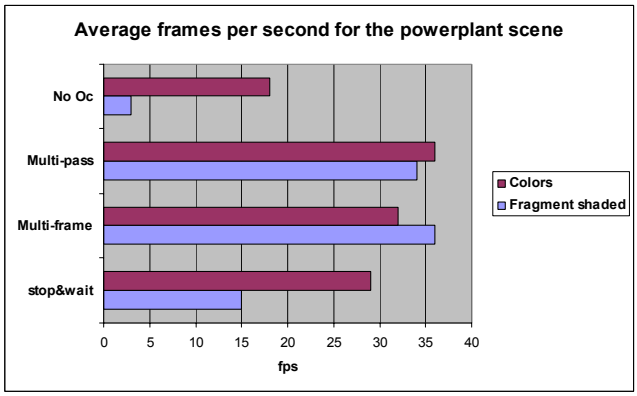


Figure 7: Power plant scene: The results are similar to the teapot scene. The fragment shading is very basic.

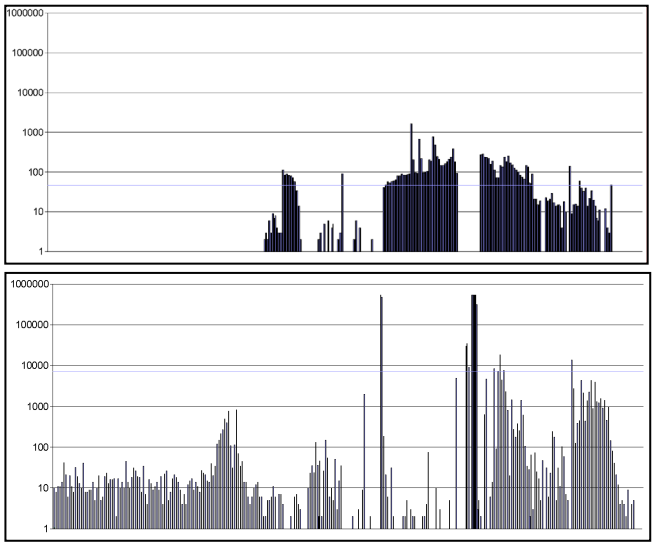
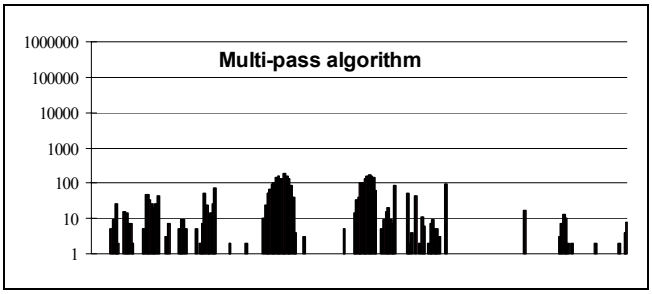


Figure 8: Pixel error comparison for the teapot scene: Both diagrams were produced using a threshold of 30 pixels. The upper diagram shows the pixel errors for the multi-pass algorithm, which has an average error of 47 pixels. The lower one shows the multi-frame approach which leads to very high errors for fast changing scenes. Here the average error is 7043 pixels.



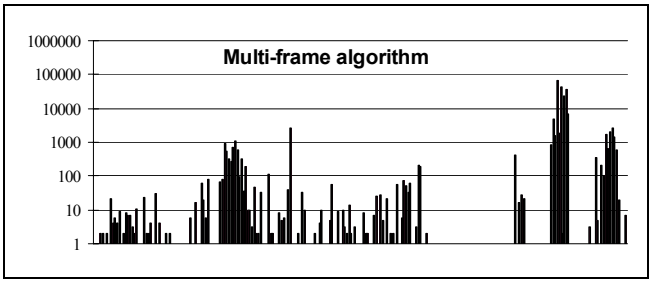


Figure 9: Pixel error comparison for the beetle scene: The threshold is 30 pixels. The upper diagram shows the pixel errors for our algorithm with an average error of 14 pixels. The lower one shows the multi-frame approach with an average error of 403 pixels.

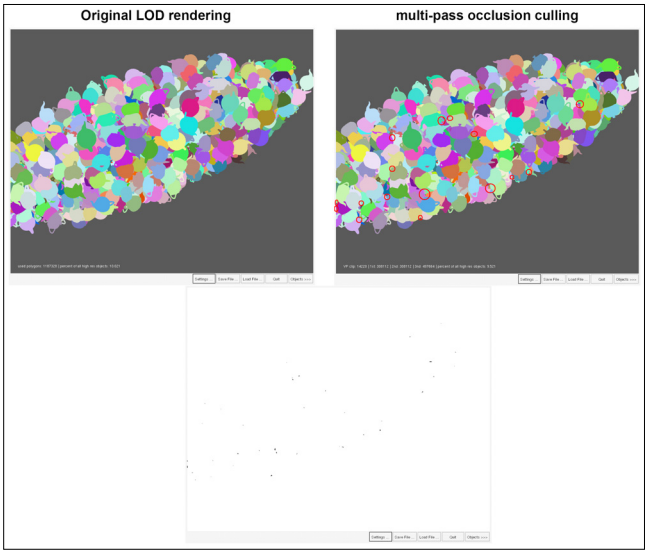


Figure 10: Pixel error visualization. The left upper frame was rendered with a standard LOD algorithm using the same LOD settings as for our multi-pass algorithm. The upper right image was rendered by our algorithm using a threshold of 20 pixels per occlusion query. The overall pixel errors in this frame are 148 (out of 906948) which is 0.0163%. (See the red circles in the right image). The incorrect pixels are displayed as black points in the lower image. They appear in small clusters due to the pixel threshold per object.

5. DISCUSSION

Our algorithm offers a lot of possibilities to choose parameters for the trade-off between frame rate and visual quality. The most relevant factors are:

Level of detail selection:
Different LODs are selected for the final render pass and the occlusion culling passes. Choosing a lower LOD for the final render pass reduces the complete visual quality of the scene but allows adjusting an adequate frame rate for complex scenes. Our approach also makes use of LODs for occlusion culling. It is assumed that high detailed LODs have a better occlusion preserving characteristic. Low resolution models are less occlusion preserving and may result in better performance but also result in more pixel errors due to incorrect occlusion query results.

Distance calculation method for the LOD selection.

Pixel threshold:
This value allows more objects to be culled, but increases the number of incorrect pixels for the final image. If the LODs are of very bad quality, the overall pixel error can add up to a very high value due to the fact that this threshold is applied for each object.

Pre-render view port:
Choosing a smaller view port for the first two passes is equivalent to increasing the pixel threshold quadratically, but it reduces the required fill rate for the first render passes.

For high performance rendering, we generated triangle strips for each object and stored the data in Vertex Buffer Objects (VBO). With the current NVIDIA graphics driver (76.67) we noticed that there was an enormous performance drop when the number of objects increased. It seemed that the driver was not able to efficiently manage many (>2000) VBOs with a small number of vertices. We realized that although VBOs are able to generate a higher performance than vertex arrays or display lists they are not yet useful for our task. Furthermore, the system behaves unstably – we noticed random crashes. Because of these problems we switched to vertex arrays and display lists which gave us a much higher overall performance without any stability problems. To optimize the vertex transfer, two different display lists were needed: one with normals and texture coordinates for final rendering pass and one containing only vertex data for occlusion culling. This increased the memory requirements for the level of detail representations which were used for all three rendering passes, because the objects have to be stored in two display lists. While this increases the overall memory requirements, it significantly speeds up the pre-rendering process, because unnecessary data like texture coordinates and normals do not have to be transferred and transformed.

The performance was optimized by disabling all non-necessary features like shading and texturing for the first two passes. The fixed function pipeline of current NVIDIA

graphics hardware offers features like double speed z-only rendering if colour buffer rendering is disabled and early z-optimizations (z-cull).

6. CONCLUSIONS AND FUTURE WORK

We presented a new algorithm, which combines level of detail rendering and occlusion culling in an efficient way. Our approach works well for interactive scenes with dynamic objects and performs better than classical stop-and-wait approaches. Our algorithm generates a significant speedup compared to standard LOD-based rendering without introducing significant quality tradeoffs. For applications which are already using different level-of-detail representations such as CAD rendering systems or computer games, major changes in the implementation are not required.

The generation of occlusion preserving and effective LODs for non-solid models is a non-trivial problem due to the fact there is no defined inside or outside for such models. However such representations could speed up the initial rendering passes significantly, since smaller models could be used. In addition the scalability of our approach would be improved and it would be turned into a conservative technique. Without occlusion preserving LODs, there is a need for measures that predict potential occlusion errors.

The combination of our technique with occlusion culling approaches for static scenes could work well for a large class of applications, since most environments consist of a static model and a limited number of freely moving objects.

ACKNOWLEDGMENTS

We thank Volkswagen AG for providing the Beetle model.

REFERENCES

BITTNER J. and WONKA P. 2003. Visibility in computer graphics. In *Journal of Environment and Planning B: Planning and Design*, volume 30, number 5, 729–756.

BITTNER J., WIMMER M., PIRINGER H. and PURGATHOFER W. Coherent 2004. Hierarchical Culling: Hardware Occlusion Queries made useful. *Proceedings of Eurographics 2004*. Computer Graphics Forum, 615-624.

COHEN-OR D., CHRYSANTHOU Y., SILVA C. and DURAND F. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 412-431.

DURAND F., DRETTAKIS G., THOLLOT J. and PUECH C. 2000. Conservative Visibility Pre-processing using Extended Projections, *Proceedings of ACM Siggraph 2000*, 239-248.

FERNANDO R. and PHARR M. 2005. GPU Gems 2, Programming Techniques for High-Performance Graphic, Addison Wesley Professional, 47-68.

GERMS R. and JANSEN F. W. 2001. Geometric Simplification for efficient occlusion culling in urban scenes, *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2001,* 291-298.

GREENE N., KASS M. and MILLER G. 1993, Hierarchical z-buffer visibility, *Proceedings of ACM SIGGRAPH 1993*, 231-238.

HILLESLAND K., SALOMON B., LASTRA A. and MANOCHA D. 2002. *Fast and Simple Occlusion Culling using hardware based depth queries*, Technical Report TR02-039, Department of Computer Science, University of North Carolina.

HEY H. and PURGATHOFER W. 2001. Occlusion Culling Methods, *Proceedings of Eurographics 2001*, p. 43.

HOPPE H., DEROSE T., DUCHARD T., MCDONALD J. and STUETZLE W. 1993. Mesh Optimization. *Proceedings of ACM SIGGRAPH 1993*, 19-26.

LAW F. and TAN T. 1999. Pre-processing Occlusion for Real-Time Selective Refinement, *Proceedings Symposium Interactive 3D Graphics*, 47-53.

PLATE J., GRUNDHOEFER A., SCHMIDT B. and FROEHLICH B. 2004. Occlusion Culling for Sub-Surface Models in Geo-Scientific Applications, *Symposium on Visualization 2004*, 267-272.

EL-SANA J., SOKOLOVSKY N. and SILVA C. 2001. *Integrating Occlusion Culling with View-Dependent Rendering*, IEEE Visualization 2001, 371-378.

SAONA-VÁZQUEZ C., NAVAZO I. and BRUNET P. 1999. The Visibility Octree. A Data Structure for 3D Navigation, *Computer & Graphics*, 23(5), 635-664.

WANG Y. G., BAO H. J. and Peng Q. S. 1998. Accelerated walkthroughs of virtual environments based on visibility preprocessing and simplification, *Computer Graphics Forum*, 17, 3, 187-194.

ZHANG H., MANOCHA D., HUDSON T. and HOFF III K. E. 1997, Visibility Culling using Hierarchical Occlusion Maps, *Proceedings of ACM SIGGRAPH 1997*, 77-88.