Occlusion Culling for Sub-Surface Models in Geo-Scientific Applications

John Plate[1] Anselm Grundhoefer[2] Benjamin Schmidt[2] Bernd Froehlich[2]
[1]Fraunhofer IMK St. Augustin [2]Bauhaus-Universitaet Weimar

Abstract
Modern graphics cards support occlusion culling in hardware. We present a three pass algorithm, which makes efficient use of this feature. Our geo-scientific sub-surface data sets consist typically of a set of high resolution height fields, polygonal objects, and volume slices and lenses. For each height field, we compute a low and high resolution version in a pre-process and divide both into sets of corresponding tiles. For each tile and for the polygonal objects, the first rendering pass computes a z-buffer image using the low resolution tiles, the polygonal objects and the non-transparent volume objects. During the second pass, we render the same objects against the z-buffer of the first pass while submitting an occlusion query with each object. The third pass reads this occlusion information back from the graphics hardware and renders only those high resolution objects, for which the corresponding low resolution objects were not completely occluded. To avoid fill rate bottle necks, the first two passes may be rendered to a low resolution window. Our implementation shows frame rate improvements for all test cases while introducing only a small overhead and no or hardly noticeable errors. Our non-conservative approach does not require front to back sorting and it works for dynamic scenes.

CR Categories and Subject Descriptors: I.3.3 [Picture/Image Generation]: Viewing Algorithms, Occlusion Culling; I.3.5 [Computational Geometry and Object Modeling]: Object Hierarchies; I.3.7 [Three-Dimensional Graphics and Realism]: Hidden Line/Surface Removal
Additional Keywords: Visibility and occlusion culling, large-scale data visualization, geo-scientific data

1. Introduction

Occlusion culling is an important technique for handling large data sets with medium to high depth complexity. Recently, most graphics cards support occlusion queries, which return the number of visible pixels for a rendered object. This feature can be used to implement efficient occlusion culling techniques.

In this paper, we present an occlusion culling approach adapted for sub-surface models in geo-scientific applications with a focus on data sets from the oil and gas industry. Our sub-surface models consist of a set of height fields, the so called horizons, a set of polygonal objects, the faults, and volume slices and volume lenses. Figure 1 shows a typical data set from the Gullfaks oil field in the North Sea. Horizons are generally high resolution height fields, e.g. 500x500 points, and they are stacked on top of each other, which results in high depth complexity. Faults intersect these horizons partially. They are polygonal surfaces containing a few thousand to ten thousands of triangles. These polygonal objects live inside the volumetric seismic volume, the central data type for oil and gas exploration. Figure 1 shows a single seismic volume, which is the frame of reference for the horizon and fault surfaces. The seismic volume is visualized through slices and volume rendering techniques. Often only local details are visualized through volume lenses such as salt domes or former river beds. Geologists often insist on seeing all the detail of their horizon and fault surfaces and they do not trust mesh decimation techniques.

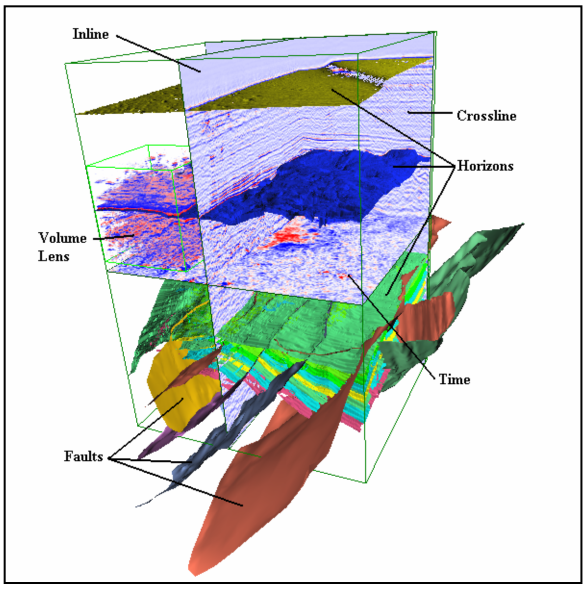

Figure 1: A typical oil exploration data set containing subsurface structures, wells, and seismic slices. The subsurface model consists of two main structures: horizons and faults. Horizons separate two earth layers, and faults are breaks in the rocks, where one side is moved relative to the other. Horizons are typically horizontal while faults are inclined. Three orthogonal slicing planes are used to visualise the seismic volume. The inline-slice is typically perpendicular to the main fault direction. The time-slice is horizontal and the crossline-slice is perpendicular to both.

Our approach generates low and high resolution versions of the original horizon surfaces in a pre-process. Both versions are divided into sets of corresponding tiles (Figure 2). For each fault there is also a low and high resolution version created. The high resolution versions may have the original resolution or might be slightly simplified. The low resolution objects may be decimated by a factor of 5 to 100 depending on the actual data. We use a three pass rendering process to implement occlusion culling. The first rendering pass computes a z-buffer image using the low resolution tiles and polygonal objects and the non-transparent volume objects. During the second pass, we render the same objects against the z-buffer of the first pass while submitting an occlusion query with each object. The third pass reads this occlusion information back from the graphics hardware and renders only those high resolution objects, for which the corresponding low resolution objects were not completely occluded. To avoid fill rate bottle necks, the first two passes may be rendered to a low resolution window.

We present an approach for occlusion culling in geo-scientific environments, which is very easy to implement. Our approach makes efficient use of the occlusion culling support provided by modern graphics hardware. We do not require a particular rendering order and handle moving objects without special treatment. We have tested our approach on large geo-scientific data sets and show that it improves frame rates in all cases while introducing a very small overhead. There is also potential for extending this approach beyond the rendering of sub-surface structures. The main requirement is that the occlusion relationships of low resolution models are equivalent to those of the high resolution models. The main limitation of our approach is the size of the low resolution data set. If this data set exceeds a certain size, it will become impractical to pre-render the low resolution models twice. Modern graphics cards are pushing this limit higher and higher. The pre-rendering stage does not require texturing or lighting and we found that current graphics cards are capable to render more than 100 million such triangles per second on a standard PC.

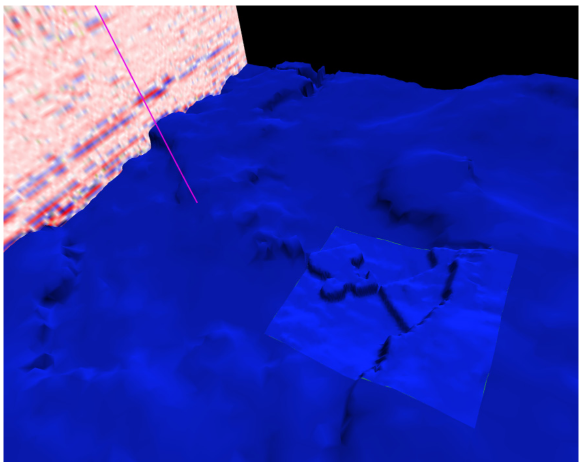
Figure 2: Horizons are divided into tiles. The light blue tile in the middle of the image is the high resolution version. The other tiles around it are low resolution versions.

2. Related Work

The standard hierarchical occlusion culling algorithm [3] based on a hardware supported occlusion culling flag [5] is often combined with a basic view frustum culling algorithm [2]. The basic idea of these approaches is to hierarchically render the scene objects inside the viewing frustum in a front to back order. For each object, the bounding box is rendered first in an occlusion culling mode, which does not affect the frame buffer or z-buffer. The graphics hardware returns the number of visible pixels or simply a flag that states if any pixel of the bounding box is visible or not. If the bounding box is visible, the contained object is rendered or the subtree is treated in a similar fashion. The rendering has to happen in this interleaved way, first the bounding box and then the contained object. The problem with this interleaved approach is that the rendering pipeline is stalled every time the occlusion flag is read back from the graphics card, since the flag can only be computed after the bounding box is rasterized and the contained object can only be submitted to the graphics card if the flag is known. A consequence is that the graphics card achieves only a fraction of its maximal performance. In addition, rendering bounding boxes may affect the fill rate requirements significantly.

Conservative occlusion culling techniques do not introduce image errors. Approximate occlusion culling techniques [7][8][9] accept small image artifacts for speeding up the rendering process. Our approach belongs to this category since we use a low resolution model to determine the visibility of high resolution objects.

Andujar et al. [1] combine level-of-detail rendering with occlusion culling. Their “hardly visible sets” (HVS) are subsets of the potentially visible cells that contributes only a small number of pixels to the overall picture. Their rendering framework uses a user defined error bound to choose from a fixed set of level-of detail representations based on the HVS error estimates. El-Sana et al. [6] have presented a novel approach for incorporating occlusion culling within the framework of view-dependent rendering. Their idea is based on estimating an occlusion probability instead of computing exact visibility. This occlusion probability and the view-parameters determine the appropriate level of detail for each frame. Our approach uses a simple level-of-detail approach in combination with hardware supported occlusion flags, but has only been tested for geo-scientific data sets.

Law and Tan [4] incorporate level of detail techniques with occlusion culling. In particular they suggest the use of simplified geometry as virtual occluders. We also use simplified geometry for our occlusion computations, but our approach is based on the idea that the occlusion relationships in between objects of the simplified model are similar to those of objects of the original model.

3. Algorithm

Our approach is divided into a pre-processing phase and the actual rendering phase, where data exploration and manipulation takes place.

3.1. Data preparation

Horizon and fault surfaces are manually or partially automatically created polygonal data sets, which are based on seismic data, well log data (data measured down a drill hole), and other data sources.

Horizon surfaces are height fields, which are represented as sets of triangles. From the original data, we generate a low resolution and a high resolution version. The high resolution version represents often the original data, sometimes the data is slightly reduced by a factor of two to three to achieve interactive frame rates. We use an adaptive simplification technique, such as Michael Garland's qslim software [11], for the mesh reduction process. The low and high resolution version of each horizon are then divided into a set of equally sized tiles, e.g. 5x5 or 7x7 tiles.

Faults are polygonal surfaces typically represented as triangle sets. Faults consist of a few thousand to a few ten thousand triangles. We use in general the original version of the faults as the high resolution representation. The low resolution is typically decimated by a factor 5 to 10. With our current implementation, we do not split faults into tiles, since they are typically much smaller than horizons, which makes it less worthwhile. Faults might occlude large parts of the horizon surfaces depending on the actual view point.

Seismic volumes represent the subsurface structure. They are acquired by sending acoustic shock waves into the ground where they are reflected and refracted. The amplitudes and travel times of acoustic waves returning to the surface are measured and processed into regular three-dimensional scalar grids. These volumes vary widely in size. Small volumes are in the range of tens of megabytes, large volumes may reach into the hundreds of gigabytes. We use Octreemizer[10], our hierarchical volume roaming toolkit, for rendering slices and volume lenses of such large data sets. Octreemizer allows roaming through very large volumes using a hierarchical two-level paging approach, which pages volume bricks from the hard disk into main memory and from main memory into texture memory. Slices through the seismic volume often occlude a substantial part of the horizons or faults, but they are rarely occluded. Volume lenses extend typically only across a small part of the total volume and show local features. They consist of a stack of slices, which is rendered in back to front order. They are typically rendered in a semi-transparent mode and do not occlude other objects, but they might be occluded – at least partially. Our current implementation does not test for occlusion of volume rendering lenses, but uses volume slices as potential occluders.

3.2 Three pass algorithm

We implemented a three pass algorithm for rendering our data sets with occlusion culling. During the pre-processing step, we generated low and high resolution versions for each tile of each horizon and for each fault. The low resolution versions are used for the first two rendering passes – the pre-rendering stage - and the high resolution versions are used for the third rendering pass. In pseudo code, the three rendering loops look like this:

```
/* first pass */
turn lighting, shading, and texturing off;
clear z-buffer;
disable framebuffer_writes; // no need to write
for each horizon h          // to the framebuffer
  for each tile t of h
    render low res version of tile t;

for each fault f
  render low res version of fault f;

for each volume slice s
  render slice s;

/* second pass */
disable z-buffer writes;
for each horizon h
  for each tile t of h
    begin occlusion query;
      render low res version of tile t;
    end occlusion query;

for each fault f
  begin occlusion query;
    render low res version of fault f;
  end occlusion query;

/* third pass */
turn lighting, shading, and texturing on;
enable z-buffer and frame buffer writes;
clear frame buffer and z-buffer;
for each horizon h
  for each tile t of h
    read back occlusion result for low res t;
    if (low res t was visible)
      render high res version of tile t;

for each fault f
  read back occlusion result for low res f;
  if (low res fault f was visible)
    render high res version of fault f;

for each volume slice s
  render slice s;
```

During the first rendering pass, all the objects potentially occluding other objects are rendered. These are the non-transparent horizons, faults, and volume slices. Transparent objects would be excluded from this pass. The first pass does not need to render to the frame buffer, we are only interested in a z-buffer image. We also turn lighting, texturing, and shading off for the first pass. If the rendering process becomes fill limited, we could render the first and second pass to a lower resolution view port than the final render pass.

For the second pass, we need to explain how the occlusion query extension for NVIDIA and ATI graphics cards work. There are basically several different occlusion operations. For each occlusion query, the user needs to generate an occlusion ID, which is associated with the occlusion query. In OpenGL, an occlusion query is started with a glBeginOcclusionQueryNV(ID), then the objects are rendered, and the occlusion query is closed with glEndOcclusionQuery(). The rendering in between the glBeginOcclusionQueryNV and glEndOcclusionQuery does not affect the frame buffer or the z-buffer. The occlusion ID is then used to check if the occlusion query results are ready and to fetch the occlusion results. The occlusion results are therefore computed completely asynchronously to the host's rendering process. The occlusion results could either be the number of visible pixels of an object or just a flag. The occlusion do not need to be fetched right after the occlusion query as it was the case with the original HP occlusion query implementation[5]. In fact, fetching the occlusion results right after submitting an occlusion query stalls the graphics pipeline, since the submitted object needs to be rasterized before the occlusion results can be computed.

Our second pass submits an occlusion query for each low resolution object we are interested in. The occlusion tests are performed against the z-buffer from the first pass, which results in completely correct occlusion information for the low resolution objects. In our example, we do not submit occlusion queries for the volume slices, since they contain typically very few polygons. Transparent objects need to be considered in this pass, since they can not occlude, but may be occluded.

For the third pass, we switch texturing, lighting, and shading back on. During this pass we fetch the occlusion query results for each low resolution object just before the corresponding high resolution object is about to be rendered. As the result of the occlusion query we get the number of visible pixels of the low resolution object. The high resolution object is rendered only if there are more pixels visible than a certain threshold. During the third pass, we also need to render all the objects that are not considered during the second pass. Transparent objects can be rendered last in a back to front order using occlusion query results from pass two.

4. Results and discussion

We implemented our occlusion culling approach in C/C++ under Windows XP using the OpenGL graphics library. Our implementation works for current NVIDIA and ATI cards that support the GL_NV_occlusion_query extension.

Our implementation was evaluated on a Dell Precision 650 dual processor PC with 2GB of main memory running Windows XP. We compared two graphics cards, an NVIDIA Quadro FX2000 and an ATI FireGL X1. Both cards as well as the main board support AGP 8x. The NVIDIA card has 128MB and the ATI card 256MB of unified video and texture memory. For our tests, we used the NVIDIA 43.45 drivers, for the ATI card we used the 10.30 drivers.

Our test data set (see figure 1) consists of eleven horizons and sixteen faults with a total of 3.88 million triangles. The volume data set has a resolution of 525x475x751 voxels. For the low resolution versions of the faults and horizons we used a 1:100 decimated version. For the simplification process, we used Michael Garland's qslim software version 2.0. For each polygonal object, we generated triangle strips and used display lists for faster rendering. The data set was illuminated with two positional light sources.

Figure 3 shows some results for two representative view points. We compare rendering the scenario with a different number of tiles per horizon on the aforementioned graphics cards. The draw time corresponds to the rendering time for the third pass and the cull time is the sum of the rendering times for pass one and two. Rendering time reduction is basically proportional to the number of occluded triangles in relation to the total number of triangles. The overhead for the first two rendering passes is in most of our cases less than twenty percent, but increases if smaller portions of the scene are visible. This is mostly due to the highly decimated horizon surfaces, but also due to the fact, that unlit and untextured triangle strips can be rendered at a rate of more than 100 million triangles per second on both graphics cards. Rendering the high resolution objects lit with two positional light sources can be rendered at around 30 million triangles per second, with one directional light source it goes up to 55 million triangles per second. Our occlusion culling approach does not impact the overall rendering performance, since we avoid stalling the graphics pipeline for the occlusion queries.

The NVIDIA and ATI cards show very similar behavior with respect to the dependency on the number of tiles used. Both graphics cards are performing roughly equal. The scale on the y-axis for the bar charts in figure 3 is quadratic. For most cases the optimal speedup is reached at around 10x10 tiles per horizon surface. If the number is lower, the occlusion culling granularity is too coarse. If the number is higher, the triangle strips for the low resolution versions get shorter and shorter, which decreases the rendering efficiency during the pre-rendering stage.

The approach does not require a static environment. Our application scenario requires that objects are sometimes moved around or removed. Fault positions need to be slightly adjusted or a horizon is moved out of the way to see occluded areas.

Rendering times for the shown data set are decreased by 30 to 80 percent depending on the actual view point and the position of the volume slices. We cannot guarantee a certain frame rate, but our approach is always faster than rendering without occlusion culling. We also tested different resolutions – up to 2048x1536 – without influence on the results. This clearly shows that we are geometry limited rather than fill limited. Even the size of the pre-rendering viewport did not have any influence on the frame rates.

Our technique is non-conservative and might incorrectly classify objects as visible or invisible. There are two main sources for incorrect classification: the pixel threshold and the use of the simplified model for determining occlusion.

The occlusion queries return the number of visible pixels for each object. A pixel threshold is used to classify barely visible objects as invisible. This threshold depends directly on the size of the pre-rendering view port, which might be smaller than the viewport for the final rendering pass. We have experimented with various thresholds in a 1280x1024 resolution window and found that a 10 pixel threshold results in hardly visible errors for our test scenario. Figure 4 shows the same view onto the high resolution data for pixel thresholds of 0 and 50 pixels. Even for 50 pixels it is hard to spot the difference. Higher pixel thresholds result typically in significant speed ups, since less objects are rendered. For the 50 pixel threshold we achieve a rendering time reduction of nearly 50% compared to the 0 pixel threshold for this particular case. We use a 10 pixel threshold as a default, which does not result in popping artifacts or obviously missing tiles for our example.

The use of simplified models for determining the visibility of an object may introduce severe errors, since in general the simplification does not preserve the occlusion relationships of the original model. However, we found that for our case the approach was rather robust. Figure 5 shows an example for simplification by a factor of 25 and 400. The original model consists of 3.8 million polygons. For factors of 25 and 100 (not shown), it is hard to find any errors. The number of rendered triangles in the third pass differs only by 1 percent, which explains the visual similarity. For a simplification factor of 400, some objects are clearly missing. The effect of the simplification factor depends strongly on the type of data. In addition, different simplification algorithms may preserve occlusion relationships between the simplified and the original model to a different degree.

5. Conclusions and future work

We presented an non-conservative occlusion culling approach for geo-scientific sub-surface data sets, which is easy to implement. We make efficient use of occlusion culling support provided by modern graphics hardware. Our technique does not require a particular rendering order and works for dynamic scenes as well. We have tested our approach on large geo-scientific data sets and show that it improves frame rates in all cases while introducing only a very small overhead in most cases. The main disadvantage of this approach is that it needs to pre-render the whole scene in low resolution during the first two passes. This approach will not scale beyond a certain depth complexity and it will not handle very large scenes beyond tens of millions of triangles.

Currently, we use only two levels of detail – a coarse level and a fine level. Using more levels of detail or even adaptive level of detail techniques for the final pass as well as for the pre-rendering stage could improve the scalability of the whole approach significantly. Pre-rendering correctness could be traded for pre-rendering time consumption. From another point of view it is a challenging research topic to develop occlusion relationship preserving mesh simplification algorithms and measures that predict potential occlusion errors.

Our current implementation is not yet perfectly integrated with Octreemizer, our volume roaming tool kit. The volume bricks for occluded parts of volume lenses or slices are currently still downloaded into the graphics card and rendered. We plan to integrate our occlusion culling approach tightly with Octreemizer. The octree structure of the volume representation integrates perfectly with our occlusion culling approach.

We have only tested our approach in the context of sub surface data. We plan to look at other application fields trying to extend the applicability of this work. Potential application domains include game engines and CAD data visualization.

Acknowledgements

This work was supported by the VRGeo consortium – www.vrgeo.com.

References

[1] C. Andujar, C. Saona-Vazquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of detail through hardly-visible sets. Computer Graphics Forum, 19(3): pp. 499–506, August 2000.

[2] D. Bartz, M. Meißner, and T. Huettner. OpenGL-assisted Occlusion Culling of Large Polygonal Models. Computers & Graphics, 23(5): pp. 667–679, 1999.

[3] D. Bartz and M. Skalej. VIVENDI - A Virtual Ventricle Endoscopy System for Virtual Medicine. In Proceedings of Symposium on Visualization, pp. 155–166, 1999.

[4] F. Law and T. Tan, Preprocessing Occlusion for Real-Time Selective Refinement, In Proceedings of the Symposium on Interactive 3D Graphics, pp. 47-54, ACM SIGGRAPH 1999.

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

[6] J. El-Sana, N. Sokolovsky, and C. Silva. Integrating Occlusion Culling with View-Dependent Rendering. IEEE Visualization 2001, pp. 371-378, 2001.

[7] J. T. Klosowski and C. T. Silva. Rendering on a budget: A framework for time-critical rendering. In IEEE Visualization 99, pp. 115–122, 1999.

[8] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. IEEE Transactions on Visualization and Computer Graphics, 6(2): pp. 108–123, 2000.

[9] H. Zhang, D. Manocha, T. Hudson, and K. Hoff III. Visibility culling using hierarchical occlusion maps. In Proceedings of SIGGRAPH 1997, pp. 77–88, August 1997.

[10] J. Plate, M. Tirtasana, R. Carmona and B. Fröhlich. Octreemizer: a hierarchical approach for interactive roaming through very large volumes. Proceedings of the symposium on Data Visualisation 2002, pp. 53ff, 2002

[11] M. Garland and P. Heckbert. Surface Simplification Using Quadric Error Metrics. PIVC. of ACM SIGGRAPH, pp. 209-214, 1997.

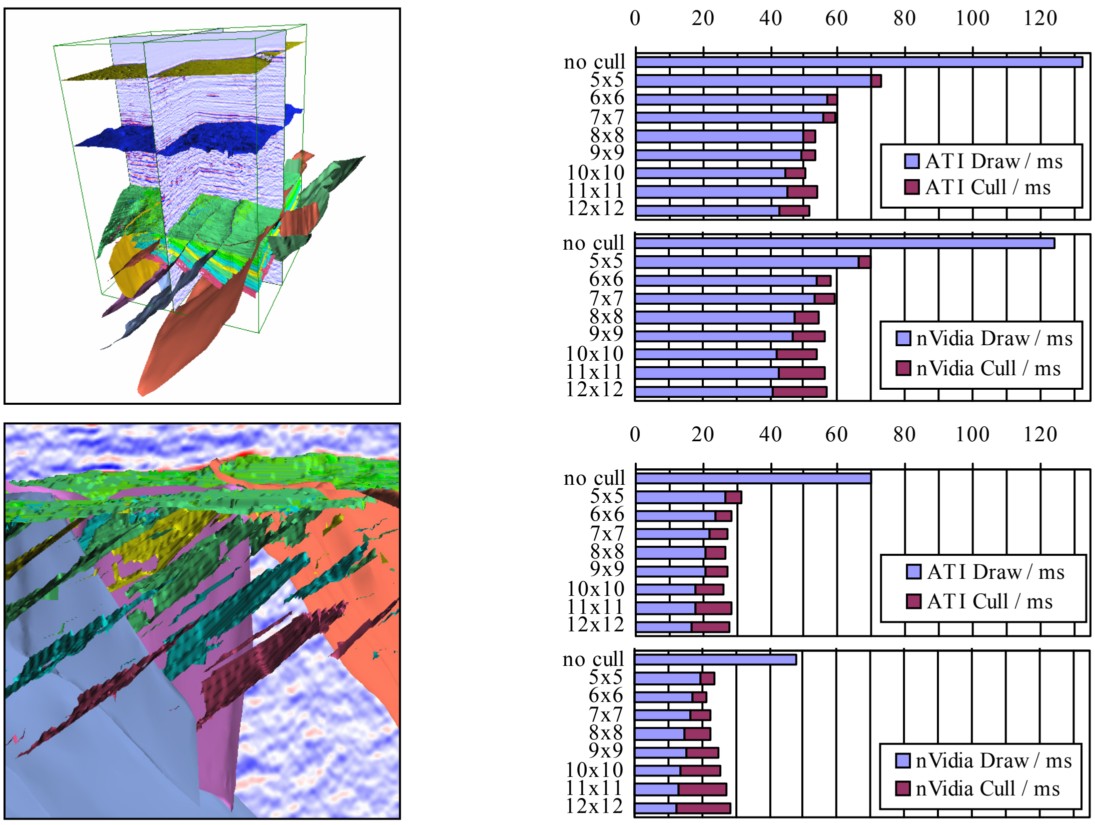

Figure 3: Two representative views onto the data. Both graphics cards behave pretty similar overall.

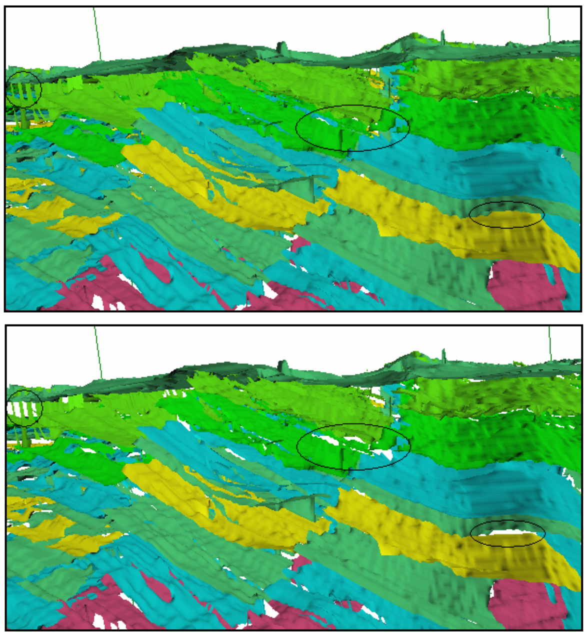

Figure 4: The effect of the pixel threshold (0 pixels on top and 50 pixels on bottom) on the final image. For a threshold of 0 pixels 940k triangles are rendered, and for 50 pixels 476k. Some of the errors are marked. In these areas there are typically complete horizon tiles missing in the lower image. The colors are just arbitrarily chosen such that the horizon surface can be visually differentiated.

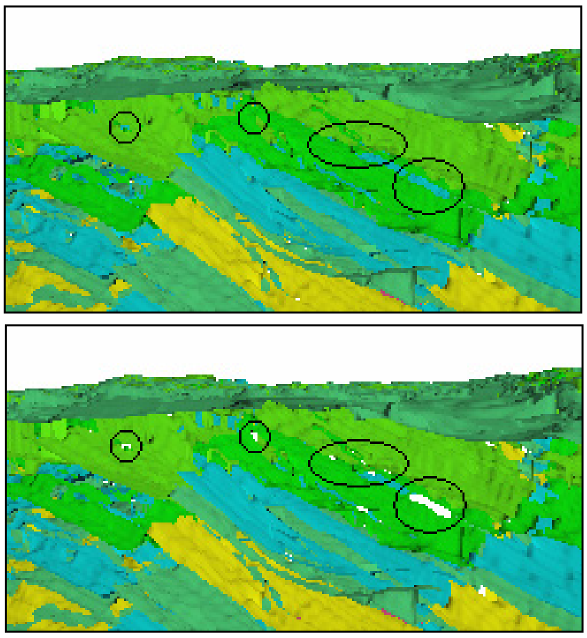

Figure 5: The effect of different simplification factors (25 on top and 400 on bottom) on the final image. For a simplification factor of 25 there are 1388k triangles rendered for the final image, and for a factor of 400 there are 1280k triangles rendered. The pictures are enlarged versions of the critical areas and some errors are marked, where horizon tiles are missing.