Adaptive Progressive Vertex Tracing for Interactive Reflections

Thomas Ullmann

Commercial Vehicle Development, DaimlerChrysler AG, Tilsiter Str.15, 71059 Sindelfingen, Germany
Thomas.Ullmann@DaimlerChrysler.com

Alexander Schmidt, Daniel Beier, Beat Bruderlin

Technical University of Ilmenau, Dept. of Computer Science and Automation
Beat.Bruederling@prakinf.tu-ilmenau.de

Abstract

This paper presents an approach for real-time rendering of physically accurate reflection effects in virtual environments. We apply a hybrid rendering of an OpenGL-generated scene, blended with correct reflection characteristics of selected scene objects.

The core of the approach consists of a special type of ray tracing, the so-called vertex tracing. Real-time performance, even for complex CAD scenes, is achieved by progressive adaptive refinement, (derived from the geometry in object space) as well as by parallelization of the algorithm. A mesh-based load balancing yields a uniform distribution of the computing load in a heterogeneous network with resources with widely varying performance.

The performance of the overall system is demonstrated using a truck interior in a Virtual Reality simulator.

1. Introduction

Computer graphics offers a broad spectrum of methods for the simulation of light distribution, which reach from the strongly approximate to the physically accurate. Ray tracing has been established as the de facto standard approach for accurate rendering of specular reflections. Although it is extremely computationally intensive, ray tracing is widely applied for the calculation of high quality lighting scenarios. In combination with other lighting approaches, such as radiosity, ray tracing is also suitable for global illumination computation. At the other end of the spectrum, hardware-supported environment mapping simulates reflecting surfaces at real-time frame rates; however, it is not geometrically accurate, and only renders a plausible first impression of a reflecting surface.

Our application for real-time, accurate rendering is an interactive computation to simulate reflection effects on instrument covers in a truck interior (figure 1). Here a glare-free readability of the instruments represents a substantial criterion for the vehicle ergonomics. Glare not only depends on the reflection coefficients of the surface, but also considerably on the shape and orientation of the instrument cover. From a practical standpoint, the reflection coefficients cannot be reduced arbitrarily, therefore technical designers must orient themselves toward adapting the shape of the reflector.

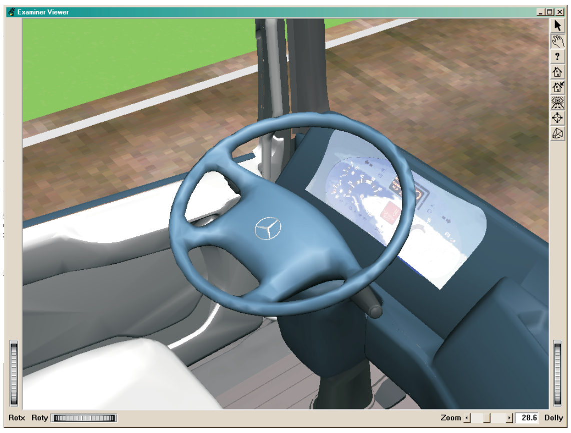

Figure 1: *Simulation of reflections on instrument cover*

Typically, ellipsoid shaped reflectors are employed for this purpose. It is known that all light rays passing through focus F_1 of an ellipsoid are reflected toward the second focus F_2 (figure 2). If a weakly diffuse reflecting object (black component in the interior) is located at F_1 and F_2 is the eye point of the driver, only the small amount of light energy that is reflected by the dark surface at F_1 reaches the eyes of the driver.

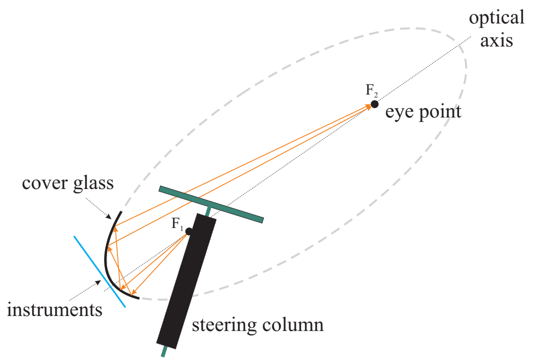

Figure 2: *Geometry of rays in the ellipsoid example*

However, the geometric solution of using ellipsoid reflector surfaces (or other geometric shapes with comparable optical characteristics such as paraboloid or hyperboloid) can represent only an idealized form of the instrument cover, since the eye point of the driver may vary in position. Thus an optimal reflector shape needs to be verified experimentally to guarantee minimal glare, as well as compatibility with the cockpit design. This application background justifies the requirement for geometric accuracy, as well as for real-time performance of the software developed in this research.

The Vertex Tracing approach introduced in this paper follows an adaptive ray tracing approach (which drastically reduces the number of rays) in combination with parallel computation. Furthermore, ray tracing is only applied to the actually reflecting surfaces, in combination with hardware-accelerated OpenGL rendering for the rest of the VR scene. The primary ray is generated directly from the tesselated reflector geometry with subsequent adaptive refinement in object space, rather than in raster (image) space. The distributed computation in a heterogeneous network uses dynamic load balancing with a demand-driven strategy. For additional acceleration, the necessary ray object intersection test on CAD data employs optimized collision detection. Complex CAD models are organized in a bounding box hierarchy[6].

In the following, after a short overview of related work, the function of the adaptive vertex tracer is described. After that we discuss the parallel mode of operation of the vertex tracer with dynamic load balancing. The results are demonstrated with a complex VR scene example.

2. Related Work

Today's ray tracing systems achieve their relatively high performance only by combining several acceleration methods. In addition to the many methods for acceleration of the intersection test, there also exists a broad set of approaches within the area of progressive or parallel ray tracing. Already, many techniques have been investigated to accelerate ray tracing[21]. For instance, fast ray intersection tests, generalized rays, a reduction of the number of computed rays or parallelization (distributed ray tracing) are well-known. However, so far no approach has achieved real-time performance for large scenes (100,000 polygons or larger).

Among the acceleration methods by means of progressive ray tracing for adaptive pixel sampling, one essentially distinguishes uniform and non-uniform sampling. JANSEN and VAN WIJK[10] early on describe a uniform sampling approach. Here the image plane is divided into regular squares and afterwards gradually refined in accordance with the intensity variance of the sub-squares. Today this method finds widespread application, due to its trivial implementation.

Adaptive, non-uniform pixel sampling describes another way of progressive refinement. In this method, the image plane is considered a continuous region with no pixel boundaries. The goal is to adapt the content of the picture with as few samples as possible. This means, the progressive refinement depends on the available edges and silhouettes, as regions with high contrast or frequency differences. In this context PAINTER and SLOAN[17] proposed a method to refine the image based on a 2D binary subdivision. Depending on the variance and the number of already computed samples, a region is refined until a particular confidence level of the image is reached.

Other sample generators[7, 12] follow the strategy of the *Delaunay triangulation*[18] in the image plane. PIGHIN et al.[16] extended these by a pre-calculation and the use of a *Discontinuity Mesh*. Here the scene is first generated by hardware rendering, in order to detect edges and silhouettes. The edges represent the basis of a mesh, whose vertices present the primary ray-traced samples. Finally, the refinement of the pre-calculated mesh is carried out by the Delaunay triangulation.

BALA[3] developed an approach describing a progressive partitioning in the object space with the help of so-called *interpolants*. In this case, the interpolants are constructed around the scene objects. They approximate the radiation within the corresponding area of the object and can be refined in accordance with the requirements. The generation of the image is carried out as line-by-line scanning within the specified representation error. A primary ray meets a valid interpolant and the corresponding pixel is determined by its radiation values.

The parallelization of adaptive progressive ray tracing becomes far more difficult compared to sequential implementations, due to dependencies of samples accross refinement levels. Discontinuity edges can occur because of the distribution of the image space over different processors.

Notkin et al.[12] and Reisman et al.[19] reduce this problem by shooting additional samples at the region boundaries. Also, Reisman et al. distribute only connected convex regions, in order to keep the boundaries between the slave areas as small as possible. Both methods are based on a distributed memory system with static load balancing, requiring analysis of the scene variance in a pre-processing step. However, for better balancing, additional dynamic distribution is performed, and the slaves can exchange their tasks among themselves.

The approach presented in our paper is based on adaptive progressive ray tracing with non-uniform sampling. The parallelization of our approach is carried out by dynamic central load balancing, and also works for heterogeneous networks.

3. Vertex Tracing

As described above, the approach introduced in this paper is based on a hybrid representation of the VR scene using OpenGL-generated graphics, in combination with ray tracing. The geometry of the scene is already in tesselated form. Therefore it is advantageous to generate so-called secondary rays directly from reflecting surfaces in object space, and to adaptively refine the polygon mesh. Here the arrangement of the vertices in object space determines the coincidental variance of the primary samples, which are assumed to be distributed reasonably over the image plane.

Owing to the fact that the vertices in object space represent the starting point for each ray, we named our approach *Vertex Tracing*.

In traditional ray tracing, for each pixel of the image plane a so-called primary ray is sent into the scene and ist tested for possible collision with all scene objects[8]. The primary ray computation takes $n \times m \times k$ calculation steps, where n and m define the size of the pixel raster, and k is the number of objects and thus the number of ray-object intersection tests per pixel. For an average rendering task, potentially several billion primary ray object intersections need to be carried out. These primary ray intersections are completely avoided by our approach.

Rather, in the case of our vertex tracer, the primary rays are generated directly between the eye point and the corresponding vertices (figure 3). In order to ray-trace only the visible object areas of the reflector, we distinguish between hidden and visible vertices. Before starting the actual ray tracing process all vertices are checked for visibility. In the case of a complete covering of the reflector, no ray tracing cost will result.

The visibility test is carried out by an ID-buffer, which is essentially a Z-buffer. All objects of the scene are indexed by a unique color-ID and rendered into the ID-buffer. Based on the resulting color-ID the visibility of each vertex is determined (figure 3).

Vertices of an edge are transferred to the base ray-tracer (a classical recursive ray-tracer for single rays), if and only if at least one of the vertices is visible. Thus it is ensured that only visible polygons are rendered[†]. After that, valid edges are stored in *EdgeList*, sorted by their length in the projection plane (figure 6), and are available for the subsequent refinement process.

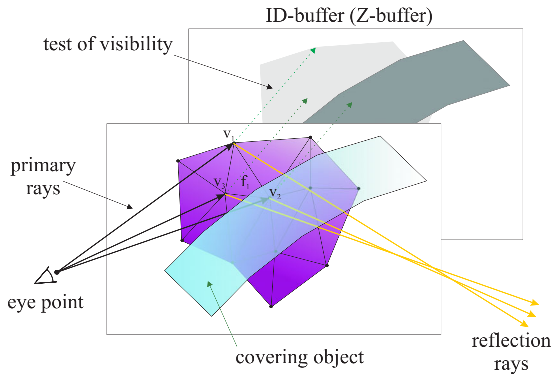

Figure 3: *Vertex-based primary ray determination*

The reflected and the refracted rays are calculated for each visible vertex, its incident primary ray and its normal. Also, so-called shadow sensor rays emanate from the position of the vertex toward the respective light sources.

For determination of a possible object collision, these secondary rays are passed to the base ray tracer. The base ray tracer first tests each ray for intersections with the voxel space[2] and only tests for collision *those* nodes of the object hierarchy tree[6] that passed the voxel test.

The intensity value (color determined by the Phong illumination model) and the determined ray tree[9] from the recursive ray-tracer are the basis for the progressive refinement of the mesh. In order to reliably detect potential discontinuity edges as well as non-linearities, the two vertices of a polygon edge are compared by such criteria. If the condition is fulfilled, a subdivision of the edge or the adjacent polygon edge is performed according to the refinement strategy (see paragraph 3.1).

Using the color intensities of the vertices determined by the base ray tracer, already a preview image can be produced. The intensity values are interpolated linearly by means of Gouraud shading, carried out off-line. This results in a texture image in separate memory which is copied into the frame buffer and combined with the rest of the scene, using stencil buffer, z-buffer and alpha blending operations. Consequently, we are able to visualize each step of the refinement process at user-defined breaks of computation.

The refinement of a polygon is terminated if the length of the edges projected onto the image plane falls below a minimal

[†] The case that all vertices are covered by concave or several objects, such that sections of the polygons remain visible, is not considered at the moment.

threshold l. In this case all pixels covered by the polygons of the last refinement step would be calculated by the base ray-tracer and are drawn directly on the image plane. This step is called *triangle filling*. It is the last step of the refinement and it enables the representation of discontinuity edges with pixel accuracy.

3.1. Progressive adaptive refinement

In contrast to other methods, our subdivision method operates not directly on faces but on the edges in *EdgeList*. Each edge is tested for subdivision, and if necessary, a new vertex is inserted.

The advantage of this method is the identical generation of child faces with a common edge e (figure 4). The *explicit subdivision* of adjacent faces, in order to prevent possible discontinuities between the face boundaries can therefore be avoided. In particular, with regards to a distribution of the ray tracing over several processors, no dependencies between the faces exists. Redundancies in a distributed computation are reduced drastically. In addition, the edge-based method generates only one new vertex v_{new} per iteration cycle and therefore only one new ray needs to be calculated by the base ray-tracer, although *four* new faces (child faces) are created, when two adjacent parent faces are subdivided.

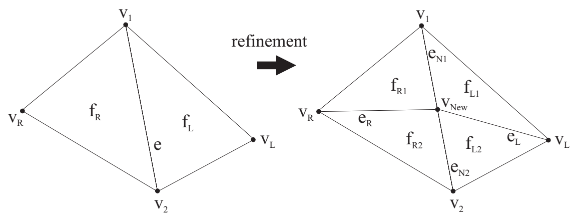

Figure 4: *Edge-based subdivision of neighbouring triangles*

Figure 6 shows the base algorithm for adaptive refinement. The refinement process is terminated if *EdgeList* contains no more edges. Based on the initial edges, starting with the longest edge, each edge is tested for a necessary subdivision with regards to the ε criterion (*CheckRefinement()*, line 4). In case of a necessary refinement (line 5) vertex v_{new} is inserted (see also figure 4). The faces f_{R1}, f_{R2}, f_{L1} and f_{L2} (*newFaces*) and their appropriate edges (*newEdges*) e_R, e_L, e_{N1} and e_{N2} are created. A progressive refinement of a triangle after several steps illustrates figure 5.

Each edge of *newEdges* is also sorted in *EdgeList* by its length (line 7). Analogously to the procedure of the primary edges, the function *RaytraceEdge()* carries out the visibility test of the corresponding vertices and transfers them to the base ray-tracer if necessary.

Line 8 allows a continuous visualization of the progressive mesh refinement by interrupting the ray tracing at predetermined times (*TimeToBreak*). For this purpose, faces which are accumulated in a list *newFaces*, are rendered into the texture buffer. The image produced in the previous step is preserved, such that only within the areas of the newly added faces a further refinement of the reflection or refraction will be visible. Finally, the texture image is copied into the frame buffer of the whole scene.

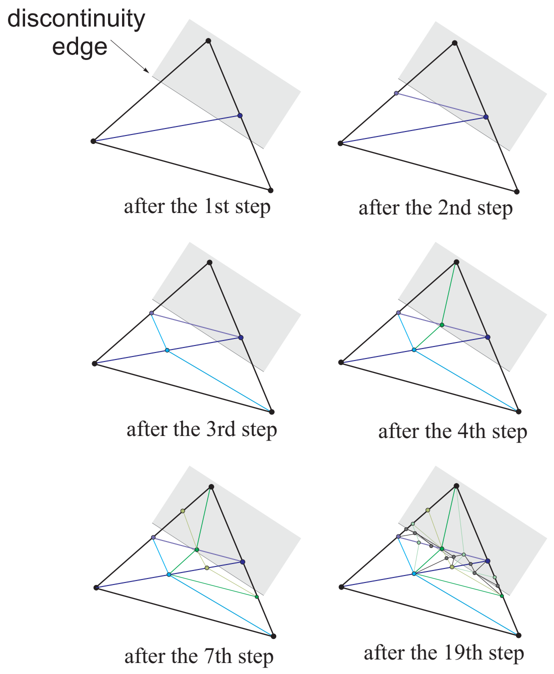

Figure 5: *Step by step refinement*

```
Refine(EdgeList, ID-Buffer)

1   Do until EdgeList is empty
2       nextEdge ← EdgeList
3       Decrease(EdgeCount)
4       If (CheckRefinement(nextEdge, ε))
5           newEdges ← RefineEdge(nextEdge)
6           For every newEdges $e_j$
7               RaytraceEdge($e_j$, ID-Buffer)
8           If (TimeToBreak) ⇒ Render(newFaces)
```

Figure 6: *Function RefineObject() of the vertex tracer*

3.2. Convergence problems

Figure 7 shows the emergence of a typical convergence problem in the edge-based refinement method. Two edges of the triangle are continuously refined, but the third edge (left edge in the figure) maintains its length, since its vertices dont't indicate intensity differences. The edges of the left triangle come closer after several recursion steps, whereby the lengths converge towards a value, which is above a fixed threshold. The procedure does not terminate in this case.

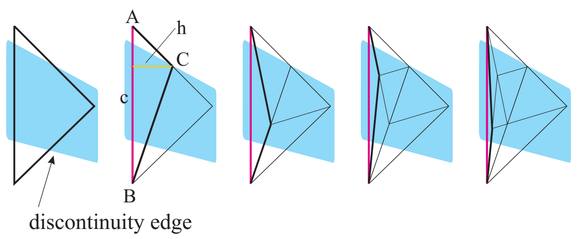

Figure 7: *Problems with convergence*

A possible solution to the convergence problem is a constrained refinement of the longest edge of such triangles. This achieves a refinement of flat triangles and forces a convergence of the triangle edges against a lower limit. The condition for the termination is the "factor of flatness," defined as the relation v of the edge c, i.e. the height h divided by c (see figure 7), as follows:

```
if ($\frac{h}{c} < v$)
    then Refine $c$
    else Ok.
```

Since the refinement process is executed based on criteria in the image plane, a discretization of the values results because of the pixel raster. Rounding errors can occur, and a height h which is smaller than a pixel creates larger triangles. The procedure may be caught in an infinite loop. In order to deal with this precision problem, the refinement is aborted if h falls below the threshold of two pixels. Beyond that, a pre-defined maximum recursion depth is applied.

3.3. Performance results of vertex tracing

For a Virtual Reality application of the vertex tracer, the requirement of interactive frame rates during the reflection computation is crucial. For performance tests, shown in the figures 10-13 a concave curved cover glass was located in front of the instrument in a truck interior. The complete scene consists of approximately 97,000 polygons.

Figure 10 shows the *preview* of the vertex tracer. As can be recognized in this figure, the approximate representation already offers a first impression of the reflection characteristics of the cover glass. The reflection of the left side window clearly appears in the lower half of the instrument cover. The total time needed for the rendering of the scene is below 220 ms on an 850 MHz PC with a GeForceDDR graphics accelerator and fulfils the minimal real time requirements of Virtual Reality. In comparison, a complete ray tracing of the reflector surface alone takes almost 2 minutes on the same workstation. Hence, an approximately 500-fold improvement of performance can be demonstrated here.

The figures 11 and 12 show the progressive refinement process up to the completion of the triangle partitioning, as well as the complete triangle filling (described above). Despite the obviously lower computation cost, compared to triangle filling (see table 1), the representation already is of very high quality.

Rendered Image	time in s	pixel	factor
Reference ray tracing	110.56	184,443	1
Preview	0.22	322	502
Refinement without triangle filling	1.43	1972	77
Refinement with triangle filling	6.12	6790	18

Table 1: *Comparison of the separate refinement steps*

Figure 13 shows the resulting adaptive triangle refinement of the vertex tracer. The approach shows a subdivision only at high-contrast reflection edges (e.g. the side window reflection) on the cover glass.

4. Vertex tracing in distributed environments

It is well known that classic ray tracing is suitable for parallelization, yielding almost linear speed-up. Generally, there exist two main models of parallel computation, the *demand-driven* and *data-driven* methods[12]. In the case of demand-driven methods, the distribution is performed dynamically, based on task packages, whereas in data-driven approaches, the distribution is done by static data analysis.

Due to the necessity of also implementing the presented approach in a heterogeneous network environment with restricted communication bandwidth, we pursued the demand-driven approach. In particular, each slave possesses a complete copy of the scene but is only responsible for an assigned part of the objects to be ray-traced.

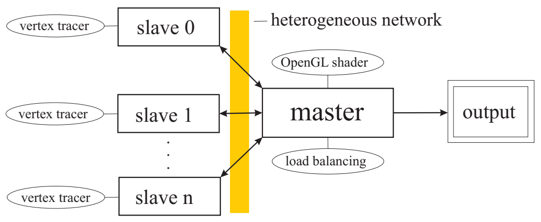

Figure 8: *Concept of distributed vertex tracing*

The master process manages the distribution (figure 8). Based on central, dynamic load balancing, the master determines the "size" of the task packages that can be computed, and redistributes these accordingly, after each frame.

After the computation, the results of the slaves are combined and blended with the OpenGL-generated scene. This concept reduces the communication costs to a minimum, since only at the beginning (when the triangles are distributed) and toward the end of a computed frame (when the resulting bitmaps are returned), communication between the master and the slaves occurs.

4.1. Discussion of parallelization

As already has been observed with classical parallel ray tracing, the speed-up of parallel vertex tracing also depends on the communication costs in connection with the resulting redundancy. In particular, in a heterogeneous network with unexpected network load, a sporadic communication failure of only *one* slave can exert a fatal effect on the performance of the overall system.

Tests showed that despite a consideration of the communication performance in the load balancing, strong fluctuations cannot be balanced within a small number of frames. However, a relatively steady but unequal communication, for example caused by a mixture of 10 and 100 MBit connections, can be successfully balanced.

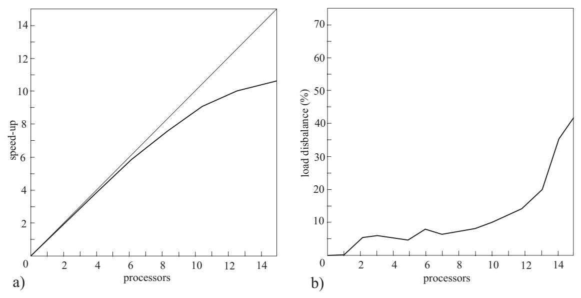

Figure 9: *Quantitative evaluation of the truck scene*

Figure 9 shows the quantitative evaluation of the speed-up (a) as well as the load-disbalance (b) in the example of the truck scene (figure 1) but with a higher geometry complexity (about 210’000 polygons). Almost linear speed-up up to approximately ten processors can be achieved. Accordingly, the load disbalance remains below 10%. The resulting increase of the disbalance negatively influences the speed-up and limits the system for a larger number of slaves.

The frame rate for a complete refinement achieved by a system with 20 PCs is about $\geq$ 1.3 s/frame. The single processor version for the same test case achieves about 13.2 s/frame. It was noticeable that, particularly with a small computation load, the communication factor has more influence and the system cannot be faster than about one second per frame, in most cases. Consequently, the speed-up tends toward worse behavior with a lower computation load.

In contrast to that on a Onyx^2 with 8 processors and shared memory we obtained in a similar test scenario up to 450 ms (Vertex Tracing without triangle filling) because of the high communication rate.

We conclude that, in comparison, the shared memory version generally shows better speed-up for a larger number of processors, whereas for distributed processing in a heterogeneous network the communication bandwidth is becoming a bottleneck. Despite this fact, we observe that for more complex rendering tasks (when rendering time dominates the data transfer rate) we can still achieve linear speed-up for a larger number of processors, even in a heterogeneous environment.

As far as the rendering quality is concerned, no difference between distributed vertex tracing and the single processor version is observed. As previously mentioned, this fact is due to the edge-based refinement strategy of the vertex tracer.

5. Summary and future work

In this paper, we propose Vertrex Tracing, an approach for interactive reflection computation based on the ray tracing algorithm for applications in complex virtual reality environments. To achieve this goal, the principle of primary ray computation was extended. We directly use the mesh structure of the reflecting objects in the scene to generate the primary rays, without costly object-ray intersections. Using a hybrid approach, hardware-accelerated rendering is used for diffusely reflective objects, as well as for the visibility determination of the vertex tracing.

Subsequently, a progressive adaptive refinement of the mesh structure is executed by edge-based partitioning for a fast convergence of the refinement process, with a minimal number of inserted vertices needed. Additionally, with regards to parallelization, the approach prevents any dependencies among the partitioned faces.

This so-called Vertex Tracing approach was extended for application in a distributed environment. With a focus on the use of a standard hardware and heterogeneous networks, a central, dynamic load-balancing was implemented. In order to cope with bad transfer conditions and to achieve minimal communication load, we proposed a load balancing based on the object geometry, which is compatible with the concept of the vertex tracer, proposed in this paper.

Interactive frame rates for preview quality visualization are already achieved with the single processor version of the vertex tracer. This, in connection with complex VR scenarios, fulfills the requirements of Virtual Reality simulations.

Nevertheless, a number of improvements of the overall system are necessary, in the future. For example, it will be necessary to improve the quality of the vertex tracing to limit the interpolation error arising from discontinuities and nonlinearities[3] as well as to exploit the scene coherence to further improve the computation rate.

In the area of parallelization, also the expansion of the load balancing on the overall scene plays a crucial role. Here better clustering strategies will be necessary, in order to group the overall scene into *spatial* sectors. Also, a combination of shared and distributed memory system rendering has not been considered, so far. The use of several shared memory machines could reduce the communication problem and thus realize an even better speed-up with a larger number of processors.

References

1. T. Akimoto, K. Mase, Y. Suengaga. *Pixel-selected ray tracing*. IEEE Computer Graphics, 11:14-22, 1991
2. John Amanatides, Andrew Woo. *A Fast Voxel Traversal Algorithm for Ray Tracing*. Dept. of Computer Science, University of Toronto, 1987
3. Kavita Bala. *Radiance Interpolants for Interactive Scene Editing and Ray Tracing*. Dissertation, Massachusetts Institute of Technology, 1999
4. Daniel Beier. *Parallel progressives VertexTracing mit dynamischen Load-Balancing*. Diplomarbeit, TU Ilmenau, 2001.
5. M. Buck. *Simulation interaktiv bewegter Objekte mit Hinderniskontakten*. PhD thesis, University of Saarland, Germany, 1998.
6. J. Eckstein. *Echtzeitfähige Kollisionserkennung für Virtual Reality Anwendungen*. PhD thesis, University of Saarland, Germany, 1998.
7. Y. Eldar, M. Lindenbaum, M. Porat, Y. Zeevi. *The farthest-point strategy for progressive image sampling*. In Proceddings, 12th International Conference on Pattern Recognition, Jerusalem, 1994.
8. Andrew S. Glassner. *An Introduction to Ray Tracing*. Academic Press, London, 1989.
9. Andrew S. Glassner. *Principles of Digital Image Synthesis*. Volume 1 + 2, Morgan Kaufmann Publishers, Inc., San Francisco, 1995.
10. Frederik W. Jansen, Jarke J. van Wijk. *Fast Previewing Techniques in Raster Graphics*. Proceeding Eurographics '83, S.195-202.
11. Leif P. Kobbelt, K. Daubert, H-P. Seidel. *Ray Tracing of Subdivision Surfaces*. Eurographics Rendering Workshop'98 Proceedings.
12. Irena Notkin, Craig Gotsman. *Parallel Progressive ray tracing*. Computer Graphics Forum, 16(1), Seiten 43-55, 1997.
13. Tom McReynolds, David Blythe. *Lighting and Shading Techniques for Interactive Applications*. Siggraph '99 Course 12, 1999.
14. J. Painter, K. Sloan. *Antialiased Ray Tracing by Adaptive Progressive Refinement*. Computer Graphics, 23:281-287, 1989.
15. S. Parker, W. Martin, P. Sloan, P. Shirley, B. Smits, C. Hansen *Interactive Ray Tracing*. Interactive 3D, April 1999
16. F. Pighin, Dani Lischinski, David Salesin. *Progressive Previewing of Ray-Traced Images Using Image-Plane Discontinuity Meshing*. Rendering Techniques 1997, S.115-126, 1997.
17. James Painter, Kenneth Sloan. *Antialiased Ray Tracing by Adaptive Progressive Refinement*. Computer Graphics (SIGGRAPH '89 Proceedings), S.281-288, 1989.
18. F. P. Preparata, M. I. Shamos. *Computational Geometry: An Indroduction*. Springer-Verlag, 1985.
19. A. Reisman, C. Gotsman, A. Schuster. *Interactive-Rate Animation Generation by Parallel Progressive ray tracing on Distributed-Memory Machines* Proceedings of The Parallel Rendering Symposium, Phoenix, October 1997.
20. Alexander Schmidt. *Ein Verfahren für die interaktive Simulation von Reflexion auf konkaven Oberflächen*. Diplomarbeit, TU Ilmenau, 2000.
21. Alan Watt, Mark Watt. *Advanced Animation and Rendering Techniques, Theory and Practice*. Addison-Wesley Publishing Company, Inc., 1992.
22. H. Weghorst, G. Greenberg. *Improved Computational Methods for Ray Tracing*. ACM Trans. Graphics, 52-69.
23. Turner Whitted. *An Improved Shading Illumination Model for Shaded Display*. in Communication of the ACM, 23(6), S.343-349, 1980.

3

2, 6

2, 3

2

3

3

2

2, 3, 5

2

2

2

3

2

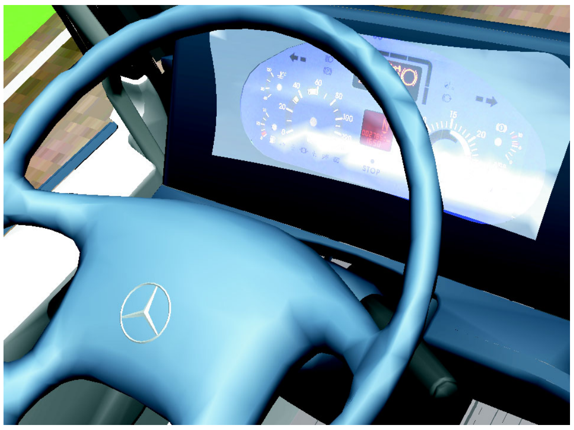

Figure 10: *Preview representation*

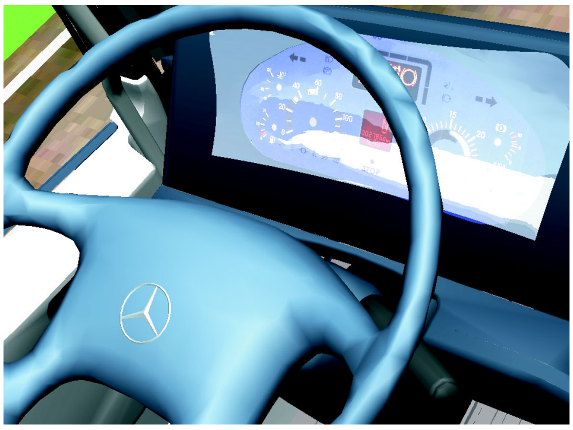

Figure 12: *Representation with maximum refinement and with triangle filling*

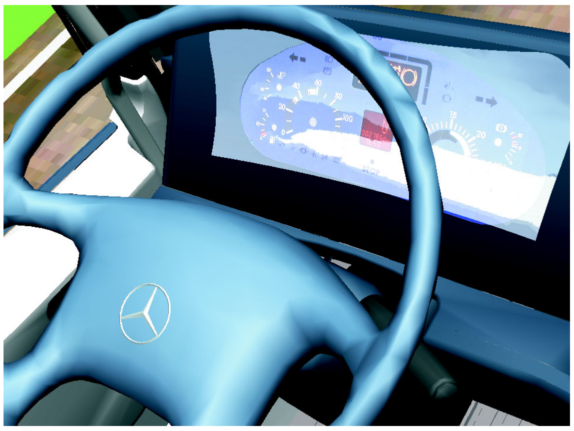

Figure 11: *Representation with maximum refinement and without triangle filling*

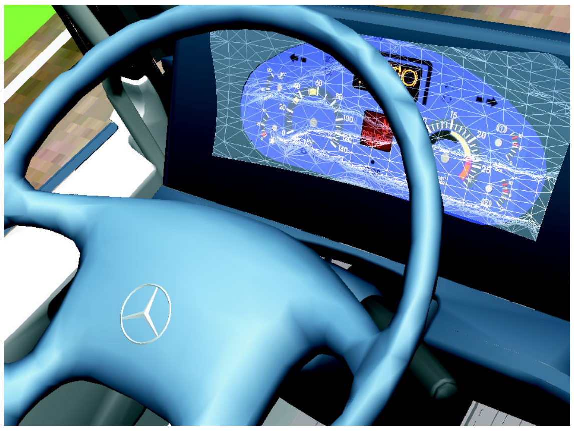

Figure 13: *Representation of generated triangles*