Layered Environment-Map Impostors for Arbitrary Scenes

Stefan Jeschke
Institute of Computer Graphics and Algorithms
Vienna University of Technology

Michael Wimmer
Institute of Computer Graphics and Algorithms
Vienna University of Technology

Heidrun Schuman
Institute of Computer Graphics
Department of Computer Science
University of Rostock

Abstract

This paper presents a new impostor-based approach to accelerate the rendering of very complex static scenes. The scene is partitioned into viewing regions, and a layered impostor representation is precalculated for each of them. An optimal placement of impostor layers guarantees that our representation is indistinguishable from the original geometry. Furthermore the algorithm exploits common graphics hardware both during preprocessing and rendering. Moreover the impostor representation is compressed using several strategies to cut down on storage space.

Key words: virtual environments, walkthroughs, image-based rendering, impostors, environment maps

1 Introduction

The real-time display of very large and complex virtual environments has been one focus of computer graphics research for a considerable amount of time already. Although many advances have been made, the goal of displaying arbitrary (i.e., without restrictions on scene structure or navigation methods) scenes of high complexity at frame rates above 30 or even 60 Hz has consistently eluded any approach.

For instance, visibility calculations [25] can dramatically reduce the geometry to be rendered in walkthroughs of densely occluded environments. However, many scenes do not provide sufficient occlusion, and even for densely occluded scenes, switching to a different navigation behavior (like flyovers) can render visibility culling ineffective. In such cases, image-based rendering (IBR) [7] methods are usually called for, because their rendering complexity only depends on the output resolution of the image, not on the total number of primitives in the scene.

In this paper, we present a new algorithm that uses several optimally placed layers of image-based primitives—so-called *impostors*—to represent distant geometry. Each layer is arranged in a fashion similar to a cubic environment map. The layered impostors are generated in a preprocessing step for individual regions of space called *view cells*, and further optimized so that they take up little storage space and can be processed efficiently by current graphics hardware. A new error metric guarantees that the representation is practically indistinguishable from the original geometry it replaces, avoiding cracks and popping artifacts. Geometry near the viewer will be displayed using polygonal rendering.

Figure 1 shows an example of our impostor technique.

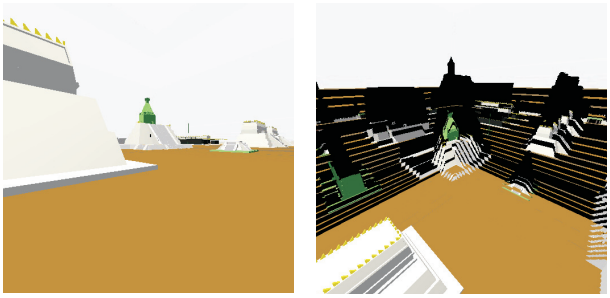

Figure 1: Example for layered environment-map impostors: (a) observer's view, (b) same scene from a bird's eye view.

The main contribution of this paper is an image-based geometry representation method that offers several advantages over previous methods:

1. It can deal with arbitrary static models with diffuse illumination, even if no scene structure is available. This is especially important because most available models consist of unstructured "polygon soups", which cannot be dealt with satisfactorily in most other methods.

2. High output image quality. The optimal layer place-

ment shown in this paper guarantees that the differences between impostor and the geometry it represents are practically imperceptible. In particular, it avoids image cracks due to missing information about hidden geometry and popping artifacts when switching between different impostors.

3. Compact impostor representation. Although the precalculated impostors for all view cells need to be stored on hard disk, the specific layer arrangement allows using special-purpose algorithms that dramatically reduce memory requirements both on harddisk and in texture memory.

4. Fast rendering. The method naturally supports conventional graphics hardware. Additionally, since no online calculations are necessary, optimal runtime efficiency is achieved.

The remainder of the paper is organized as follows: after a short review of previous work in section 2, an overview of the impostor system is given in section 3. Section 4 introduces the error metric used to calculate the optimal layer placement, and section 5 presents the techniques used to compress the impostor textures. Results of the system are given in section 6, and section 7 presents final conclusions.

2 Previous work

A huge amount of literature deals with methods to accelerate the real-time rendering of highly complex environments, including level-of-detail rendering [10], visibility culling [25], point-based rendering [24], light-field rendering [14], or image warping [7], to name just a few. Many of them share similar problems: they are not general enough for arbitrary scenes (e.g., visibility culling, see the introduction), or are not amenable to hardware acceleration, as is the case for most image-based rendering methods. We will review here only a subset of image-based methods that are directly related to our work.

The idea of using image-based representations to replace complex geometric objects in virtual walkthroughs was first introduced by Maciel in 1995 [15]. A particular object is rendered into a texture map with transparency information, and then mapped onto a quadriliteral placed into the scene in place of the object. The resulting primitive is called an *impostor*. Schaufler et al. [20] and Shade et al. [22] used this idea to build a hierarchical image cache for an online system, with relaxed constraints for image quality.

When using only one quadriliteral, the object is represented poorly by the impostor if the observer moves too far away from the reference viewpoint. Therefore, several authors have presented methods that use varying levels of geometric information to overcome this drawback. Starting from a planar impostor, depth information can be added using triangles [8, 16, 23], with the resulting primitive sometimes called textured depth mesh (TDM). TDMs are, however, prone to distortion and disocclusion artifacts. Depth can also be added per point sample—in particular, layered depth images (LDI) [17, 21] provide greater flexibility by allowing several depth values per image sample. However, LDIs contain more information than necessary for a good representation of parallax effects, and are not amenable to hardware acceleration. TDMs [1] and LDIs [2] can also be used to represent distant geometry for a whole view cell.

Finally, in a way closely related to our work, depth information can be added using layers [13, 18, 19]. There, an object is sliced into several impostors, each of them representing a different depth range. However, this technique was only used to simplify individual objects in a scene, and it is based on equidistant layers.

In this paper, we show how to use layered impostors to simplify distant geometry, and how to optimally place the layers so as to provide the highest possible image quality [5], while at the same time keeping storage cost to a tolerable level using an image-based visibility algorithm which is conceptually related to the extended projections technique [9].

3 Overview

Our system consists of two stages: a preprocessing stage in which layered impostors are calculated and optimized for all view cells in the scene, and a runtime component in which the compressed impostors are prefetched on demand and displayed in place of the original geometry. Each layer is arranged similar to a cubic environment map around the view cell.

More specifically, the *preprocessing stage* consists of the following steps:

1. The user needs to decide on the size of the view cells, on the number of layers, and on the display resolution the impostors should be calculated for.

2. Partition the space of possible viewing positions into view cells.

3. Generate the layered impostors in the following steps:

   (a) Determine the distance to the individual impostor layers (section 4.2) and the regions for which the impostors are used.

   (b) Render the geometry for each impostor layer into a texture using conventional graphics hardware.

(c) Compress the impostor textures by exploiting the special structure of the problem (section 5):

i. Erase those parts of every impostor texture that are always occluded by impostor layers closer to the view cell (section 5.2).

ii. partition the impostor polygon into smaller polygons that are well filled.

iii. combine the textures for the smaller polygons into larger textures for efficient graphics hardware treatment.

iv. compress those textures using PNG for efficient storage on disk.

(d) Finally, a list of the near geometry that is not represented by impostors is generated.

The *runtime component* is a simple 3D viewer where the user can freely navigate through the 3D model. Since most of the work was already done during preprocessing, the runtime component only has to:

1. Determine the current view cell of the observer.

2. Prefetch the geometry and impostors of adjacent view cells in order to avoid varying frame rates [11] (assuming—as is common for most prefetching schemes—a limited observer speed). To amortize the cost of texture downloads over several frames, prefetched impostor textures are downloaded to texture memory in smaller chunks.

3. Render the near geometry not represented by impostors.

4. Render the impostor polygons for the current view cell.

Note that the last two steps are done using graphics hardware.

4 Optimal impostor-layer placement

This section explains how the impostor layers are arranged around a view cell and how the optimal placement is calculated.

4.1 Layer arrangement

Figure 2 shows how impostor layers are arranged as *impostor cubes* around a particular view cell. Note that each impostor layer represents geometry within a certain depth range to the front and to the back of the layer. The *borders* show where the transition between two layers takes place: objects in front of a particular border are represented by the impostor nearer to the view cell, objects to the back are represented by the impostor farther from the view cell.

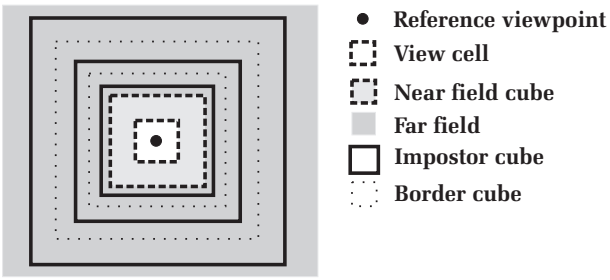

Figure 2: Layout of the impostors around a view cell.

When generating an impostor layer, each side of its impostor cube is rendered from the *reference viewpoint*, i.e., the center of the view cell, with the near and far clipping planes set to the appropriate sides of the adjacent border cubes.

The innermost border cube (near field cube in figure 2) is especially noteworthy: it defines the border between the *near field* and the *far field*. All geometry in the near field will be rendered using polygonal rendering, whereas geometry in the far field is represented by impostors.

While the size of the view cell is defined by the user directly, the placement of the impostor layers and their corresponding depth ranges (i.e., position of the borders) will be calculated automatically. The only user inputs to this calculation are the desired number of impostor layers and the desired image resolution.

4.2 Layer placement calculation

Two errors have to be taken into account when calculating an optimal placement of the layers: parallax errors and gaps between texels of consecutive layers.

Parallax errors: The impostor layers need to be placed so that every layer "faithfully" represents the geometry it replaces as long as the observer stays within the associated view cell. In order to quantify "faithfully", we characterize the error that occurs when viewing the impostor from a position different from the reference position using the *parallax angle* α (see figure 3). This is the angle between the true 3D position of a point F and its projection F' to the impostor, when seen from a position V different from the reference viewpoint V_r (see also [22]).

The goal is to find out in which configuration the maximum parallax error occurs, so that we can calculate from this the maximum possible depth range that guarantees that a given parallax angle will not be exceeded. Therefore, while the parallax angle can be calculated for any viewing position and any point on an object, we are actually only interested in the extremal case where the viewpoint is moved to a corner of the view cell, and we only

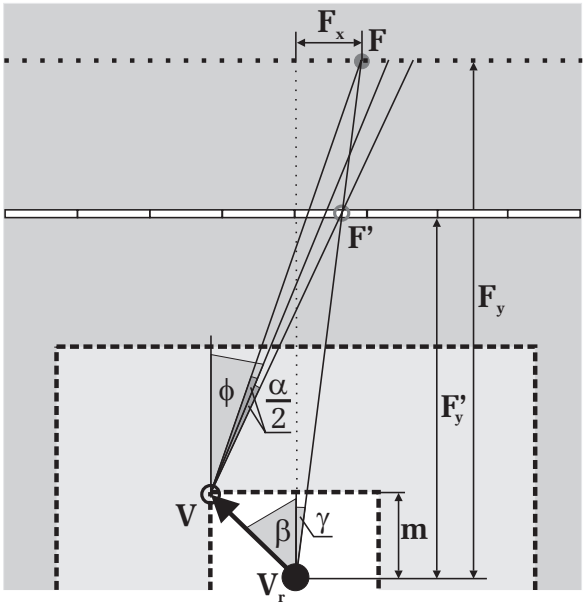

Figure 3: Maximum parallax angle within a view cell.

consider points on a border cube, i.e., points on a plane parallel to the impostor. For this case, a very intriguing result can now be shown[1]:

> Assume a given view cell, an impostor layer and a border cube. Then the maximum parallax angle between a point on the border cube and its projection to the impostor, when seen from the corner of the view cell, *always* occurs for a viewing direction of $\phi = \frac{\pi}{4} - \frac{\beta}{2}$ from the view cell corner, where $\beta = \frac{\pi}{4}$ for the 2D case and $\beta = \arctan(\sqrt{2})$ for the general 3D case (see figure 3).

The following equations show how to calculate from a known border F_y the next closer layer F'_y (or respectively from a known layer F_y the next closer border F'_y):

$$\begin{aligned} m &= \frac{size_{viewcell}}{2}; \\ F_x &= \tan(\phi - \frac{\alpha}{2})(F_y - m) - m\tan(\beta); \\ \gamma &= \arctan(\frac{F_x}{F_y}); \\ F'_y &= \frac{m\cos(\gamma)\sin(\phi + \frac{\alpha}{2} + \beta)}{\cos(\beta)\sin(\phi + \frac{\alpha}{2} - \gamma)}. \end{aligned} \quad (1)$$

When calculating the following border (or respectively layer), F_y must simply be set to F'_y from the previous step. To obtain the distance to the first impostor layer, F_y can be set infinitely far away, where γ becomes $\phi - \frac{\alpha}{2}$.

Gaps between texels: Two successive impostor layers should not move more than one texel against each other because visibility gaps might appear in a continuous surface represented in both layers.

As can easily be shown, the maximum texel movement appears when the observer looks from one corner of the view cell to texels at the opposite border of the impostor as shown in figure 4.

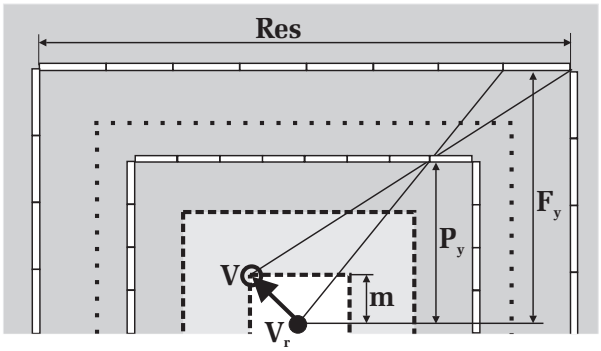

Figure 4: Maximum pixel movement within a view cell.

The minimum allowable distance to the next closer impostor layer, P_y, is then

$$P_y = \frac{F_y\, m\, Res}{m\, Res + F_y - m}, \quad (2)$$

where *Res* is the resolution of an impostor layer. This value is compared to the value F'_y obtained by two successive evaluations of equation 1. The larger of the two defines the actual distance of the impostor layer. If P_y is chosen, then the border between the two layers can be recalculated using equation 1 by setting α to

$$\alpha = 2\arctan\left(f\cos(\beta)\sqrt[4]{\frac{P_y(F_y - m)}{F_y(P_y - m)}} - \tan(\beta)\right) - 2\phi.$$

Here $f = 2$ in the 2D case and $f = 3$ in the 3D case.

4.3 Discussion

Choosing α: An obvious choice for α is the minimal angle subtended by a pixel in the output image.

Choosing the number of layers: Using only one impostor for the far field is equivalent to a conventional cubic environment map, except that the error incurred by using the environment map is a maximum of one pixel. The resulting near field is too large for most practical purposes, however. For example, an output resolution of 512x512

[1] For a detailed proof of this statement see [12].

pixels and a view cell size of 10x10 m would result in a near field distance of 1738 m.

Using more layers, which represent different depth ranges of the far field, dramatically reduces the size of the near field because the parallax movements are "split" and assigned to different layers. For example, with the same output resolution of 512x512 pixels and a view cell size of 10x10 m, using 64 layers will place the near field distance already at 42 m.

5 Impostor compilation

The layered impostors generated for each view cell need to be compressed before they can be used. For example, the impostors for a single view cell would require 384 MB at a resolution of 512x512 texels and using 64 layers. Therefore, we apply several methods to reduce the storage requirements of impostors on harddisk and, even more importantly, in texture memory. Finally, before generating the impostor geometry, it is necessary to close one-pixel visibility gaps that might arise between two impostor layers.

The methods presented here exploit the fact that all impostor layers use the same resolution, and that two adjacent layers only move by a maximum of one pixel against each other in any view. This allows us to use fast image processing techniques as described in the following sections.

5.1 Gap filling

The aim of gap filling is to prevent holes from appearing in continuous surfaces represented by different layers. Figure 5 (left) shows the problem in 1D: The line

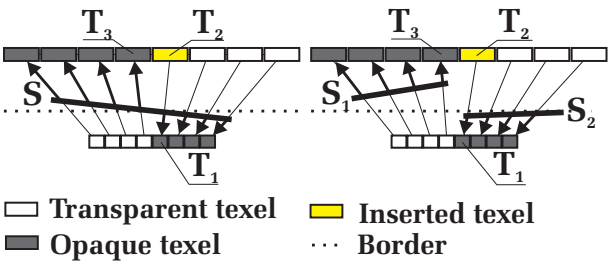

Figure 5: Filling the appearing gap between T_1 and T_3 by copying the texel T_1 to T_2. This may result in correct (left) or incorrect (right) visibility.

S is sampled so that the texel T_2 is not opaque. If the viewer now moves from the center of the view cell to the left, a hole of one texel appears between the near and the far layer. To avoid such undesirable artifacts, we fill the texel T_2 by copying the color information from texel T_1 to it. Thus we can always guarantee that continuous surfaces will have no holes.

The right side of figure 5 shows a case where the filling of T_2 results in incorrect visibility, but the error introduced thereby is usually not noticeable. It is not possible to avoid this problem, since after sampling the geometry, it is not possible to distinguish between the two cases.

In practice, the hole-filling operation is done easily by copying opaque texels that have at least one transparent neighbor (*opaque border texels*) of the current (closer) layer to texels at the same position in the next (more distant) layer, that are either opaque texels or *transparent border texels* (transparent texels that have at least one opaque neighbor).

5.2 Removing invisible texels

Normally, many texels in each layer are occluded by closer layers[2]. Therefore, we remove texels that are never visible because they are always hidden by texels of closer layers.

The algorithm for doing that proceeds by repeating the following operations for each pair of adjacent layers (see figure 6 for the problem in 1D, (a) shows the initial configuration):

Mark texels in the more distant layer as hidden if they are behind opaque texels of the closer layer which are not border texels (see figures 6 (b) or (c)). If texels in the closer layer were already marked as hidden in the previous iteration, they must be interpreted as opaque in this iteration (see figure 6 (c)) and set as transparent before proceeding to the next pair of layers (this can be seen in figure 6 (d)).

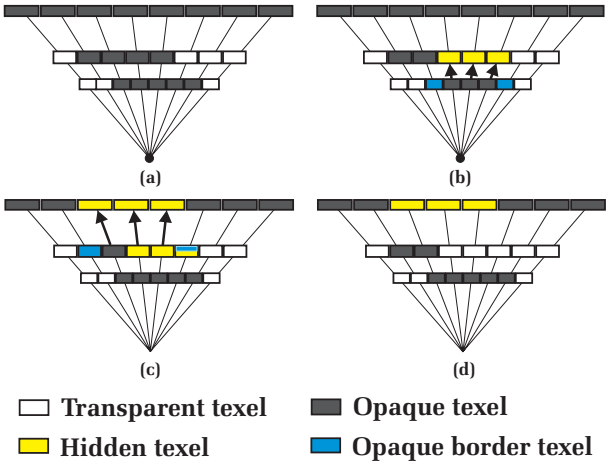

Figure 6: Removing texels in the layers that are hidden behind layers closer to the viewpoint.

[2]Note that occlusion due to near-field geometry is very complex and not addressed in our system.

After removing invisible texels, the layers contain exactly the information that might become visible when the viewer walks within the associated view cell.

5.3 Generation of impostor polygons

Since occluded texels have been removed in the previous step, most layers will contain a large amount of transparent texels that actually need not be stored at all. Therefore, we extract the opaque texels of every layer by splitting the texture into a number of smaller textures (*micro textures*) that tightly cover the opaque regions in the impostor texture, and creating corresponding rectangular impostor polygons (see the example in figure 7). The goal is to find a good tradeoff between minimizing the number of impostor polygons and the number of transparent texels in the microtextures.

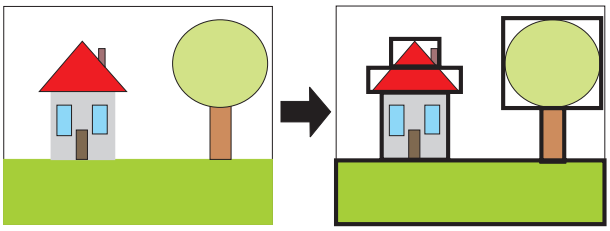

Figure 7: Representation of an impostor layer with only few covering polygons.

Since this problem is NP-complete, we employ a fast, greedy algorithm to find a good solution. First, overlay the impostor texture with a regular grid—with smaller grid size giving preference to a more accurate solution, but creating more polygons (we used an 8x8 grid). Then, as long as there is such a cell left, we choose a cell that is not completely transparent and try to grow it until the percentage of opaque pixels in the cell falls below a user-defined threshold. The cell can be grown in any direction (including diagonals), but is constrained to maintain rectangular shape. In each growing step, the direction that gives the best percentage of opaque pixels is chosen. If the cell cannot be grown further (either because there are no non-transparent cells in any direction, or because the threshold is reached), a microtexture and its corresponding impostor polygon is created, and all its grid cells are set to transparent.

Even tighter packing might be achieved using more involved algorithms [4] at the expense of longer preprocessing time.

5.4 Microtexture packing

As the number of microtextures created for a view cell in the previous step can be quite large (hundreds or even thousands, depending on the parameters), we try to pack microtextures into larger *macrotextures*. This reduces the general overhead involved in handling a large number of textures, and especially favors current graphics hardware which penalizes textures switches. Furthermore, we can account for the fact that graphics hardware can usually only handle textures with resolutions of powers of two (this restriction can be lifted for newer hardware).

Again, since the 2D rectangle packing problem is NP-complete [3], we use a fast, greedy algorithm that tries to generate very few macrotextures, while achieving a good coverage of the textures with opaque texels. First, all microtextures are rotated so that they are top-down. Afterwards they are sorted into a list by decreasing height and—for equal heights—by increasing width. The height of a new macrotexture is then determined using the height of the first microtexture in the list enlarged to the next power of two. The width of the macrotexture is calculated by dividing the summed area of all microtextures by the determined height and also enlarge this value to the next power of two (assuming the first microtexture will fit).

For filling the new macrotexture with microtextures, a bottom-left placement rule [6] is used beginning with the first in the sorted list, until the right border of the macrotexture is reached. The remaining space is filled by the remaining microtextures using a left-bottom placement rule beginning with the last (smallest) microtexture in the list.

After all microtextures are tested and possibly placed, the algorithm resumes by constructing the next macrotexture. This is repeated until there is no microtexture left.

The final result is a low number of relatively well-filled macrotextures that can be sent directly to graphics hardware. For storing them on disk, they are compressed using PNG compression.

6 Results

We have implemented the methods presented in this paper in C++ and OpenGL, and applied them to a 3D model of the ancient Aztec city of Tenochtitlan (which is freely available on the Internet). In order to provide adequate complexity, we replicated the model 3x3 times for a total of 854586 polygons. The Aztec model demonstrates the performance of the method in a scene with wide open spaces and sparse buildings, and shows that the method performs equally well for walkthrough situations and flyovers. A PC with an Athlon 1.2 GHz processor, 512 MB of main memory and a GeForce 3 graphics card was used for all tests.

Table 1 shows some statistics for the model. A corridor of 11x27 view cells (i.e., 154x378 m) was selected for preprocessing, which took slightly above 12 hours in our unoptimized implementation. Preprocessing the

whole model would have taken prohibitively long, so—as is usual for methods involving preprocessing—it is advisable to carefully select the areas where the observer is expected to move around.

Polygons of whole model	854586
Side length of whole model	1350 m
Side length of view cell	14 m
Side length of near field cube	118 m
Avg. preproc. time per view cell	2.5 min

Table 1: Model statistics

Size of impostor layers	256 MB
Size of macrotextures	6018KB
Size of macrotextures after PNG	288 KB

Table 2: Effect of compression on avg. per view cell

The selected output resolution was 512x512 pixels, and the number of layers was set to 64. Note that since we move relatively near to the ground plane and the sky is not complex enough to warrant an efficient impostor representation, we only calculated impostors for the 4 directions orthogonal to the ground plane. Table 2 shows the compression achieved on average when going from the layers of a view cell to macrotextures, and after PNG compression of the macrotextures, for these selected parameters.

In order to test the runtime performance of the system, we have recorded a path (with both walkthrough and flyover characteristics) through the preprocessed part of the model, containing 2197 frames. Figure 8 shows the frame rates our system achieves for each frame of the path, and for comparison, the frame rates of an unaccelerated system using only OpenGL display lists. It can be seen that

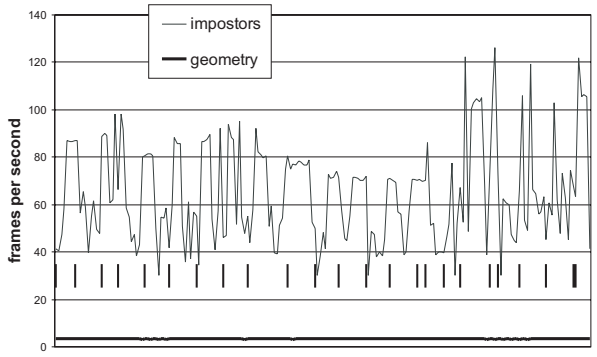

Figure 8: Frame rates for the recorded path containing 2197 frames. The vertical lines indicate where the view cell changed.

while the unaccelerated system never exceeds 3 frame per second (fps), our layered impostor system consistently reaches 40 fps, with an average of of more than 60 fps. This means that our system achieves a speedup of more than one order of magnitude over the unaccelerated system.

Finally, figure 9 shows the number of polygons used for impostors, the number of polygons remaining in the near field, and the total number of polygons for each frame of the same walkthrough.

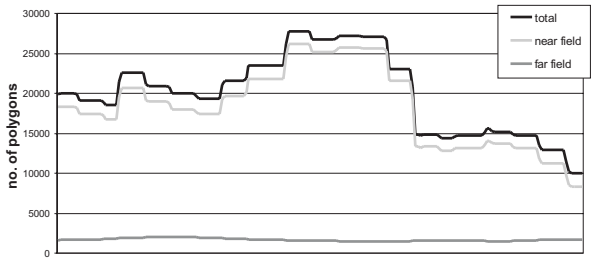

Figure 9: Complexity of the near field and far field for the recorded path.

Reconstruction quality is limited to the sampling provided by OpenGL at the moment, and includes an additional bias towards the reference viewpoint, which is however negligible in practice due to our error metric. While it would theoretically be possible to improve reconstruction quality by storing view-dependent information in the textures and using multiple rays for filtering [24], the expected preprocessing times and space required for the additional texture information would be prohibitively high.

7 Conclusions

We have presented a novel approach to render very complex, arbitrary virtual environments. The system represents distant geometry as layered textured polygons arranged similar to environment maps. The major contributions of the paper are twofold:

First, the impostor layers are optimally placed so that the parallax error remains bounded. This is particularly advantageous for distant geometry, since the layers can be placed increasingly farther apart. In addition, this placement guarantees that the output image never differs by more than a pixel from an image obtained with the original geometry.

Second, the special-purpose algorithms for compressing the generated impostor textures achieve a significant reduction in storage space. In particular, these algorithms also reduce the amount of texture memory required for

impostor textures on the graphics hardware, which is crucial insofar as this allows textures for several view cells to be held in texture memory simultaneously, allowing prefetching.

In general, the method can be said to decouple the rendering speed of the far field from its geometric complexity. Note especially that the placement of the farthest layer—and therefore the number of layers required—does not depend on the extent of the scene (which can be arbitrarily large), but only on the desired output resolution. It also exploits the considerable polygon-rendering and texture-mapping capabilities of today's graphics hardware by using standard polygonal rendering for the near field (where using impostors would not be feasible because of the large parallax errors), and impostor polygons for the far field, which would overwhelm the graphics hardware if rendered using geometry alone.

We believe that layered environment-map impostors can be used to render models of hitherto unseen complexity. The method is especially interesting in applications where conventional acceleration methods such as visibility culling, level-of-detail rendering or previous image-based rendering methods fail due to the nature of the scene (e.g., only little occlusion), the navigation metaphor (e.g., flyovers), or missing structure information about the scene (e.g., polygon soups). However, since complexity is basically shifted into the preprocessing phase, some attention should be given to the choice of the possible viewing region. While the preprocess can be trivially parallelized to run on multiple PCs, a large possible viewing region can take prohibitively long to compute on a single computer.

In terms of future research, it is desirable to have an automatic method for choosing the size and the arrangement of the view cells. Furthermore, as is typical of preprocessing methods, the algorithm partly shifts the necessary processing-power for rendering a model from the pure polygon throughput of the graphics hardware towards the bandwidth of the bus between main memory and graphic system, and to the CPU. We are therefore investigating novel prefetching algorithms to make best use of this tradeoff.

Acknowledgements

This work was supported by the German Research Foundation (DFG) in the frame of the postgraduate program "processing, administrating, visualization and transfer of multimedia data—technical basics and social implications", and the Austrian Science Fund (FWF) contract no. P13867-INF.

References

[1] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *1999 Symposium on interactive 3D Graphics*, pages 199–206, 1999.

[2] D. Aliaga and A. Lastra. Automatic image placement to provide a guaranteed frame rate. In *SIGGRAPH 99 Conference Proceedings*, pages 307–316, 1999.

[3] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest. Orthogonal packings in two dimensions. In *Proc. 16th Annual Allerton Conf. on Communication, Control, and Computing*, pages 626–635, 1978.

[4] Becker, Franciosa, Gschwind, Ohler, Thiemt, and Widmayer. An optimal algorithm for approximating a set of rectangles by two minimum area rectangles. In *CGMAA: Computational Geometry–Methods, Algorithms and Applications*, 1991.

[5] J. Chai, X. Tong, S. Chan, and Heung-Yeung Shum. Plenoptic sampling. In *SIGGRAPH 2000 Conference Proceedings*, pages 307–318, 2000.

[6] B. Chazelle. The bottom-left bin-packing heuristic: An efficient implementation. *IEEE Transaction on Computers C-28(8)*, pages 697–707, 1983.

[7] S. Chen. QuickTime VR – an image-based approach to virtual environment navigation. In *SIGGRAPH 95 Conference Proceedings*, pages 29–38, 1995.

[8] L. Darsa, B. Costa Silva, and A. Varshney. Navigating static environments using image-space simplification and morphing. In *1997 Symposium on Interactive 3D Graphics*, pages 25–34, 1997.

[9] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. In *SIGGRAPH 2000 Conference Proceedings*, pages 239–248, 2000.

[10] R. Scopigno E. Puppo. Simplification, lod and multiresolution - principles and applications. In *Eurographics'97 Tutorial Notes PS97 TN4*, pages 31–42, 1997.

[11] T. A. Funkhouser, C. H. Sequin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. In *1992 Symposium on Interactive 3D Graphics*, volume 25, pages 11–20, March 1992.

[12] S. Jeschke and M.Wimmer. An error metric for layered environment-map impostors. Technical Report TR-186-2-02-04, Vienna University of Technology, 2002.

[13] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH 94 Conference Proceedings*, pages 451–458, 1994.

[14] Marc Levoy and Pat Hanrahan. Light field rendering. In *SIGGRAPH 96 Conference Proceedings*, pages 31–42, 1996.

[15] P. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *1995 Symposium on Interactive 3-D Graphics*, pages 95–102, 1995.

[16] W. Mark, L. McMillan, and G. Bishop. Post-rendering 3D warping. In *1997 Symposium on Interactive 3D Graphics*, pages 7–16, 1997.

[17] N. Max and K. Ohsaki. Rendering trees from precomputed Z-buffer views. In *Rendering Techniques '95*, pages 74–81, 1995.

[18] A. Meyer and F. Neyret. Interactive volumetric textures. In *Rendering Techniques '98*, pages 157–168, 1998.

[19] G. Schaufler. Per-object image warping with layered impostors. In *Rendering Techniques '98*, pages 145–156, 1998.

[20] G. Schaufler and W. Stürzlinger. A three-dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–235, 1996.

[21] J. Shade, S. Gortler, L. He, and R. Szeliski. Layered depth images. In *SIGGRAPH 98 Conference Proceedings*, pages 231–242, 1998.

[22] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH 96 Conference Proceedings*, pages 75–82, 1996.

[23] F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum*, 16(3):207–218, 1997.

[24] M. Wimmer, P. Wonka, and F. Sillion. Point-based impostors for real-time visualization. In *Rendering Techniques 2001*, pages 163–176, 2001.

[25] P. Wonka, M. Wimmer, and F. X. Sillion. Instant visibility. *Computer Graphics Forum*, 20(3):411–421, 2001.