Eurographics Workshop on Rendering (2002)
Paul Debevec and Simon Gibson (Editors)

Textured Depth Meshes for Real-Time Rendering of Arbitrary Scenes

Stefan Jeschke and Michael Wimmer

Institute of Computer Graphics and Algorithms, Vienna University of Technology

Abstract

This paper presents a new approach to generate textured depth meshes (TDMs), an impostor-based scene representation that can be used to accelerate the rendering of static polygonal models. The TDMs are precalculated for a fixed viewing region (view cell).

The approach relies on a layered rendering of the scene to produce a voxel-based representation. Secondary, a highly complex polygon mesh is constructed that covers all the voxels. Afterwards, this mesh is simplified using a special error metric to ensure that all voxels stay covered. Finally, the remaining polygons are resampled using the voxel representation to obtain their textures.

The contribution of our approach is manifold: first, it can handle polygonal models without any knowledge about their structure. Second, only scene parts that may become visible from within the view cell are represented, thereby cutting down on impostor complexity and storage costs. Third, an error metric guarantees that the impostors are practically indistinguishable compared to the original model (i.e. no rubber-sheet effects or holes appear as in most previous approaches). Furthermore, current graphics hardware is exploited for the construction and use of the impostors.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation – Display algorithms, I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – Virtual reality

1. Introduction

One main focus of computer graphics for a considerable amount of time already has been the real-time display of very large and complex geometric models. Many different approaches have been invented addressing this aim, including visibility calculations[20], level-of-detail approaches[13] or image-based rendering[2]. In many cases, visibility culling is not efficiently applicable due to insufficient scene occlusion, and level-of-detail rendering also cannot be used either, for instance because of unsuitable scene structure. In such cases, image-based rendering (IBR) methods are usually called for, because their rendering complexity mostly depends on the resolution of the output image, and not on the total number of primitives in the scene. Image-based entities used as alternative representations for fast rendering of scene parts are usually called *impostors*.

One special kind of impostors are *textured depth meshes* (TDM)[1, 18]: a texture with detailed scene appearance information is mapped to a simple polygonal mesh that emulates the rough scene structure, thus cutting down on rendering and storage costs for the represented scene part. The advantages of the TDM technique compared to other approaches are:

1. Parallax movements that appear when the observer moves in the scene are correctly represented so that the impostor is valid longer than simpler representations such as billboards[10].
2. The storage cost of TDMs is low compared to more complex image-based representations like layered impostors[14] or layered depth images[16].
3. Efficient treatment by graphics hardware is guaranteed due to the polygonal nature of TDMs in contrast to more complex representations[19], which are not directly amenable to hardware acceleration.

However, although TDMs are very useful to accelerate the rendering of large scene parts, their creation is in general a difficult task which made them to date hard to use in practical applications.

In this paper, a new algorithm will be presented that generates high-quality textured depth meshes for a fixed region of space called a *view cell*. The algorithm extends work presented by the authors in a previous paper[8]. Basically, it relies on the following principle: the scene is sampled using a special kind of slicing, thereby creating a layered voxel representation. In the previous approach, for every voxel layer a small set of polygons is computed that tightly covers the rendered voxels. The polygons are then placed into the scene, representing different ranges of depth to emulate parallax movements within the scene. Because layer spacing gets very tight in the vicinity of the view cell, the approach results in too many polygons when representing objects near the view cell.

This paper overcomes this drawback by extracting information about the connectivity between successive voxel layers: using the voxel field, polygon meshes are generated that are complex, but have a very simple structure and cover all the recorded voxels. After the scene is completely recorded, the resulting meshes are simplified. Finally, the voxel field is used to obtain textural information for the polygons. A special error bound guarantees that the final representation is practically indistinguishable from the original geometry it replaces.

The main contribution of this paper is an algorithm for generating textured depth meshes which offers several advantages over previous approaches:

1. It can deal with arbitrary static models, even if no scene structure is available.
2. The algorithm shown in this paper guarantees high output image quality by ensuring that the differences between the impostor and the geometry it represents are practically imperceptible. In particular, no image cracks or rubber-sheet effects due to missing information about hidden scene geometry or aliasing effects due to over- or undersampling of the scene appear at all.
3. Only scene parts that definitely become visible when the observer moves within the view cell are covered by the TDM. This results in an extremely compact impostor representation.
4. In contrast to the layer-aligned impostor representation introduced by the authors, it is now possible to compute a simple mesh consisting of very few textured polygons even for objects with a relatively small distance to the view cell.
   Through this, the geometric complexity of the impostor representation is totally decoupled from the number of voxel layers that where used to generate the TDM.

The remainder of the paper is organized as follows: after a short review of previous work in section 2, an overview of the TDM generation process is given in section 3. Section 4 introduces the algorithm to sample the scene and create the voxel field. Section 5 describes the generation and section 6 the simplification process of the meshes. After that, section 7 describes how the textural information for the polygons is computed using the voxel field, and section 8 gives the formulas for the transformation of the generated mesh into the scene. Results obtained with our approach are given in section 9, and section 10 presents final conclusions.

2. Previous work

Many image-based methods have been published in recent years that accelerate the real-time rendering of highly complex environments, including point-based rendering[19], light-field rendering[9], or image warping[2], to name just a few. We will review here only a subset of these methods that are directly related to our work.

The idea of using image-based representations to replace complex geometric objects was first introduced by Maciel in 1995[10]: a particular object is rendered into a texture map with transparency information, and then mapped onto a quadrilateral placed into the scene in place of the object. The resulting primitive is called an *impostor*. Schaufler et al.[15] and Shade et al.[17] used this idea to build a hierarchical image cache for an online system, with relaxed constraints for image quality.

When using only one quadrilateral, the object is represented poorly by the impostor if the observer moves too far away from the point the image was produced (the so-called *reference viewpoint*). Therefore, several authors have presented methods that use varying levels of geometric information to overcome this drawback: depth can be added in layers[14] or per point sample—in particular, layered depth images (LDI)[12, 16] 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.

Depth information can also be added using triangles[5, 11, 18], which results in textured depth meshes. To handle the problem of image cracks or rubber-sheet effects due to missing scene parts in the impostor representation, Decoret et al.[6] used several mesh layers of different depth to include hidden geometry. Furthermore their approach combines precalculated TDMs with online optimization to improve overall image quality. However, the approach is based on the restriction that the observer is only allowed to move along a line rather than inside a view cell and furthermore, no criteria guarantees that image cracks or rubber-sheet effects are avoided (the last drawback also applies to Darsa's[5] and Aliaga's[1] approach).

In this paper, a general geometry-sampling algorithm presented by the authors in previous work[8] is used for generating a high-quality textured depth mesh, drastically simplifying arbitrary geometry and at the same time providing high output image quality.

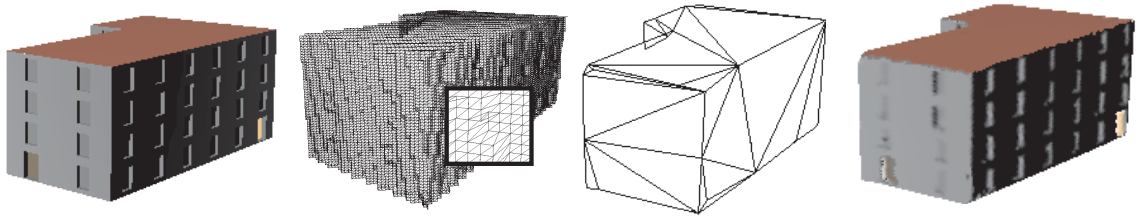

Figure 1: *Overview of our method for constructing textured depth meshes (from left to right): the original object (1880 polygons), the initial mesh constructed from the voxel field (61584 polygons), the simplified mesh (35 polygons), close-up view of the final textured mesh.*

3. Overview

The algorithm that generates the TDMs involves the following steps (see figure 1):

1. The user needs to decide on the size of the view cell, on the desired display resolution the TDM should be created for, and on the scene parts that should be represented by the TDMs.
2. A voxel field is generated by rendering the scene from the reference viewpoint using our previous algorithm, which will be covered briefly in section 4 for better understanding and for introducing slight changes pertaining to the present paper.
3. An initial (very complex) mesh that covers the whole voxel field is created (section 5).
4. The complex mesh is simplified (section 6).
5. Textures for the polygons of the simplified mesh are generated by sampling them in the voxel field (section 7).

4. Recording a voxel field

This section shortly reiterates the process of recording scene parts into a voxel field by slicing the scene into several *voxel layers* that represent different depths ranges. Modifications to the original algorithm are discussed, most notably the transformation of the final voxel field into a uniform grid in section 4.3.

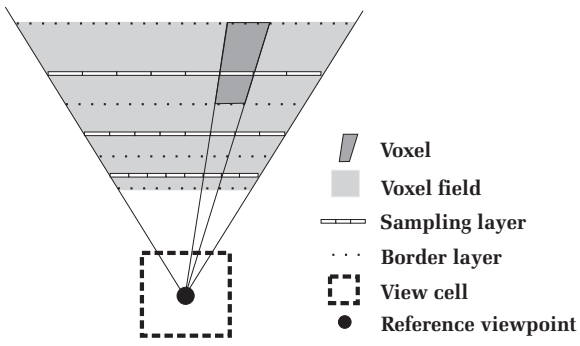

Figure 2: *Layout of the voxel field for a view cell.*

Figure 2 shows how voxel layers are arranged for a particular view cell (w.l.o.g. the algorithm is discussed for a cubic view cell). Note that each layer represents geometry within a certain depth range. The *sampling layers* are positioned in the middle (with respect to the *parallax angle* described further below) of each voxel layer. The *border layers* show where the transition between two layers take place: objects in front of a particular border are represented by the sampling layer nearer to the view cell, objects to the back are represented by the sampling layer farther away from the view cell.

When generating a voxel layer, the appropriate sampling layer is rendered from the reference viewpoint, with the near and far clipping planes set to the adjacent border layers. The colors of the rendered pixels in a sampling layer define the color of the respective voxel.

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

4.1. Layer placement calculation

Two errors have to be taken into account when calculating an optimal placement of the voxel layers: parallax errors and movements between voxels of different layers.

Parallax errors: The sampling layers need to be placed so that they "faithfully" represent the geometry they replace as long as the observer stays within the view cell. In order to quantify "faithfully", we characterize the error that occurs when viewing the sampling layer 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 sampling layer, when seen from a position V different from the reference viewpoint V_r (see also[17]). Obviously, if α is less or equal to the minimal angle subtended by a pixel in the output image for every voxel, the difference between the projected geometry

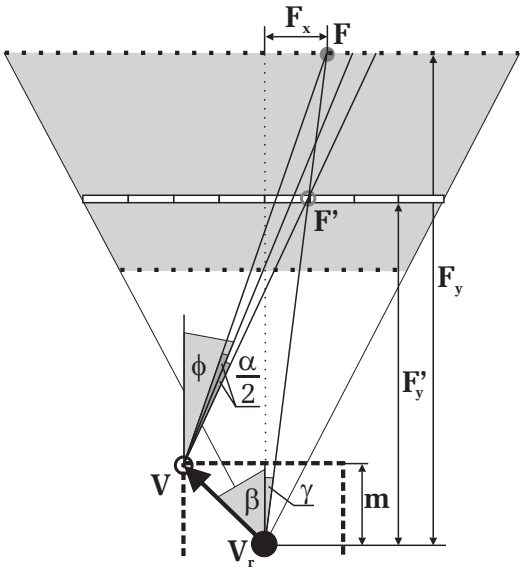

Figure 3: *Maximum parallax angle between a 3D point F and its projection F′ on the sampling layer within a view cell.*

and its voxel representation is not greater than one pixel in the output image.

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. As was stated previously[8], this maximum error occurs from the corner of the view cell under a viewing angle of ϕ between the point one of the border layers and its projection to the sampling layer (see figure 3).

Here, the original algorithm has to be modified: since for recording TDMs, the actual field of view is defined by the size and distance of the part of the scene to be recorded rather than using the special case of 90 degrees, F_x (figure 3) might exceed the view frustum to the left or right, as will occur for small and distant objects forming a narrow frustum. In this case, F_x is replaced by the intersection of the border layer with the respective view-frustum boundary. The remainder of the calculation for the parallax error is not affected.

Movements between pixels: Two successive sampling layers should not move more than the size of one pixel against each other. This restriction is the basis for the algorithm that removes invisible scene parts in the voxel field (see further below). Obviously, the maximum pixel movement appears when the observer looks from one corner of the view cell to pixels at the opposite side of the voxel field as shown in figure 4.

Again we cannot rely on a 90 degree field of view, so the minimum allowable distance to the next closer sampling

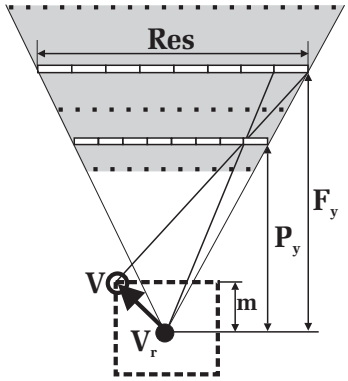

Figure 4: *Maximum pixel movement of a sampling layer within a view cell.*

layer, P_y (figure 4), should be evaluated by:

$$P_y = \frac{(\tan(\frac{FOV}{2}) + 1) \cdot F_y}{1 + \tan(\frac{FOV}{2}) \cdot (1 + 2 \cdot \frac{F_y - m}{RES \cdot m})}, \qquad (1)$$

where *Res* defines the resolution of the sampling layers and FOV is the field of view of the frustum enclosing the selected scene part. P_y must be calculated in both sampling directions (x and y) of the voxel field, and the larger value has to be chosen. It is then compared to the value F_y' (see figure 3, obtained by the calculation addressing the parallax error). The larger of the two defines the actual distance of the sampling layer.

For a detailed description of all necessary calculations and a discussion on the number of layers necessary to cover a certain depth range, our previous paper should be consulted[8].

4.2. Removing invisible scene parts

Normally, many voxels in a distant layer are occluded by voxels in layers closer to the view cell. The algorithm for removing those invisible voxels is briefly repeated here, followed by a description of crucial modifications necessary for the present paper.

The algorithm consists of the following steps, repeated for each pair of adjacent sampling layers (see figure 5 for the problem in 1D, (a) shows the initial configuration).

1. Mark pixels in the more distant sampling layer as hidden if they are behind opaque texels of the closer layer which are not border pixels (see figure 5 (b)). Pixels in the closer layer already marked hidden in a previous step must be interpreted as opaque in this step (figure 5 (c)).
2. Set all hidden pixels in the closer layer as transparent, thus excluding them from further consideration (figure 5 (d)). Repeat step one with the next more distant layer.

In our previous approach, gaps between layers were filled

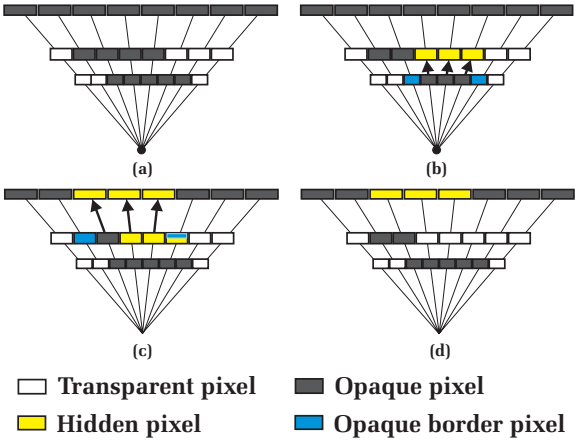

Figure 5: *Removing pixels in the sampling layers that are hidden behind layers closer to the viewpoint.*

with information from the neighboring pixels as long as the gap was not larger than one pixel, which is ensured by the error metric described above. However, surfaces viewed from a sharp angle may not be recorded at all in some layers, because the part of the surface visible in those layers falls between two adjacent pixel centers in the sampling layer. In this case, gap filling does not work because information is missing in more than two consecutive layers, which also creates problems for the mesh generation step further on.

To remedy the situation, in this paper we take a different approach to gap filling and treating surfaces under oblique viewing angles. The basic idea is to ensure that the following condition is always fulfilled when rendering the voxel layers: if a continuous surface extends over two consecutive voxel layers, no discontinuities are allowed between the representations in the two layers, i.e., all the pixels defined by the intersection of a polygon at the far border in the closer layer are also represented in the following layer. Note that if this condition is observed, all voxels occluded by the continuous object surface will be properly deleted in the following layers by the removal algorithm described above.

As described above for the example of viewing oblique surfaces, this condition is not always fulfilled due to the rasterization rules of graphics hardware (see figure 6, left).

Therefore we have to make sure that along the intersection of polygons with the near and far border layer of a voxel, pixels are drawn even if they are only partly covered by a polygon, especially if the pixel center is not inside the polygon. The way to ensure this is to manually clip each polygon to the corresponding near and far border layer of a voxel, and draw the resulting polygon edges explicitly (see figure 6, right). On today's hardware, lines also have to be drawn in a predefined direction (e.g., from left to right) to ensure that identical pixels are rasterized for every edge. The result is

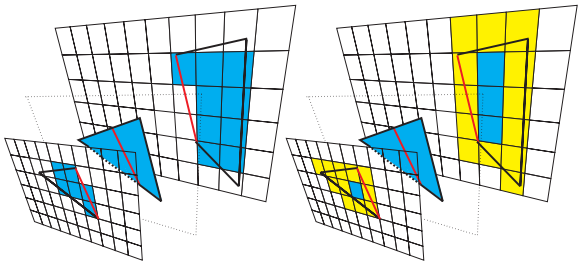

Figure 6: *Left: the polygon edge clipped on the border plane is not represented completely in both adjacent sampling layers. Right: by drawing the outline of the clipped polygon explicitly (light pixels), a complete representation of the clipped edge is generated in both sampling layers.*

that all pixels belonging to edges clipped on the border layers are represented in *both* adjacent voxels.

After using the algorithm described at the begin of this section—including the modification just discussed—to remove invisible pixels (and hence the respective voxels), each layer contains exactly the information that might become visible when the viewer moves within the view cell.

4.3. Uniformly sized voxels

The voxel field generated so far contains layers with non-uniform spacing according to the parallax and visibility error measures described above. For further treatment, however, it is advantageous to transform the voxel field into a normalized, uniform voxel grid (i.e., a grid with cubic voxels and a maximum total sidelength of 1, see figure 7).

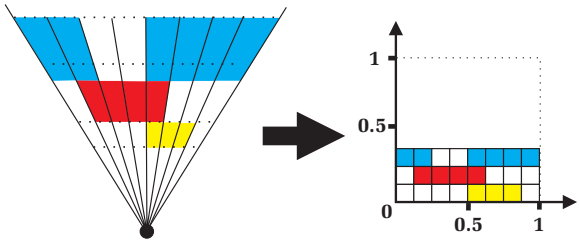

Figure 7: *Transforming the recorded voxel field (left) into a normalized, uniform voxel field (right).*

The transformed voxel centers can actually be determined using only the layer numbers and integer pixel coordinates. First, determine s_{max}, the maximum of the number of voxel layers and the sampling resolutions in x and y. The position p^v of the voxel corresponding to a pixel with 2D coordinates PIX_x and PIX_y in sampling layer l is then calculated as follows:

$$p_x^v = \frac{PIX_x + 0.5}{s_{max}},$$
$$p_y^v = \frac{PIX_y + 0.5}{s_{max}},$$
$$p_z^v = \frac{l}{s_{max}}.$$

This special voxel arrangement influences the error metric used during the mesh-simplification step later on in a way such that projected errors for views within the view cell are accounted for rather than the simple geometric distance between vertices.

5. Mesh generation

The first step in creating the final TDM representation from the voxel field discussed in the previous section is to produce an intermediate mesh that is highly complex, yet of very simple structure. While the intermediate mesh can be quite large, it is very easy to create and is highly amenable to simplification.

The basic approach is to apply a meshing operator after rendering every sampling layer. This operator connects all adjacent voxels in the current layer and closes connections to the previous (less distant) one. Those connections are detected with the help of the consistent representation of clipped edges for continuous surfaces as was described in section 4.2.

For clarity, we start by describing the process for a single voxel layer. An obvious approach would be to simply connect the centers of adjacent voxels as shown in figure 8, left. However, this leaves voxels with only one neighbor unaccounted for. To overcome this problem, four vertices are distributed evenly on the sampling layer of a voxel and used for connecting to adjacent voxels (figure 8, right).

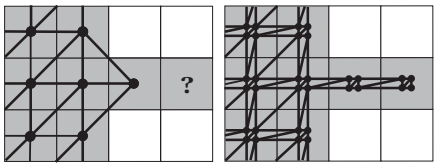

Figure 8: *Covering voxels with only one neighbor is not possible using a single vertex (left). At least two connecting vertices are necessary (right).*

This configuration creates more polygons, but allows covering all connected voxels in a layer with a mesh by applying a series of simple steps for every voxel as described in the following. The illustrations show the previous (closer) and the current (more distant) layer as seen from the reference viewpoint. Gray signifies filled voxels.

In the first phase, connections are made in the current layer if any of the involved voxels in the previous layer is not filled. From left to right and top to bottom in figure 9, the current voxel is first connected to itself, then to the right, to the bottom, and to all 3 neighboring voxels, if the voxels shown in gray are filled in the current layer, but at least one of them is not filled in the previous layer.

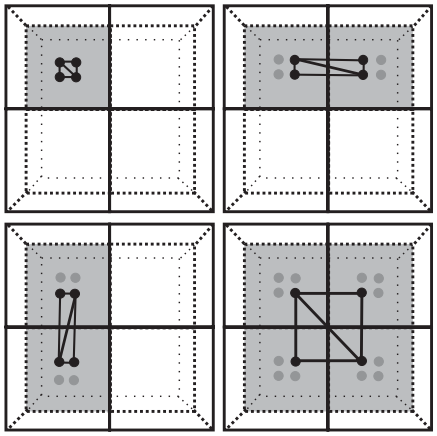

Figure 9: *Connections made in the current layer if any of the voxels in the previous layer above the filled (gray) voxels is not filled.*

In the second phase, the current voxel is "sealed off" towards each of 4 sides if both the voxel itself and the corresponding voxel in the previous layer is filled. In addition, for a particular side, the voxel is *only* sealed off if the voxel in the previous layer towards this side is *not* filled, as shown in figure 10.

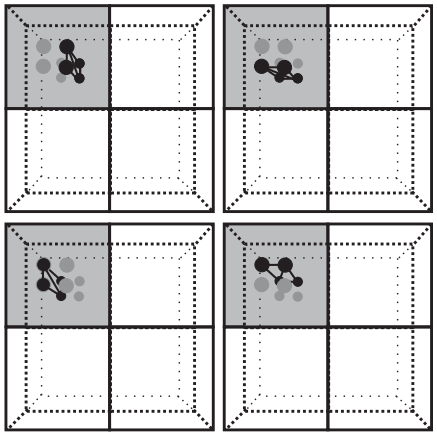

Figure 10: *Sealing off towards each side if a voxel towards a side is not filled in the previous layer.*

The third and final phase connects 2x2 blocks (2 in the current, 2 in the previous layer) of filled voxels if any of the

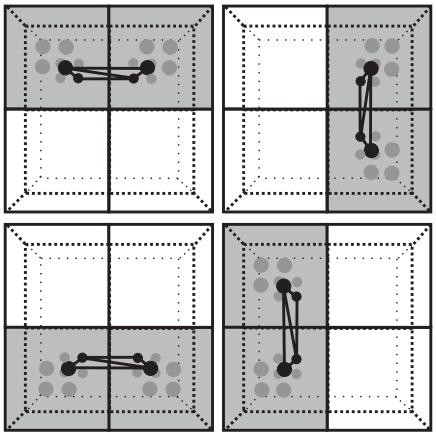

Figure 11: *Connecting 2x2 blocks of voxels.*

2 facing voxels in the previous layer is not filled, as shown in figure 11.

Note that the total number of polygons generated in this algorithm could be reduced by a factor of 2 to 3 by adaptively introducing extra vertices only where needed. However, this would also complicate the mesh-generation process and should be considered only in memory-constrained situations. Also note that the number of polygons generated is output sensitive due to the removal of invisible parts of the scene, and therefore the mesh is not expected to grow beyond a size that can easily be handled in memory.

One of the challenges in the mesh-generation process is recording and maintaining mesh connectivity. In practice, when a new polygon is created through one of the steps described above, a vertex table is used to determine whether the polygon can be directly connected to any of the existing meshes. If only one candidate mesh is found, the polygon is added to it and the algorithm proceeds. If it can be connected to more than one mesh, however, all the candidate meshes are merged using this new polygon as a connection. If the polygon can not be connected to any existing mesh, it is used to start a new mesh containing only this polygon.

The method described in this section does not produce more individual meshes than strictly necessary because of occlusion and disjoint parts of the scene. When rendered, the set of resulting meshes covers all the voxels from the original representation. Note that while at the borders the resulting meshes may be slightly smaller than the set of voxels they cover, this is correct since due to rasterization, the voxels are usually slightly larger than the objects they represent.

6. Mesh simplification

The meshes produced by the algorithm presented in the last section provide a good starting point for the application of a mesh simplification algorithm in order to significantly reduce their complexity. The aim is to produce a set of simple meshes that still cover all the voxels touched by the initial mesh. Although most of the simplification approaches presented in recent years[13] could be applied, the specific structure of the problem suggests the development of a new method.

The approach presented here is conceptually related to simplification envelopes[4] by its use of a surrounding volume bounding the simplification process. Individual simplifications are based on edge contractions (using the QSlim simplification software developed by Michael Garland[7]).

The algorithm proceeds by inserting all edges into a priority queue sorted by edge length, and iteratively removing the shortest edge from the queue. This edge is checked against the voxel grid (see below), and if it can be contracted, all edges affected by the contraction are resorted or reinserted in the queue according to their new length. If contraction fails, the edge will only be reconsidered if it is reinserted into the queue because a neighboring edge is contracted. The process continues removing the shortest edge from the queue until the priority queue is empty.

Always choosing the shortest edge for contraction favors a uniform simplification and prescribes a useful order even for finely tesselated coplanar regions. Using other error metrics for ordering the priority queue would fail in such cases and lead to slither triangles because the simplification order would be more or less random. Also note that an edge connecting two sampling layers has the same length as an edge between two pixel centers in the warped voxel grid the algorithm is operating in (see section 4.3). But this is exactly what we are looking for, because the length in the warped grid corresponds approximately to the perceived length after projection to the screen. In this way, edges that appear to have the same length on screen are treated equally, regardless of their length in world space.

A crucial part of the algorithm is the test whether an edge can be contracted. This test basically checks whether the resulting mesh still covers all relevant voxels and is described in the following.

At first, while creating the initial mesh, each polygon stores identifiers for all voxels it covers (this is trivial to compute due to the mesh structure). Now, assuming a contraction between the vertices v_1 and v_2 should be performed, the necessary steps are:

1. Collect the voxels covered by the polygons that include v_2 by simply accumulating the stored voxels for every involved polygon.
2. Contract v_2 to v_1 and check if the voxel set computed in the previous step is also covered by the resulting polygons (using a polygon-box intersection test). If not all voxels are covered, undo the contraction, and either ex-

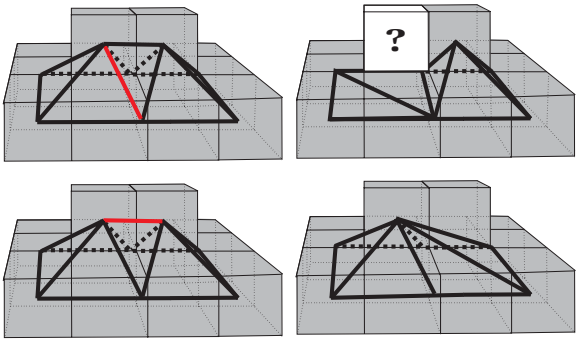

Figure 12: *An example for a forbidden (top) and an allowed (bottom) contraction in a 3D case.*

change v_1 and v_2 and repeat with step 1, or, if v_1 and v_2 were just exchanged, exit (the contraction failed).

3. If all relevant voxels are covered by the new polygons, store for each new polygon the set of voxels that are covered by it (this information has already been computed as a byproduct of step 2).

See figure 12 for a 3D example. This test is adequate for the interior of the mesh, but for boundary edges, special treatment is required. First, contracting a boundary vertex towards the interior of the mesh might be allowed because the resulting mesh still covers all relevant voxels—however, this will create a crack in the representation (see figure 13 for a 2D case). Therefore, contractions involving boundary vertices are only allowed if the contraction is "towards" a boundary vertex.

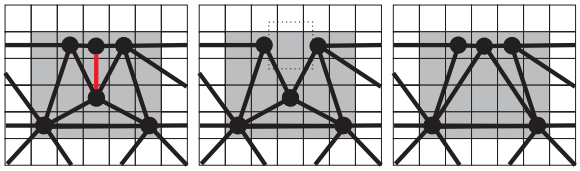

Figure 13: *Left: The initial configuration with the light edge to be contracted. Middle: This contraction away from a boundary vertex is forbidden because it would create a gap, although all (enlarged) voxels are still covered. Right: Contracting towards the boundary vertex leads to a correct mesh.*

Second, contracting a boundary edge may incorrectly cause empty voxels to be covered (see figure 14). This can be prevented by calculating the intersection intervals of the new boundary edge with all relevant voxels (using a fast edge-cube intersection routine) and test whether the union of these intervals completely covers the edge. If an empty edge interval remains, the contraction is rejected. These two constraints prevent errors on the boundaries while still allowing for sufficient simplification.

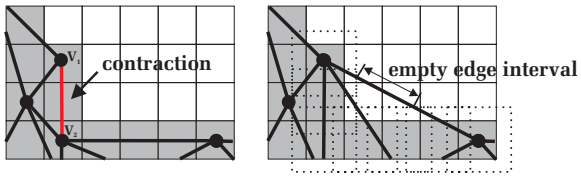

Figure 14: *The shown contraction is not allowed because an empty edge interval would result. The dotted boxes indicate the enlarged voxels for intersection testing with the polygon.*

Finally, it should be noted that the size of the box representing a voxel for testing against a polygon can be used as a simple criterion to trade off rendering speed against accuracy: increasing the size of a voxel results in more simplified, but slightly less accurate meshes. Here again the warped mesh representation calculated in section 4.3 plays an important role, as it provides a one-to-one correspondence of screen-space pixel error to the factor used to increase voxels. A factor of about 1.5 to 2 has proven to result in dramatically simplified (see section 9), but still fairly accurate meshes.

7. Texture generation

The result of the simplification step is a simple geometric representation that covers the voxel field. In the next step, the polygons in this representation need to be assigned color information via texture maps.

This can be done efficiently by sampling the voxel map for each polygon: first, generate a texture so that the texel size approximately corresponds to the side length of a voxel, and the extents of the texture are slightly larger than the polygon in order to ensure that all texels needed for rasterization are present, even when bilinear texture filtering is used. This texture is then sampled in voxel space by searching for every texel center the n closest voxel centers and average the corresponding color values according to distance. For texels lying completely inside the polygon, it is sufficient to consider only the voxels stored with the polygon, while for texels at the border or outside the polygon, the whole voxel field has to be searched. Applying oversampling (i.e., setting $n > 1$) results in smoother, but also somewhat blurrier textures.

After all textures are generated, they can be stored using a *texture atlas* as presented by Cignoni et al.[3], in order to minimize state-changing overhead during rendering and to make more efficient use of texture memory. As already noted by Darsa et al.[5], perspective correction for those textures should be disabled during rasterization, since perspective is already recorded in the texture.

8. Reprojecting the mesh into world space

The final step is to reproject the vertices of the simplified and texture-mapped mesh to their correct positions in world

space. Since no new vertices were introduced during simplification, all vertices still lie within one of the sampling layers. The formulas to reproject a vertex from uniform voxel space v^v to world space v^w are therefore:

$$\begin{aligned} l_z &= layer[v_z^v \cdot s_{max}], \\ v_x^w &= 2 \cdot l_z \cdot \tan(\frac{FOV}{2}) \cdot (\frac{v_x^v \cdot s_{max}}{RES_x} - \frac{1}{2}), \\ v_y^w &= 2 \cdot l_z \cdot \tan(\frac{FOV}{2}) \cdot (\frac{v_y^v \cdot s_{max}}{RES_y} - \frac{1}{2}), \\ v_z^w &= l_z, \end{aligned}$$

where RES_x and RES_y are the resolutions of the sampling layer in x and y, and $layer[i]$ is the distance of layer i from the reference viewpoint in world space.

9. Results and discussion

The algorithms presented in this paper were implemented and tested on a standard PC with an Athlon 1800+ processor, 1 GB of main memory and a GeForce3 Ti 500 graphics card.

To demonstrate the wide variety of possible usages of the approach, two very different models have been chosen for testing: first, a wide urban environment (a model of the Aztec city of Tenochtitlan, which is freely available on the internet), and second, a single, relatively small object (a car), using exactly the same algorithm for both models. The resulting model statistics are shown in table 1.

Model	Tenochtitlan	Car
Polygon count of the model	158944	1188
Size of the model	450x450x50m	4.7x1.8x1.5m
Size of (cubic) view cell	42m	63m
Distance view cell to model	487m	73m
No. of generated meshes	12	2
Polygons in initial meshes	637462	83481
Polygon count of TDMs	946	220
Processing time	1.5 min	5 sec

Table 1: *Model statistics.*

The urban model includes large occluded scene parts, which are automatically recognized and excluded by the algorithm (see figure 16). On the other hand, the method guarantees that all visible scene parts are represented, so that no rubber-sheet effects or image cracks appear. While degenerate meshes can occur under rare circumstances, the quality of the output meshes is usually very high: they follow quite precisely the shape of the objects to be represented, even when relaxing the parameter for mesh simplification, as can be seen for the car model in figure 17 (a factor of 2 was used to enlarge the voxel size for both test scenes).

Due to the special layer arrangement, all scene parts are recorded with the proper resolution, which also helps to avoid temporal aliasing (i.e., flickering). The effect of oversampling the impostor textures can be observed in figure 15, where the result looks smooth, but also a bit blurred.

Although the number of polygons in the intermediate mesh representation may seem high, this is mainly because of the number of pixels on screen and not because of the complexity of the original scene, thanks to visibility processing, and it is unlikely that polygon counts will grow much beyond what can be seen here. Taking into account that the method is very general and can handle arbitrary input meshes, we consider the processing times reasonable, and we are optimistic that several time-consuming stages can still be accelerated significantly.

Note that although the images show only the resulting impostors, typical usage scenarios would integrate TDMs for mid-range to distant objects with standard polygonal rendering for the near field. This allows most of the available polygon budget to be spent on the areas where it is needed most, i.e., on a very detailed representation of special effects, nearby static objects and moving characters, while at the same time, far objects are still rendered with high fidelity and the minimum amount of polygons necessary. Through mesh simplification, view cells can be made much larger than if using layered impostors[8] alone, and fewer view cells are needed. Savings in rendering time stem not only from the significant reduction in polygon count for objects represented by TDMs (see table 1), but also from the simplicity of the representation: no state changes are necessary when rendering TDMs, allowing graphics hardware to be utilized to its fullest potential.

10. Conclusions

This paper introduced a novel approach to generate textured depth meshes in order to accelerate the rendering of very complex, arbitrary geometric models. The contribution of the approach is manifold:

It generates high-quality textured depth meshes of arbitrary scenes, which are practically indistinguishable from the original rendered models when seen from a fixed viewing region. A single parameter is provided for controlling the tradeoff between mesh complexity and accuracy during the simplification algorithm. The sampling layers are optimally placed, so that the parallax error remains bounded. Because of this, no image cracks or rubber-sheet effects due to missing information about hidden parts of the scene appear. Moreover, only potentially visible parts of the scene are represented in the TDM, thus minimizing the storage cost of the impostor representation.

In contrast to the authors' previous approach, which produces disconnected impostor layers, the complexity of the new impostor representation is completely decoupled from the number of layers used to create it, and therefore also allows a reduction of polygon counts and texture memory requirements for scene parts near the viewer, albeit at the cost of longer preprocessing times.

By accelerating the rendering of distant geometry, the method is well suited to allow larger environments to be displayed in real time, and also frees up enough rendering capacity so that nearby objects can be rendered in more detail. It also compares favorably to using LOD rendering for distant geometry, since it allows arbitrary parts of a scene to be simplified, regardless of their extent (scenes may extend to infinity), topology, connectedness, or rendering method, and it automatically handles only visible geometry.

In terms of future research, we plan to integrate the algorithm into a real-time rendering system. Here, the selection of view cells and objects to be represented by TDMs will be a major issue to be tackled. Further, we will try to improve the versatility of our representation by combining the meshes with normal maps to better exploit the multitexturing capabilities of today's hardware, and allow for dynamic lighting effects—to date absent in most impostor representations. Also, multitexturing could be used for enhanced filtering of different positions inside the view cell as demonstrated for point-based impostors[19].

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. 1, 2

2. S. Chen. QuickTime VR – an image-based approach to virtual environment navigation. In *SIGGRAPH 95 Conference Proceedings*, pages 29–38, 1995. 1, 2

3. P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. A general method for preserving attribute values on simplied meshes. In *Proceedings IEEE Visualization'98*, pages 59–66, 1998. 8

4. J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 119–128, 1996. 7

5. 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. 2, 8

6. X. Decoret, F. Sillion, G. Schaufler, and J. Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3):61–73, 1999. 2

7. M. Garland and S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 209–216, 1997. 7

8. S. Jeschke and M. Wimmer. Layered environment-map impostors for arbitrary scenes. In *Graphics Interface 2002*. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 2002. 2, 4, 9

9. Marc Levoy and Pat Hanrahan. Light field rendering. In *SIGGRAPH 96 Conference Proceedings*, pages 31–42, 1996. 2

10. 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. 1, 2

11. W. Mark, L. McMillan, and G. Bishop. Post-rendering 3D warping. In *1997 Symposium on Interactive 3D Graphics*, pages 7–16, 1997. 2

12. N. Max and K. Ohsaki. Rendering trees from precomputed Z-buffer views. In *Rendering Techniques '95*, pages 74–81, 1995. 2

13. E. Puppo and R. Scopigno. Simplification, lod and multiresolution - principles and applications. In *Eurographics'97 Tutorial Notes PS97 TN4*, pages 31–42, 1997. 1, 7

14. G. Schaufler. Per-object image warping with layered impostors. In *Rendering Techniques '98*, pages 145–156, 1998. 1, 2

15. G. Schaufler and W. Stürzlinger. A three-dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–235, 1996. 2

16. J. Shade, S. Gortler, L. He, and R. Szeliski. Layered depth images. In *SIGGRAPH 98 Conference Proceedings*, pages 231–242, 1998. 1, 2

17. 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. 2, 3

18. F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulationfor real-time visualization of urban scenery. *Computer Graphics Forum*, 16(3):207–218, 1997. 1, 2

19. M. Wimmer, P. Wonka, and F. Sillion. Point-based impostors for real-time visualization. In *Rendering Techniques 2001*, pages 163–176, 2001. 1, 2, 10

20. P. Wonka, M. Wimmer, and F. X. Sillion. Instant visibility. *Computer Graphics Forum*, 20(3):411–421, 2001. 1

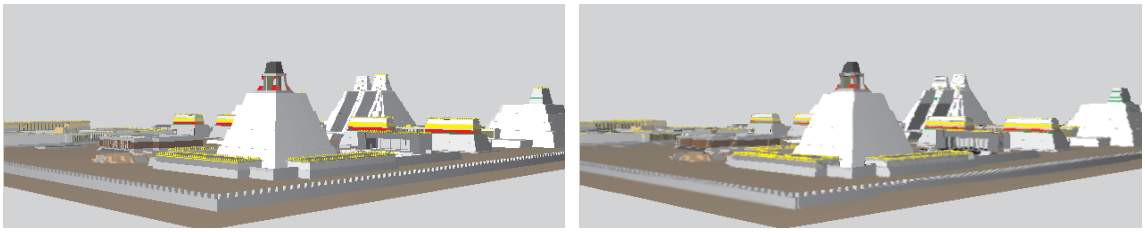

Figure 15: *Left: geometric model consisting of 158944 polygons. On the right, the TDM representation consisting of only 946 polygons in 12 meshes, with the textures 4 times oversampled. Both views are from a viewpoint at the border of the view cell.*

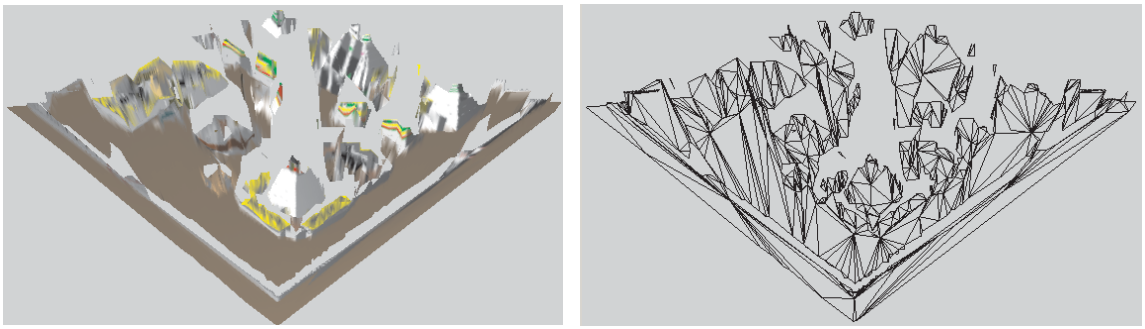

Figure 16: *Left: the same scene from a bird's-eye view. Notice that no occluded scene parts are present in the TDM. Right: the corresponding mesh.*

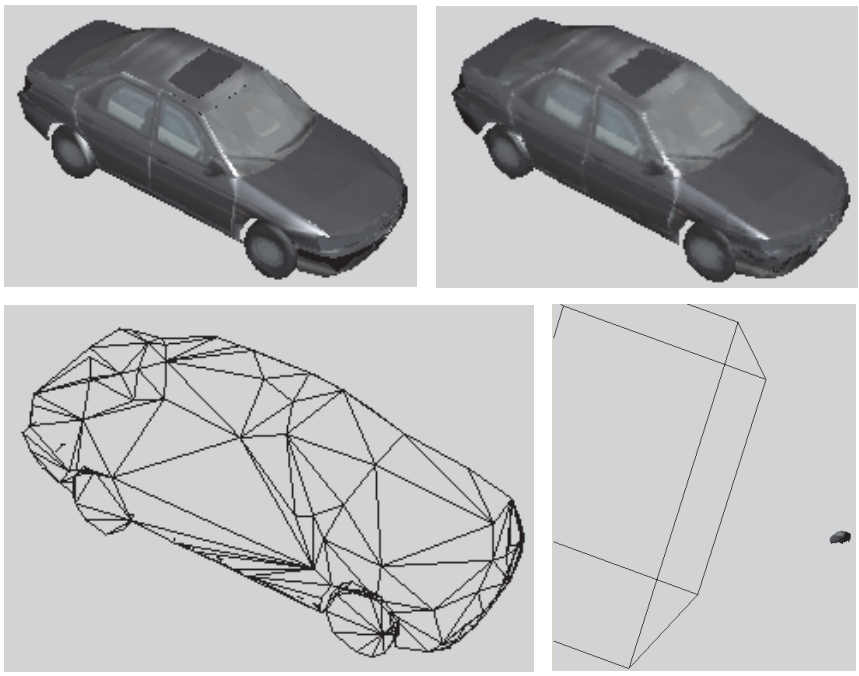

Figure 17: *Top left: close up of the original car rendered using geometry (1188 polygons). Top right: close up of the TDM, containing 220 polygons and non-oversampled textures. The bottom-left figure shows the mesh of the TDM. To the bottom right, the proportion between the car and the associated view cell is shown (both are seen from the side).*