Internal Version (Early Draft)

View-Dependent Rendering of Multiresolution Texture-Atlases

Henrik Buchholz[*]
Hasso-Plattner Institute
University of Potsdam, Germany

Jürgen Döllner[**]
Hasso-Plattner Institute
University of Potsdam, Germany

ABSTRACT

Real-time rendering of massively textured 3D scenes usually involves two major problems: Large numbers of texture switches are a well-known performance bottleneck and the set of simultaneously visible textures is limited by the graphics memory. This paper presents a level-of-detail texturing technique that overcomes both problems. In a preprocessing step, the technique creates a hierarchical data structure for all textures used by scene objects, and it derives texture atlases at different resolutions. At runtime, our texturing technique requires only a small set of these texture atlases, which represent scene textures in an appropriate size depending on the current camera position and screen resolution. Independent of the number and total size of all simultaneously visible textures, the achieved frame rates are similar to that of rendering the scene without any texture switches. Since the approach includes dynamic texture loading, the total size of the textures is only limited by the hard disk capacity. The technique is applicable for any 3D scenes whose scene objects are primarily distributed in a plane, such as in the case of 3D city models or outdoor scenes in computer games. Our approach has been successfully applied to massively textured, large-scale 3D city models.

CR Categories and Subject Descriptors: I.3.6 [Computer Graphics]: Methodology and Techniques – Graphics data structures and data types; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – Color, shading, shadowing, and texture

Additional Keywords: Multiresolution textures, texture level-of-detail, realtime rendering, view-dependent rendering

1 INTRODUCTION

Rendering massively textured scenes in real-time is an important issue for many applications, particularly in the field of 3D geovisualization and computer games. One problem related to texture complexity is the number of texture switches per frame because a texture switch is an expensive operation. A second problem arises if the set of simultaneously visible textures exceeds the capacity of the graphics memory or even of the main memory. For the problem of texture switches an established solution is to combine several textures in texture atlases [24]. This is, however, only sufficient if the number of the texture atlases needed to fit in all scene textures is small enough for fast rendering. Additionally, even in this case, potential problems arise when texture coordinates of single triangles exceed the [0,1] range – there are three methods to handle repeated textures in texture atlases: The first method is to replicate repeated textures multiple times in a texture atlas. The second method is to tessellate the scene fine enough so that the texture coordinates of all triangles are always in the [0,1] range. Both methods fail, however, if the texture repeat counts are too high. For instance, in one of our test models, splitting triangles until their texture coordinates fitted into the [0,1] range increased the geometric complexity by a factor of 10. The third method is to implement clamp, wrap, and mirror addressing with pixel shaders. As Wloka [24] points out, however, this has some drawbacks. Particularly, the necessity of a pixel shader is not desireable because many current computer games are pixel-shader bound.

In this paper, we propose a view-dependent rendering technique based on multiresolution texture-atlases that provides real-time rendering of massively textured scenes. The approach is suitable for all 3D scenes that are *planary distributed*, i.e., scene objects are primarily distributed on a plane. More precisely, for a sufficiently fine subdivision of a properly chosen plane P into areas A_i, for all A_i the set of all triangles whose orthogonal projection onto P intersects A_i is small enough to be rendered quickly with original scene textures. It should be noted that this definition does not restrict our approach to terrains, but includes a broad range of 3D scenes, such as outdoor environments in computer games or 3D city models in geovisualization. For the case of terrain rendering, specialized terrain techniques are probably more useful. The central features of our approach are:

- It can cope with scenes, for which the set of simultaneously visible textures exceeds the main memory capacity.
- It keeps the texture switches permanently low, even for "worst case perspectives", e.g., if the whole scene is visible at once and several parts of the scene are near to the camera.
- It provides explicit runtime-control of the texel-per-pixel ratio, except for possible short delays due to dynamic texture loading. Usually, we use a ratio of 1:1. If desired, the ratio can be reduced to achieve higher frame rates on slower graphics hardware.
- Since the major complexity of our approach is located in the preprocessing part, the runtime overhead for the rendering algorithm is very small. Particularly, our approach does not consume any shader performance.
- It does not make any assumptions about the way the user acts within the virtual environment, i.e., it does not favor or restrict specific navigation techniques.

The basic principle of view-dependent rendering of multiresolution texture-atlases is similar to multiresolution-texturing approaches for terrain rendering, such as the technique described in [8]: In each frame, the texture resolution is chosen in a way that the texel-per-pixel ratio is always near to 1 so that the amount of necessary texture data remains small. To achieve this, we use a hierarchical data structure, called the *texture-atlas tree*, which provides atlas-texture representations of all visible scene textures at appropriate resolutions.

2 RELATED WORK

The problem of texture complexity has been addressed in several approaches. Clipmaps [21] and 3Dlabs' Virtual Textures can cope with textures that exceed the main memory capacity. They re-

[*] buchholz@hpi.uni-potsdam.de
[**] doellner@hpi.uni-potsdam.de
Helmert-Straße 2-3, 14482 Potsdam, Germany

quire, however, specialized hardware support. Cline and Egbert [2] proposed a software approach for large texture handling. At runtime, they determine the appropriate mipmap-level for a group of polygons based on the projected screen size of the polygons and the corresponding area in texture space. The paper concentrates on rendering of a single large texture and demonstrates the effectiveness for the example of a terrain texture. For arbitrary massively textured scenes containing large numbers of singular textures, however, additional problems have to be addressed. For instance, the texture density may be irregularly distributed in the scene and all textures may have different resolutions. Lefebvre et al. [12] proposed a GPU-based approach for large-scale texture management of arbitrary meshes. The novel idea of their approach is to render the texture coordinates of the visible geometry into texture space to determine the necessary texture tiles for each frame. The technique is applicable for arbitrary scenes. The rendering part of the approach, however, requires a cost-intensive fragment shader for correct filtering and the geometry has to be rendered multiple times per frame.

Another way to manage the texture memory problem is texture compression (e.g., S3). However, the compression ratio is not high enough to cope with massively textured scenes. The adaptive texture maps [10] can cope with complex texture data by a sophisticated texture representation that allows for textures with locally refined areas and with non-rectangular borders. While their approach is effective for higher-dimensional textures, it is mostly not applicable for 2D textures because it is based on the fact that the multidimensional textures usually need to be detailed only in certain regions. Carr and Hart [1] introduced a texture atlas for real-time procedural solid texturing. They partition the mesh surface into a hierarchy of surface regions that correspond to rectangular sections of the texture atlas. This structure supports mipmapping of the texture atlas because textures of triangles are merged only for contiguous regions on the mesh surface. The combination of texture atlases and mipmapping is similar to our approach but since the approach focuses on procedural solid texturing of singular manifold meshes, it is not applicable to our problem of arbitrary complex textured scenes. In a number of applications, procedural texturing [13] can be used to render high-detailed textured scenes without explicitly storing the texture data.

Several approaches for handling texture complexity have been developed in the scope of geovisualization. Shi et al. [19] described a multiresolution-texturing approach for terrain and 3D city models considering texture data for terrain and for building facades. They divide the terrain into a regular grid and store textures of different resolutions in each grid cell. They assume relatively small facade textures and, hence, concentrate on terrain textures. Döllner et al. [3] described a multiresolution technique for terrain textures. It permits the combination of multiple large-scale textures of different size. The approach is, however, specialized for terrain textures and is not applicable for arbitrary virtual environments. Wahl et al. [22] also presented an approach for rendering complex terrains with large-scale textures. They combine geometric simplification, texture level-of-detail, and texture compression with occlusion culling and impostors to make the approach more output sensitive. Hwa et al. [8] proposed adaptive 4-8 hierarchies for terrain texture level-of-detail. By using a 4-8 refinement of raster tiles instead of the commonly used quadtree subdivision, they doubled the number of available texture level-of-details, leading to higher frame-to-frame coherence and improved texture quality. Lakhia [11] described an approach for interactive rendering of detailed city models based on Hierarchical Levels of Detail (HLODs) [4]. To support texturing, they store down-sampled versions of the original scene textures with each HLOD. Since the algorithm concentrates on geometric simplification, texture resolution is not explicitly considered as a factor for the scene subdivision and is not regarded in the cost heuristic. Frueh et al. [5] described an approach to create texture maps for 3D city models. The technique includes the creation of a specialized texture atlas for building facades and supports efficient rendering for virtual fly-throughs. The created atlas is static, and different texture resolutions are not considered. Hesina et al. [6] described a texture caching approach for complex textured city models. Their approach is restricted to interactive walk-throughs.

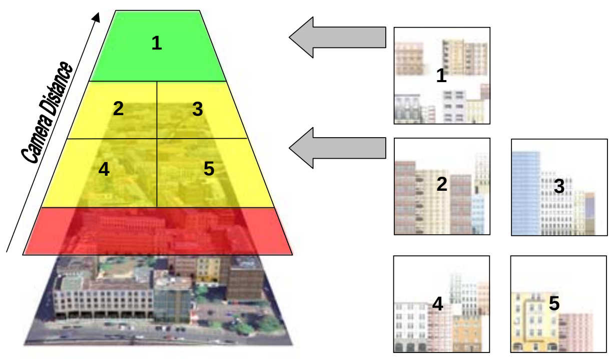


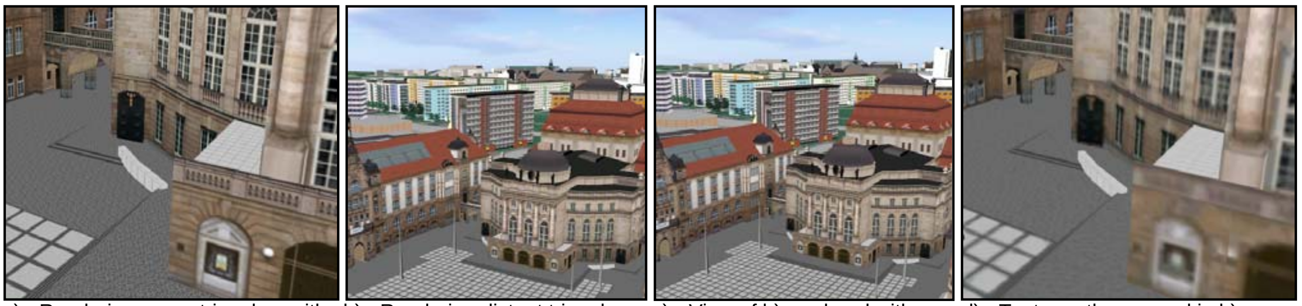
Figure 1. Distance-dependent texture selection. Original textures for the near area, 4 texture atlases for the middle area, and a single combined texture atlas for the far area.

There are also some works addressing support of textures in existing geometric multiresolution techniques. Most of them do not consider the texture complexity itself as a problem, but rather the texture error caused by geometric simplification [18]. Sander et al. [17] presented an approach to support textures in progressive meshes [7]. The original mesh is subdivided into charts, and their texture information is sampled into a texture atlas, minimizing texture stretch and texture deviation. Williams et al. [23] extended the perceptual simplification framework of Luebke and Hallen [15] by considering textures and dynamic lighting effects in the simplification algorithm. Sormann et al. [20] described an approach to integrate texture support in the view-dependent simplification framework (VDS) [14]. They support texturing discontinuities and combine multiple textures hierarchically into lower-resolution textures. So, their approach reduces texture switches and texture memory requirement. For scenes with thousands of textures, however, the time spent in their runtime-texture selection algorithm becomes a significant factor of the rendering performance.

3 TEXTURE-ATLAS TREE

The input data needed to create a texture-atlas tree is a 3D scene description consisting of a set of triangles with texture coordinates and a set of related textures. As stated in section 1, we assume the scene to be planary distributed. Without loss of generality, we will further assume the related distribution plane *P* to be the *x*-*y*-plane.

Figure 1 illustrates the basic principle of the texture-atlas tree. For each frame, the tree provides a small collection of texture atlases containing each visible texture of the scene at an appropriate resolution. For a given camera specification we consider the resolution of a texture as appropriate if it is minimized in a way that magnification is guaranteed to be avoided. The computation of the necessary texture resolution is explained in more detail in Section 5.2.1. Only for triangles close to the viewer original textures are used (Figure 2a). Starting from a certain camera distance, texture atlases are applied, each of them replacing all original textures within the related scene part (Figure 2b). As comparison with Figure 2c shows, the resolution of the texture atlases in Figure 2b is sufficient, although it is considerably smaller than the original texture resolution (Figure 2d). With increasing camera



a) Rendering near triangles with original textures. b) Rendering distant triangles with texture atlases. c) View of b) rendered with original textures. d) Texture atlases used in b) are inappropriate for close views.

Figure 2. View-dependent texture selection.

distance, the necessary resolution for each texture decreases further, so that more original textures covering larger scene parts can be combined in a single texture atlas.

The basic structure of the texture-atlas tree is illustrated in Figure 3. A texture-atlas tree defines a quadtree subdivision of the scene in the *x*-*y*-plane (Figure 3 left) and a corresponding hierarchy of texture atlases (Figure 3 right). Each node N represents a part of the scene geometry and stores its bounding box N.bounds. The quadtree subdivision of the scene may be irregular, i.e., the scene parts represented by 4 children of a node may have different sizes. The scene subdivision algorithm is explained in Section 5.3. In addition to N.bounds, each node stores a single texture atlas N.atlas for all triangles of the related scene part. All texture atlases have equal size. The atlas of an inner node contains downsampled versions of its child nodes. Furthermore, each node stores a distance value N.minDistance, which represents the minimum distance between camera and the node's bounding box to ensure that the node's texture atlas resolution is sufficiently high. The scene geometry is stored in the leaf nodes. Each leaf node contains the triangles of its corresponding scene part and the related subset of the original textures of the input scene. Two sets of texture coordinates are specified for the triangles, one referring to the original textures, and the other referring to the texture atlas of the leaf node. Finally, each node stores an additional texture matrix, which allows for using the same texture coordinate set for all atlas hierarchy levels.

4 Rendering

The rendering is performed by a top-down traversal of the texture-atlas tree. For the moment, we assume all nodes to be in memory. The necessary changes for dynamic loading and deletion are discussed in Section 6. For each traversed node N, a simple hierarchical view frustum test is performed. If the bounding box N.bounds is completely outside the view frustum, the node is skipped. If N is visible, we compute the distance *d* between the camera and N.bounds. If *d* is greater than N.minDistance, the triangles of all leaf nodes of the subtree with root N are rendered using the atlas texture N.atlas and the texture matrix N.textureMatrix. In the case $d <$ N.minDistance, the resolution of the texture atlas of N is not sufficient. So, if N is an inner node, the traversal is continued with the child nodes of N. If N is a leaf, the triangles of N are rendered using the original scene textures.

For the preprocessing phase we will assume a previously known screen size and a fixed target texel-per-screen-pixel ratio of 1. However, if the screen size is rescaled at runtime by a certain factor, we only need to rescale the minimum distance values of the nodes by the same factor. Accordingly, the texel-per-screen-pixel ratio can be increased to consider anti-aliasing or reduced to run the algorithm on slower graphics cards.

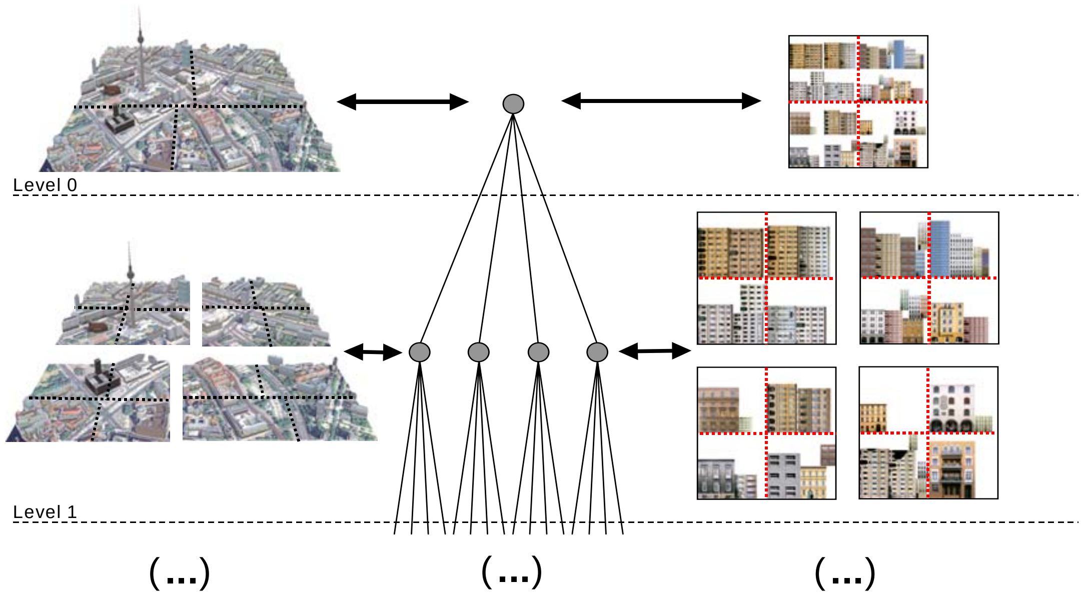


Figure 3. Structure of the texture-atlas tree. Each node represents a certain scene part and provides a texture atlas for this part.

5 PREPROCESSING

In this section we describe how to create the texture-atlas tree. We give an overview of the preprocessing, followed by a description of the most important steps.

5.1 Overview

The preprocessing starts with a tree consisting of a single root node containing all triangles of the scene. For each triangle, we keep a reference to its original texture. First, we try to create a single texture atlas of fixed size for all textures that are used by triangles of the node. Section 5.2 describes the creation of the texture atlas. If the atlas creation fails, the node is split, i.e., four child nodes are created and the triangles of the node are shared among its child nodes. Section 5.3 describes the subdivision algorithm for the triangle set of a node. After a node split, the procedure described above is recursively repeated for the new child nodes. If the atlas creation for a node is successful, the node becomes a leaf of the final tree. In this case, the node stores its related triangles, the corresponding original textures, the computed atlas texture, and the texture coordinates for original textures as well as for the atlas texture. The minimum distance value N.minDistance of each leaf is set to the global minimum distance value d_{min} that have been used for the atlas creation (see Section 5.2).

At this point, the tree structure has been created, and all leaf nodes have been filled with data. Next, we compute texture atlases for the inner nodes in a bottom-up fashion. To obtain the texture atlas of an inner node, the four texture atlases of the child nodes are combined and down-sampled by a factor of 2. The minimum distance values of the inner nodes are computed together with the texture atlases. Each down-sampling step reduces the resolution of an atlas by a factor of two in both axes. Therefore, an appropriate minimum distance value for an inner node is the maximum of all minimum distance values of its child nodes multiplied with 2. Note that the minimum distance values of the child nodes are not necessarily equal because the quadtree may be incomplete.

The final step is the computation of a texture matrix for each node and an appropriate modification of the atlas texture-coordinates of the triangles stored in the leaf nodes. Without this step, the atlas texture coordinates stored of the triangles were valid only for the texture atlases of the leaf nodes but not for the combined atlases of inner nodes. The aim is to use a single set of texture coordinates for all atlas textures, independent of their hierarchy level in the tree. As illustrated in Figure 4, each node atlas corresponds to a squared area of the root node atlas. Considering the path from the root to a leaf, we convert the atlas texture-coordinates of each leaf node to the corresponding texture coordinates in the root atlas. Thus, we set the identity matrix as the texture matrix for the root node. Based on atlas texture coordinates of a node, the atlas texture coordinates for its child nodes can be obtained by multiplying by two and discarding the integer part. Therefore, we can use the root atlas texture coordinates for a node N at level k by setting a scaling matrix with factor 2^k as texture matrix and rendering the atlas texture of N in repeat mode.

5.2 Node Atlas Creation

This subsection describes the creation of a node atlas for a given set of textured triangles. The atlas is created in a way that for a given global parameter d_{min} it is guaranteed, that no magnification occurs for any part of the texture atlas, as long as the camera has a distance of at least d_{min} from all triangles. The global parameter d_{min} should be set small enough, so that from each point of the scene all triangles within a distance of d_{min} can be rendered quickly with original textures. Smaller d_{min} values increase the size of the tree, finally resulting in higher preprocessing time and

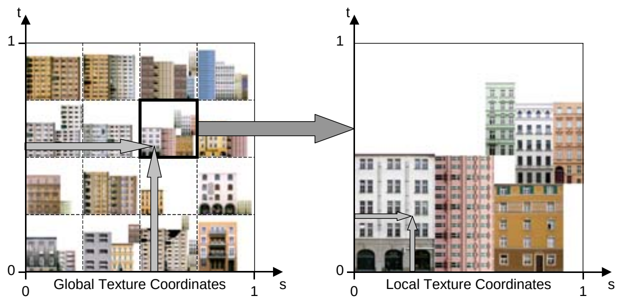


Figure 4. The local texture coordinates for the atlas at tree level k = 2 are obtained by scaling the root atlas coordinates by a factor of 4 = 2^k and discarding the integer part.

a larger paging file. The atlas is created in a fixed predefined size, e.g., 1024 × 1024 texels. The atlas creation is performed in 3 steps:

1. For each texture, we calculate minimum width and height the texture must have to avoid magnification at a distance of d_{min}.
2. Based on step 1, we compute for each texture the size of the required area in the atlas texture and the corresponding source area in texture space.
3. If possible, we create a single texture atlas of fixed predefined size for all textures based on the results of step 2.

5.2.1 Texture Size Estimation

Given a texture and a set of triangles using the texture, we have to compute the minimum required width and height to avoid magnification at a camera distance of d_{min}. For this, we compute these values for each triangle separately and take the maximum of all calculated minimum widths respectively all calculated minimum heights as the final required texture size.

For a single triangle and a camera distance of d_{min} or greater, we compute the largest possible screen-space projections s_{screen} and t_{screen} of the unit vectors (1, 0) and (0, 1) in texture space (Figure 5). If s_{screen} has a length of m pixels, the texture must have a width of at least m texels. The same applies for t_{screen} and the height of the texture. So, we need an upper limit for the lengths of s_{screen} and t_{screen}.

Let $T = (v_0, v_1, v_2, t_0, t_1, t_2)$ a triangle defined by vertex positions and texture coordinates. We can assume (v_0, v_1, v_2) to be non-degenerate because world-space degenerate triangles can be ignored. In the following, we restrict the description to the calculation of the minimum width. The upper bound for the length of s_{screen} is obtained in two steps:

1. Computing the length of s_{world}, the projection of the unit vector (1, 0) to world-space coordinates.
2. Computing of an upper limit for the scale factor involved by a perspective projection from world-space to screen space.

If (t_0, t_1, t_2) is degenerate, s_{world} is not properly defined. For the case $t_0 = t_1 = t_2$ we skip the triangle and set a fixed minimum size of 1×1. If (t_0, t_1, t_2) is degenerate with a non-degenerate longest edge $e = (t_i, t_j)$, we skip step 1) and estimate the length of s_{world} by $\text{abs}(v_i - v_j) / \text{abs}(t_i - t_j)$. So, we can assume (t_0, t_1, t_2) to be non-degenerate as well. Without loss of generality, we also assume $v_0 = 0$ and $t_0 = 0$. Let P_T be the parameterization for T with

$$P(a,b) := a \cdot v_1 + b \cdot v_2, \quad \text{and}$$

$$Tex_T(a,b) := a \cdot t_1 + b \cdot t_2.$$

Tex_T maps the parameterized representation of a vector $v = a{\cdot}v_1 + b{\cdot}v_2$ on the triangle plane to the corresponding texture coordinates. Tex_T is linear and invertible, since (t_0, t_1, t_2) is non-

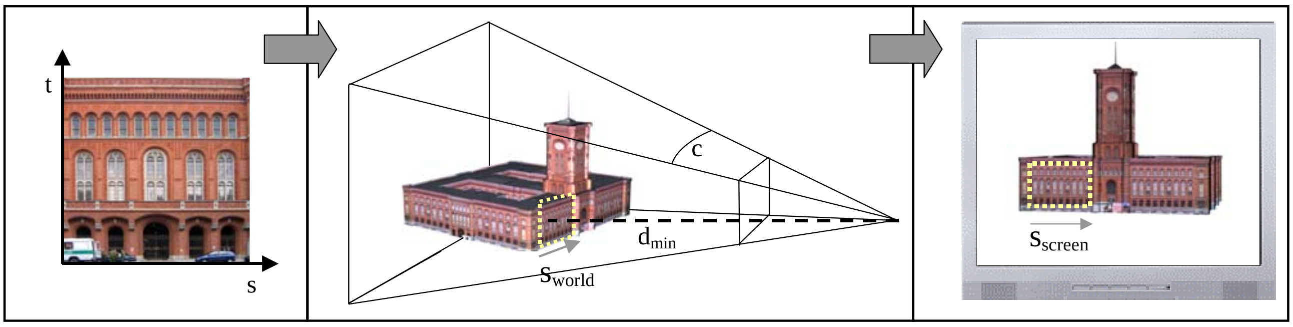


Figure 5. Computation of the largest possible screen projections of the unit vector (1,0) in texture space. The length of s_{screen} is chosen as required texture width.

degenerate. So, the result of step 1) can be calculated using the formula

$$s_{world} = P_T(Tex_T^{-1}((1, 0)))$$

Given a minimum camera distance *d*, a pre-defined field-of-view angle of *c*, and a predefined screen width w_{screen}, we now have to compute the scale factor for step 2). For the sake of simplicity, we assume identical scale factors for both screen axes. The screen projection of a line segment of the length s_{world} viewed from the distance *d* has maximum size if it is oriented orthogonal to the viewing direction. So, we obtain a projection factor

$$f_{proj} = w_{screen} / (2\ d \tan(c/2)).$$

Note that the constant f_{proj} is equal for all triangles and all textures. Finally, we obtain the required texture size by:

$$w_{tex} = \text{abs}(s_{world}) \cdot f_{proj} \qquad \text{and}$$

$$h_{tex} = \text{abs}(t_{world}) \cdot f_{proj.}$$

5.2.2 Computation of the Required Atlas Area

In the last section we obtained for each texture a minimum width and a minimum height to avoid magnification. These values apply for a single texture repetition. To find the required area for a texture in the texture atlas, we have to take into account the area in texture space defined by all texture coordinates for the texture and an additional border to avoid texture pollution (Figure 6). First, we compute the bounding area A_{tex} of all texture coordinates for a texture. Textures that are only partially used in a certain atlas node, are only partially needed in the texture atlas. Repeated textures have to be repeated in the texture atlas. Note that high repetition numbers of textures do not cause problems because the finally required atlas area size depends only on the triangle area in world space. Considering A_{tex} we multiply w_{tex} and h_{tex} with the width and height of A_{tex}.

Finally, we add a texture border area to the required texture area to avoid mixing adjacent textures in downsampled versions of the texture atlas. For our implementation, we use a constant border width of 8 texels. Although this guarantees the avoidance of texture pollution only for the first 3 down-sampling levels, the visual artifacts were hardly noticeable. To fill the border with content of the original texture, we scale the texture space area A_{tex} around its center by a corresponding scale factor.

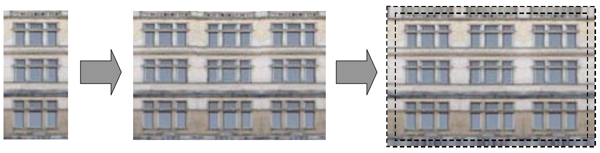


Figure 6. Computation of the required atlas area: a) Area needed for a single repetition; b) Size with considered bounding area in texture space; c) Size with added borders.

5.2.3 Texture Atlas Generation

As a result of step 2, we obtain for each texture the required width and height inside the atlas texture and the source area in texture space from which the original texture content has to be copied. Using the atlas-packing algorithm described in [9], we try to find positions for all textures inside an atlas texture of fixed predefined size. If the texture areas do not fit completely in the atlas image, the atlas creation fails. In this case, the atlas creation is retried after a node split for a smaller subset of the triangles, as described in Section 5.1.

If all textures fit into the atlas, we create the texture atlas image. To accelerate the preprocessing, we use the GPU for the atlas creation by rendering textured quads into the P-buffer. Alternatively, more sophisticated filtering algorithms could be used. For each original texture, we render a textured quad filling the corresponding atlas area, using the corners of A_{tex} as texture coordinates.

5.3 Triangle Set Subdivision of a Node

If the atlas creation for a node fails, the triangles of the node have to be subdivided into 4 groups, one for each child node. A well-chosen subdivision should met two conditions:

a) The overlap of the group's bounding boxes should be small.

b) The summed number of texels for all required texture areas (see 5.2) of a group should be approximately equal for all groups.

To simplify the problem, we assume that for each texture each point in texture space is mapped to at most one point in world space. That is, if for a single texture the same area is mapped to two different triangles, this is not considered for the subdivision. Since the rendering algorithm can also handle scenes with completely individually textured triangles, multiply referenced texture areas do not need to be exploited for optimization. Given the above-formulated assumption, the summed number of texels for all required texture areas is approximately proportional to the summed areas of all triangles in world space. So, the condition b) can be substituted by the condition

b') The summed area of all triangles of a group should be approximately equal for all groups.

To find a proper subdivision according to both criterions, we use a simple greedy algorithm: First, the complete set of triangles is sorted according to increasing *x*-coordinates of triangle midpoints. As stated in Section 3, we assume that the scene can be properly subdivided in the *x*-*y*-plane. Next, the ordered sequence is split into two groups. The first one contains the first *k* triangles of the sequence. *k* is chosen so that the summed area of the first *k* triangles is as close as possible to the $A / 2$, where A is the summed area of all triangles. Finally, we split both groups in the same way along the y-axis. Assuming single triangles to be sufficiently small compared to the node size, the resulting 4 groups have approximately equal summed areas and the overlap between their bounding boxes is very small.

6 MEMORY MANAGEMENT

This section describes the memory management strategy to use the texturing algorithm for scenes whose complete set of textures exceeds the main memory capacity. Dynamic loading and deleting of textures is applied to texture atlases and to original textures of leaf nodes. All other data of the tree is kept in memory.

As a first step, the rendering procedure has to be slightly changed to cope with missing textures. When a texture atlas is not sufficient for a node N and one or more child nodes are not in memory, all triangles of the missing child nodes are nevertheless rendered immediately using the atlas texture of N. The root atlas node is always in memory. Thus, in the worst case some triangles are rendered with too coarse textures but for all triangles there are at least approximate textures available.

The management strategy handles atlas textures and original textures simultaneously in the form of memory blocks of approximately equal size. Each block represents either an atlas texture or a set of original textures of a leaf node. Large sets of original textures are split into multiple blocks. The memory management strategy is based on the following considerations:

- The number of blocks in memory must not exceed a predefined maximum capacity k.
- Blocks needed for the current frame must be preferred.
- If the atlas of a node N is in memory, the atlases for all nodes on the path to N should also be in memory. Similarly, if the original textures of a leaf node are in memory, the atlas of N should be in memory as well.
- If multiple blocks have to be loaded, the blocks for the nearest nodes should be loaded first. This is particularly important for slow loading speeds such as in the case of web-streaming.
- If the number of simultaneously needed blocks exceeds k, the memory management strategy loads and keeps only the k blocks for the nearest nodes.
- Textures of nodes near the camera that are currently not used due to view-frustum culling should be scheduled for prefetching because they can become visible very quickly, even with slow camera movement.
  - If the atlas texture of a node N is currently used, the atlas textures of the child nodes of N respectively the original textures of N if N is a leaf node should be scheduled for prefetching.

For a given camera position and view direction, we define the priority of a block based on the criteria usage, distance and hierarchy level. The usage categories ordered by decreasing priority are needed, culled, parent needed, and not needed. Needed textures are all textures that would have been used for rendering if the whole tree were in memory. Culled textures are those, that have appropriate texture accuracy, but are currently not needed due to view-frustum culling. For the atlas texture of a node, the usage value parent needed indicates that the atlas texture of the parent node is currently in the needed category. For blocks representing original textures of a leaf node N, the usage value parent needed indicates that the atlas texture of N is currently needed. The usage of each node is determined in a separate tree traversal. In contrast to the modified rendering traversal, the additional traversal does not stop at missing child nodes, so that needed nodes are also recognized if their parent nodes are currently not in memory. Among blocks of equal usage value, the priority is determined by the distance between the bounding box of the corresponding node and the current camera position. If usage and distance are equal for two blocks, the block with the lower tree level is preferred. In this way it is ensured, that the parent node of a needed node has always higher priority.

Based on the above defined priorities we create a set of blocks that should currently be in memory, called the *ideal block set*. The ideal block set is updated for each frame and contains all blocks that have been identified for the categories needed, culled, and parent needed. If the ideal block set exceeds the maximum number of blocks k, it is reduced to the k blocks of the highest priority. All blocks in memory that are not contained in the ideal block set, are moved to the unused category.

To keep the frame rate constant, we maintain a priority queue of currently requested blocks. In each frame, a fixed time slot is reserved to continue the loading process of the first blocks in the queue. A block is removed from the queue if it has been completely loaded or if it is not in the ideal node set anymore. As long as the queue does not exceed a predefined maximum length, new blocks can be added to the queue. To determine the next block to add, we traverse the ideal node set in the order of decreasing priority. The most important node that is currently neither in memory nor queued is added to the queue. The overhead for the management of the ideal node set is not critical because its size is usually at a scale of 100 or lower.

7 RESULTS

In our first test case, we used the Berlin city model, consisting of 4,300 block buildings (Figure 7). For each building, an individual texture of 512×512 texels resolution was repeated around the facade. Since we only had 300 different facade photographs available, we created 4,300 individual textures by labeling each texture with an identifier. The aerial image used for terrain and building roofs was rendered using [3]. The performance tests, however, were made only for the building facades rendered by our approach. The summed uncompressed size of all facade textures was about 3.15 GB. All tests were performed at a screen resolution of 800×600 with a texel-per-pixel ratio of 1 on a notebook with Pentium-M processor at 1.5 GHz, 1 GB main memory, and an ATI Radeon 9600 Mobility graphics card with 128 MB graphics memory. The preprocessing time for the atlas tree creation was approximately 80 minutes. Since the scene did not completely fit into main memory, a considerable part of the preprocessing time was spent in loading image files. The total file size of the atlas tree was 2.28 GB. The tree depth was 6. Figure 8a shows the measured frame rates for the façade rendering. In all tests, we used view-frustum culling and chose a camera path with rapidly changing visibility, ranging from a single building to a complete overview. We achieved an average frame rate of 69 fps. In the worst case, when the whole scene became visible, the frame rate was about 12 fps.

In the second test case, we used a detailed city model of Chemnitz, Germany, to check the ability of our approach to cope with more irregularly textured scenes. The geometric complexity of 300,000 triangles is not critical for current graphics cards. The model contains, however, about 2,000 different textures, which are all visible at once from an overview perspective. Although the summed size of all textures is only about 36 million texels, the textures cannot simply be combined into usual texture atlases because several textures are massively repeated even on single triangles, such as the tile textures in Figure 2a. Considering repetitions on single triangles, the summed texture size of the scene expanded to multiple billion texels. Since the scene contains transparent parts, we used two atlas trees, one for RGB and one for RGBA textures. The two trees required an overall preprocessing of 8 minutes and 223 MB disk space.

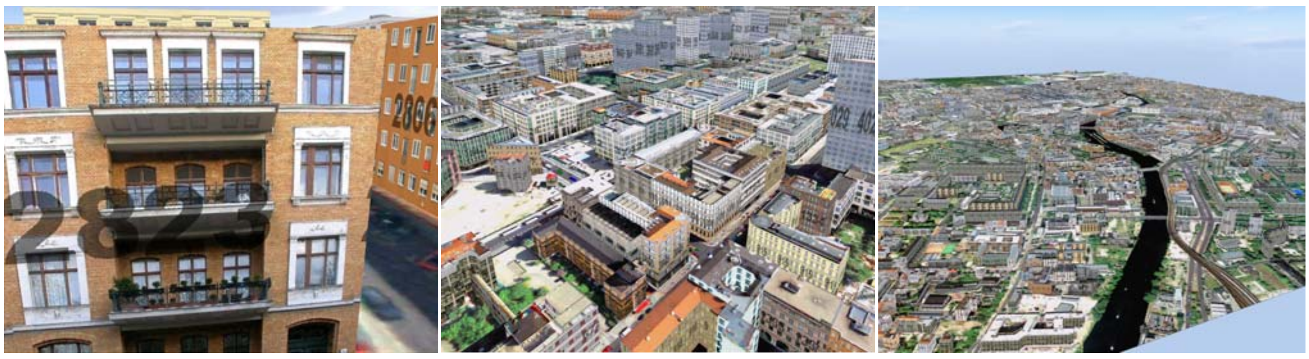
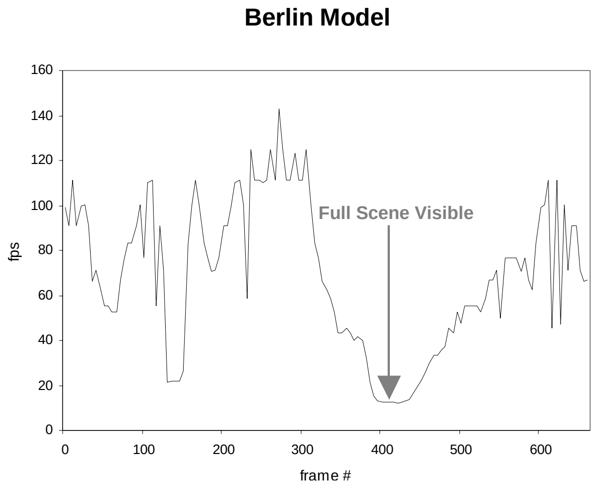
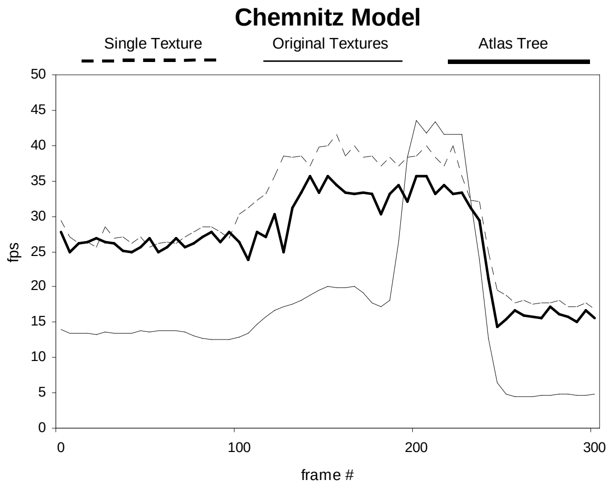

a) Close view to a 512 x 512 façade texture. b) Flythrough perspective: 23 atlas nodes. c) Overview: 13 atlas nodes.

Figure 7. Snapshots of the city model of Berlin used in the first test. Original textures were only needed in a).

Figure 8b shows the time measurements for our test camera animation. The thin solid line indicates the frame rate achieved for rendering the scene with original textures and view-frustum culling. The dashed line shows the frame rates we achieved with rendering the scene without texture switches, i.e., using a single dummy texture for the whole scene. We used the dummy texture frame rates as an approximate indicator for the maximum performance that can be achieved by optimizing texturing performance. The thick line indicates the frame rate achieved using the two atlas trees simultaneously. For all three measurements, we rendered the complete scene without alpha tests and alpha blending because the alpha test caused a dependency between texture content and frame rate, which is misleading for comparison between the atlas tree frame rate and the frame rate with the dummy texture. The average frame rates were 17 fps for the original textures, 30 fps without texture switches and 27 fps for the atlas tree. At the overview perspective the original texture rendering dropped below 5 fps, while the atlas tree was permanently over 14 fps. Only at extremely low visibility, the original scene rendering was fastest because the necessity to separate more objects supported the view-frustum culling. For the final rendering with correct blending of alpha textures we achieved average frame rates of 21 fps for the atlas tree and 13 fps for the original textures.

For all tests we used an atlas resolution of 512×512 texels. We chose $d_{min} = D/151$ for the Berlin model and $d_{min} = D/62$ for the Chemnitz model, where D is the diagonal of the bounding box of the scene, respectively. Since we did not use occlusion culling, the number of nodes used for rendering was at a maximum for small camera heights. Even so, the number of nodes used in the Berlin model was mostly below 20 and did never exceed 40. Due to the more heterogeneously distributed texture density and the additional RGBA tree, the Chemnitz model required slightly more nodes in dense regions. We observed a usage of about 30 nodes on average and about 60 in the worst case. For both models, we used S3 texture compression to reduce the disk space requirements and to speed-up the dynamic loading. To ensure the activity of the dynamic loading process during the tests, we limited the numbers of blocks in memory to 160. Hence, for the Berlin model, the memory requirement of the application including textured roofs and terrain did never exceed 350 MB. Although the images obtained from rendering with texture atlases were not exactly identical to those achieved using original textures, the visual difference was very small due to the explicit texel-per-pixel control (Figure 2). We only got some visual artifacts caused by the S3 compression. The dynamic loading of textures was hardly noticeable for normal navigation speeds. Only if we zoomed rapidly into a completely unloaded scene part within fractions of seconds, the algorithm needed 1-3 seconds to refine the textures appropriately.





a) Frame rates measurement for 3.15 GB facade textures. b) Frame rate comparison between rendering with original textures, rendering without texture switches, and rendering with the atlas tree.

Figure 8. Performance-measurements for camera paths with strongly varying visibility.

8 Conclusions and Future Work

The atlas tree has proved to be an effective technique for interactive rendering of massively textured scenes. It can be used for rendering scenes that exceed the main memory capacity, to receive textures via web-streaming, or just to reduce the number of texture switches if the textures are too large to be combined in usual texture atlases. The application of the atlas tree is not restricted to scenes in which textures are distributed homogenously. Theoretically, the restriction to planary distributed scenes can be relaxed as well by a small change: The only part of the algorithm where the quadtree subdivision is explicitly addressed is the subdivision in Section 5.3. Instead of subdividing according to x and y axis, we could choose the two axes, for which the bounding box of the triangle set to be subdivided has the largest extend. The application for completely arbitrary scenes, however, has to be properly evaluated first.

Future improvements could consider the size of the atlas tree file. 1) The atlas packing scheme could be optimized to increase the atlas coverage [1]. 2) Repeated textures have currently to be repeated in the texture atlas. In scenes such as the Chemnitz city model, the file size could be dramatically reduced by an explicit support for repeated textures. 3) Shared textures have to be stored once for each leaf node in which they are used. However, an explicit support for these triangles is probably only rarely helpful because triangles sharing a texture might be arbitrarily distributed. In addition, triangles of several leaf nodes could be batched together. Although a single call for rendering a batch of geometry is lightweight compared to a texture switch, the number of geometry batches should be reduced as well.

To achieve full scalability in our approach, two points have to be considered: 1) The dynamic loading has to be extended from just textures to whole parts of the tree. 2) For buildings at extreme distances it is not useful to render the whole geometry and to combine the textures of several buildings in single texels. Those buildings can better be replaced for instance by a terrain texture.

We are also investigating the combination of our approach with techniques for reduction of geometric workload and fill rate, either by geometric simplification or by impostor techniques. In addition, we are currently extending our approach by the capability of interactive manipulation and replacement of scene textures.

9 Acknowledgements

We would like to thank 3D Geo GmbH (www.3dgeo.de) for providing the implementation platform LandXplorer, an authoring and presentation system for virtual 3D city models and landscape models as well as Remote Sensing GmbH for Berlin city model, and IMK automotive GmbH for the Chemnitz city model. We are also grateful to the German environmental foundation Deutsche Bundesstiftung Umwelt for supporting the Lenne3D project (www.lenne3d.de). Finally, we would like to thank Konstantin Baumann for helpful comments and implementation support.

References

[1] Nathan A. Carr, John C. Hart. Meshed Atlases for Real-Time Procedural Solid Texturing. *ACM Transactions on Graphics*, 21(2), 106-131, ACM Press, 2002.

[2] David Cline, Parris K. Egbert. Interactive Display of Very Large Textures. *Proceedings of IEEE Visualization 1998*, 343-350, IEEE Computer Society Press, 1998.

[3] Jürgen Döllner, Konstantin Baumann, Klaus Hinrichs. Texturing Techniques for Terrain Visualization. *Proceedings of IEEE Visualization 2000*, IEEE Computer Society Press, 2000, 207-234.

[4] Carl Erikson, Dinesh Manocha, William V. Baxter, III. HLODs for Faster Display of Static and Dynamic Environments. *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, 111-120, ACM Press, 2001.

[5] Christian Frueh, Russel Sammon, Avideh Zakhor. Automated Texture Mapping of 3D City Models with Oblique Aerial Imagery. *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization, and Transmission*, 396-403, IEEE Computer Society Press, 2004.

[6] Gerd Hesina, Stefan Maierhofer, Robert F. Tobler. Texture Management for High-Quality City Walk-Throughs. Technical Report, No. 25, 2004.

[7] Hugues Hoppe. Progressive Meshes, *Proceedings of ACM SIGGRAPH 1996*, 99-108, ACM Press, 1996.

[8] Lok M. Hwa, Mark A. Duchainau, Kenneth I. Joy. Adaptive 4-8 Texture Hierarchies. *Proceedings of IEEE Visualization 2004*, 219-226, IEEE Computer Society Press, 2004.

[9] Takeo Igarashi, Dennis Cosgrove. Adaptive Unwrapping for Interactive Texture Painting. *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, 209-216, ACM Press, 2001.

[10] Martin Kraus, Thomas Ertl. Adaptive Texture Maps. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 7-15, Eurographics Association, 2002.

[11] Ali Lakhia: Efficient Interactive Rendering of Detailed Models with Hierarchical Levels of Detail. *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization, and Transmission*, 275-278, IEEE Computer Society Press, 2004.

[12] Sylvain Lefebvre, Jérome Darbon, Fabrice Neyret. Unified Texture Management for Arbitrary Meshes, INRIA Research Report No. 5210, 2004.

[13] Sylvain Lefebvre, Fabrice Neyret. Pattern-Based Procedural Textures. *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, 203-212, ACM Press, 2003.

[14] David Luebke, Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. *Proceedings of SIGGRAPH 1997*, 199-208, ACM Press, 1997.

[15] David Luebke, Ben Hallen. Perceptually Driven Simplification for Interactive Rendering. *Proceedings of 2001 Eurographics Rendering Workshop*, 223-234, Springer-Verlag, 2001.

[16] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16, 187-260, ACM Press, 1984.

[17] Pedro V. Sander, John Snyder, Steven J. Gortler, Hugues Hoppe. Texture Mapping Progressive Meshes, *Proceedings of ACM SIGGRAPH 2001*, 409-416, ACM Press, 2001.

[18] Andreas Schilling, Reinhard Klein. Rendering of Multiresolution Models with Texture. *Computers and Graphics*, 22(6), 667-674, 1998.

[19] Wenzhong Shi, Bisheng Yang, Qinquan Li. Integrated Dynamic Model for Multi-Resolution Data in Three Dimensional GIS. *Proceedings of the Joint International Symposium on Geospatial Theory, Processing, and Applications*, 2002.

[20] Mario Sormann, Christopher Zach, Konrad F. Karner. Texture Mapping for View-Dependent Rendering. *Proceedings of the 19th Spring Conference on Computer Graphics*, 131-148, ACM Press, 2003.

[21] Christopher C. Tanner, Christopher J. Midgal, Michael T. Jones. The Clipmap: A Virtual Mipmap. *Proceedings of the SIGGRAPH 1998*, 151-158, ACM Press, 1998.

[22] Roland Wahl, Manuel Massing, Patrick Degener, Michael Guthe, Reinhard Klein. Scalable Compression and Rendering of Textured Terrain Data, *Journal of WSCG*, 12(3), 521-528, UNION Agency - Science Press, 2004.

[23] Nathaniel Williams, David Luebke, Jonathan D. Cohen, Michael Kelley, Brenden Schubert. Perceptually Guided Simplification of Lit, Textured Meshes. *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, 113-121, ACM Press, 2003.

[24] Matthias Wloka. Improved Batching via Texture Atlases. *ShaderX3*, 155-167, Charles River Media, 2005.