Texture Mapping for View-Dependent Rendering

Mario Sormann *
Christopher Zach †
Konrad Karner ‡

VRVis Research Center

Abstract

View-dependent multiresolution meshes allow smooth interactive animation and optionally time-critical rendering of huge geometric data-sets and are therefore an important tool for large-model visualization. So far most view-dependent rendering frameworks are restricted to models with a topologically simple texture mapping. Our approach overcomes this restriction with a new texturing technique, which allows texture mapping during the runtime simplification process. In fact, novel algorithm generates a graph of textures in a preprocess automatically. This texture graph is furthermore integrated into a view-dependent rendering approach. Particularly we perform a texture validation step for the whole vertex tree, which is the basic data structure in the framework. At runtime the vertex tree is traversed and we introduce a texture proxy map to assure correct texture mapping during view-dependent rendering. Additionally the new technique allows us to guarantee a constant frame rate. Finally the results of our method enhancing visual detail of geometric models are shown.

Keywords: texture mapping, texture generation, texture atlas, level of detail, multiresolution meshes, real-time rendering.

1 Introduction

Real-time visualization of very large data-sets is a challenging problem in computer graphics, especially it is a growing domain with an important number of applications. Example areas are architectural visualizations, virtual environments and flight simulations. However, the bottleneck of these applications is the rendering performance. To guarantee interactive or even real-time rendering view-dependent rendering of multiresolution polygonal meshes can be utilized. View-dependent rendering of multiresolution meshes is sometimes referred as *dynamic levels of detail*. These multiresolution meshes can be generated automatically using well known simplification techniques. Approaches of this type update the displayed mesh according to the current viewing parameters. These updates are essentially refinements or coarsening of local patches of the mesh. A single refinement or coarsening step applied to a textured mesh may create regions with potentially much clamping artifacts. To avoid these artifacts and consequently meshes with very poor visual quality, a texture atlas [12] (i.e. a set of textures mapped onto patches) can be utilized. Our work describes primarily the integration of texture atlases into a view-dependent rendering framework. Several algorithms generate such atlases for multiresolution meshes, but normally approaches of this type are not suitable for real-time rendering, because they assume simple textured models with perhaps one or a limited number of texture images. In a real-time rendering framework one has to handle richly textured multiresolution representations with several associated texture images and texture coordinate discontinuities.

In this paper we present a general dynamic level of detail framework, which can handle textured view-dependent multiresolution meshes. We use the view-dependent simplification (VDS) framework [10] to provide view-dependent rendering. Without a convenient texture hierarchy the simplification process is very limited [17], therefore it is necessary to utilize a kind of texture atlas.

In a preprocessing step the texture hierarchy (graph) is generated, i.e. several textures are replaced by one with a lower resolution. The obtained texture is mapped on a larger patch of the mesh. After several iterations we obtain a graph of textures with different resolutions and geometric extents. Furthermore we integrate this texture hierarchy into the VDS framework, in particular we validate a suitable texture set for each possible triangle. At runtime appropriate textures are selected for each visible triangle. Consequently we are able to handle richly textured view-dependent multiresolution meshes and we reduce clamping artifacts due to the simplification process.

*sormann@icg.tu-graz.ac.at
†zach@vrvis.at
‡karner@vrvis.at

2 Related Work

2.1 View-Dependent Multiresolution Meshes

The generation of view-dependent multiresolution meshes is a very active research area in recent years. A very popular approach is the view-dependent progressive mesh concept proposed by Hoppe [7] and its variants [4, 16]. In these approaches the key operation to locally modify the displayed mesh comprise the edge collapse, which replaces two vertices in the source mesh by one, and its inverse operation, the vertex split. One edge collapse usually removes one edge and several triangles in the current mesh. The sequence of edge collapses should be carefully chosen, because the overall shape of the model has to be preserved. Several metrics to rank edge collapses were proposed [5, 6].

Many researchers observed that the linear sequence of edge collapses can be replaced by a more general partial ordering, which can be encoded as a vertex tree. Our LOD-framework is based on `VDSlib`[1], which is the implementation of the approach proposed by Luebke and Erikson [10]. They introduced a generalized *vertex tree*, which can incorporate other mesh simplification methods than edge collapses. The refinement operation is *vertex unfolding*, which replaces one vertex by several new vertices. The inverse operation clusters (folds) a set of vertices into one representative vertex. These local mesh operations insert or remove vertices, and additionally displayed triangles may change their shape. The vertex tree is created during a preprocessing step and at runtime the nodes in the tree can be separated into three categories, which are illustrated in Figure 1. Rendered triangles of the currently displayed mesh belong to *active nodes*. *Boundary nodes* comprise the corners of displayed triangles, whereas *inactive* nodes do not contribute to the displayed mesh at all. For each frame, the vertex tree is traversed and the actual rendered mesh, in particular the active nodes are selected according to some error metric.

2.2 Texture Mapping for Multiresolution Models

Several authors discussed the problem of texture mapping in combination with multiresolution meshes. Most approaches assume a 2D parameterization of the original model to interpolate texture coordinates during simplification. Sander et al. [12] created a global texture map called texture atlas from the mesh texture information. With the help of this texture atlas a progressive mesh sequence is constructed, such that the texture deviation caused by the simplification procedure is minimized.

Garland and Heckbert [5] explicitly included material properties in their quadric error metric. Cohen et al. [3] proposed a linear mapping function for each simplification and utilized this function to measure deviation of the current surface from the previous level of detail and furthermore to compute appropriate texture coordinates.

Our texture hierarchy generation method (described in Section 3.1) extends the approach proposed by Hoppe [8]. He introduces a block based simplification scheme, which constructs a progressive mesh as a hierarchy of block refinements. Initially the geometry is simplified such that generated triangles stay within the texture borders at the highest level until no further simplification is possible. Afterwards the next coarser texture is used to allow further edge collapses. These steps are repeated until the root texture is reached. In Hoppes work the texture hierarchy consists of a quad-tree like texture tree.

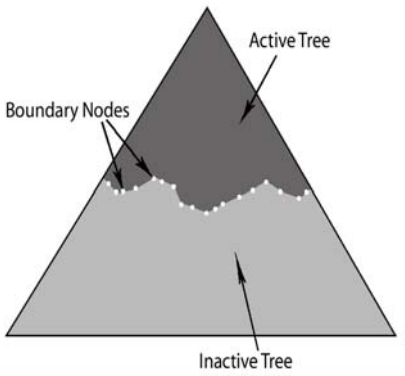


Figure 1: A schematic view on the vertex tree. The triangles associated with (dark shaded) active nodes are actually displayed. Boundary nodes comprise the corners of rendered triangles.

2.3 Impostor for Real-Time Visualization

The idea of using image-based representations to replace complex geometric objects in virtual environments was first introduced by Maciel and Shirley [11]. A particular object is rendered into a texture map with transparency information and then mapped onto a quadrilateral placed into the scene instead of the object. The resulting quadrilateral is called *impostor*. In other approaches [9, 13] images of different distance are cached and replace complex geometry according an image discrepancy criterion.

Sillion et al. [14] introduced a method based on the general level of detail approach. The method utilizes impostors for distant scenery in combination with three dimensional geometry to correctly model large depth effects and therefore reducing the parallax problem. Additionally such impostors, often called 3D impostors, can be used for a much larger number of viewpoints, which further increase the visual quality of the scene.

A similar approach for rendering 3D geometric models at a guaranteed frame rate is proposed by Aliaga and Las-

[1] see `http://vdslib.virginia.edu`

tra [1]. A preprocessing algorithm stores the 3D model in a hierarchical data structure (e.g. octree) and at each grid point a well defined number of images is created to represent a subset of the model. The runtime system determines images from a grid point closest to the current viewpoint and a render engine displays these images surrounded by geometry.

A rather new impostor enhancement is the use of point-based impostors instead of regular textures [15]. Such impostors allow longer cache life, this means that, similar to 3D impostors, they are visually correct for a larger number of different viewpoints.

3 Texture Mapping for View-Dependent Multiresolution Meshes

In this Section we present a texture mapping technique for view-dependent multiresolution meshes in a real-time rendering framework. As already mentioned our approach is based on freely available `VDSlib` library. We integrate texture mapping into this view-dependent simplification framework, avoiding clamping and other rendering artifacts. Our approach consists of a preprocessing component which automatically generates a graph of textures. This graph is used for the texture validation pass in our multiresolution framework. At runtime we select the most suitable texture for each visible triangle.

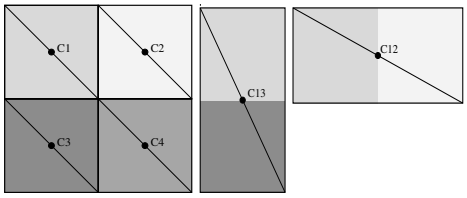
3.1 Automatic Texture Hierarchy Generation

Here we address the question how to automatically create a graph of textures from a set of well defined and structured texture rectangles. This means that in case of terrain textures a quad-tree structure can be used to accomplish this type of problem. But for more complex scenes the generation of a texture hierarchy can not be done in such a simple way.

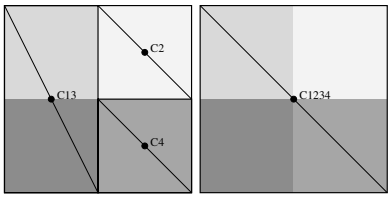
In our approach we replace two textures by one with a lower resolution and a larger geometric extent (i.e. the resulting texture can be mapped on all triangles associated with the source textures). Furthermore we regard textures as rectangular images orthographically projected onto mesh patches. In order to generate a *texture graph*, we have to perform more pairwise texture merges through every iteration. Hence we define a metric to rank all possible texture merges.

In fact, our texture metric is represented as a tuple of some parameters. Thus we get for every pair of textures a score according to the defined metric. In every iteration of the texture hierarchy generation those pairs of textures with highest scores are merged. During the iteration process textures are replaced by the combined one. The texture generation terminates, when the best possible texture score is less than some predefined threshold. As output, the algorithm provides a graph of textures with increasing coarser resolutions. Figure 2 illustrates the simplification sequence.

Finally this texture graph, the textures and important texture parameters are stored for further use. It should also be mentioned that the resampling of coarser textures is done with OpenGL.



(a) Original texture planes (b) Best and additional candidate



(c) After first iteration (d) After last iteration

Figure 2: An example for a simple texture generation sequence.

3.1.1 Texture Metric

The basic criterion and procedure to generate a texture hierarchy is described in [17].

In order to define some metric to rank texture merges, every texture is represented as a tuple of some parameters. These parameters are made up of a volume enclosing the triangles associated with this texture, called the geometric hull H, the center of projection of the texture $\vec{c}$, the direction of projection $\vec{d}$ and finally $\vec{u}$ and $\vec{v}$ are the right and the up vectors of the texture (see Figure 3). These values induce an orthographic projection and consequently a projection matrix.

The score function is formulated in terms of these texture parameters and based on some observations:

- *Relative orientation* of textures: the directions d of two merge candidates should be similar (collinear) to avoid textures with very poor visual quality.
- *Loss*: Textures are rectangular, thus there is usually some loss, if one has to merge two arbitrarily oriented textures. Further, a large loss usually implies low visual quality.
- *Absolute texture size*: Also the size of the new texture is important for the resulting score function.

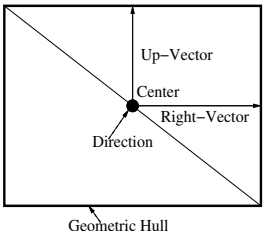


Figure 3: Basic texture characteristics. Geometric hull H, center of projection $\vec{c}$, direction $\vec{d}$ and right and up vectors $\vec{u}$ and $\vec{v}$ of the texture are the basic parameters to describe a single texture.

- *Depth distance*: Additionally, in consideration of the merge result, it is not useful to merge textures with very distant texture hulls.

All the above aspects are covered by the score function, assigning higher scores to potentially good texture merges. In fact, the score function sf is a linear combination of terms quantitatively evaluating the mentioned items

$$sf = w_1 * s_1 + w_2 * s_2 + w_3 * s_3 + w_4 * s_4$$

where $w_{1...4}$ are the weights and $s_{1...4}$ are the scores of introduced terms.

3.1.2 Texture Graph Details

Each node in the texture graph includes basic texture parameters, which are used in our extended VDS framework for the texture validation process and the texture mapping. These parameters include the texture projection matrix, which is used for the texture validation process, the two texture coefficients for automatic texture coordinate generation and the direction of the texture plane. Furthermore to produce synthetic scenes with several same textures mapped onto different parts of the mesh, it is necessary to introduce a texture name as well as a texture identity. Figure 4 gives an example of a texture graph, produced by the texture generation algorithm.

3.2 Texture Validation

The vertex tree as illustrated in Figure 5, created during a preprocessing stage, is the fundamental data structure of the VDS framework. Leaf nodes of the tree specify single vertices in the original model and interior nodes are representative vertices of its successor nodes. Every node stores geometric data, its children and a list of associated triangles, the so called *subtris*. To support texture mapping it is necessary to implement some extensions in the data structure of the vertex tree. The essential internal structure is the so far mentioned sub-triangles. They are used to validate suitable textures for each node in the vertex tree.

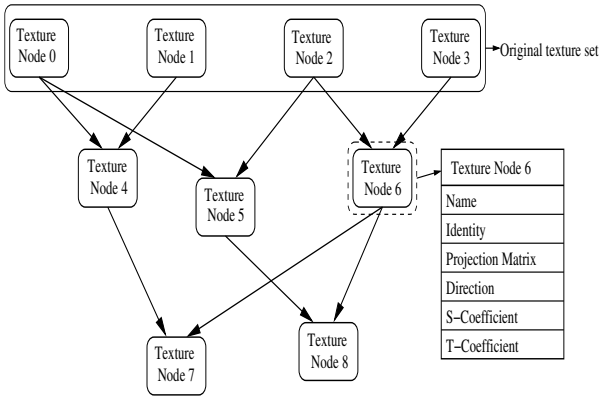


Figure 4: A small texture graph example. Nodes correspond to texture images and arrows represent texture merge steps. The nodes in the bottom row represent coarse texture covering large portions of the mesh.

3.2.1 The Validation Process

In our implementation the texture validation process is performed offline as a preprocessing step. In Figure 5 some nodes are emphasized with a rectangle. These nodes are possible corners for the sub-triangle $\overline{V1V2V3}$. Additionally each sub-triangle has its own original texture. In order to discover which textures stored in the texture graph are suitable candidates for a node, it is necessary to check all possible variations of corners (i.e. all shapes of triangles possible at runtime). Hence we keep two of three corners constant and the third one is varied according to the valid path in the vertex tree. After all three corners are tested, we obtain a set of possible triangles. Now the algorithm determines which triangles from this set exceed the primary texture, such that texturing this triangle yields to clamping artifacts. For these critical triangles we use a depth-first-search to acquire all valid textures in the texture graph. We extend the data structure of the vertex tree to provide these valid textures for the runtime management. Hence the valid textures for each sub-triangle are inserted into the set of allowed textures of the owner node.

In case of no crossing texture borders the original texture is well suited and no new texture is registered to the node structure. The algorithm stops when all sub-triangles are processed. Now the vertex tree provides for each simplification possibility a set of suitable textures. To concretise these ideas the pseudo-code in Section 3.2.3 gives an overview of our implementation.

3.2.2 Data Structure Extensions

In our approach we implement three essential extensions in the data structure of the VDS-framework. For texture handling we first need a general structure for them. This is accomplished with the following basic structure:

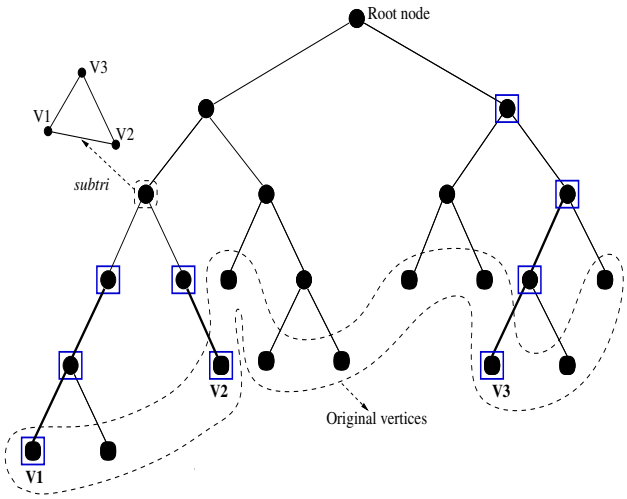


Figure 5: The vertex tree. Leaf nodes correspond to original vertices in the source mesh, whereas interior nodes are generated by the mesh simplification procedure. At runtime the original corners of the highlighted triangle can be replaced by one of its (emphasized) ancestor nodes.

```
struct RuntimeTexture {
   string       texturename;
   unsigned int textureid;
   float        svec[4];
   float        tvec[4];
   Vector3f     direction;
   Matrix4f     projection;
};
```

Consequently the *RuntimeTexture* stores the relevant parameters acquired from the texture graph. Further, the triangle structure must be extended with a texture identity parameter, thus we are able to obtain which textures are used for rendering. Finally, as mentioned above, each node of the vertex tree is upgraded with a set of suitable textures:

```
struct RuntimeNode {
   vds relevant parameters;
   ...
   RegTextures  rtextures;
};
```

With these additional data we are able to select the most suitable textures for each frame at runtime.

3.2.3 Implementation

In this section we show the straightforward pseudo-code for the texture validation process, which is currently implemented in C++.

3.3 Runtime Management

This section describes the texture mapping process during runtime. As in Section 2.1 already mentioned at runtime

Algorithm 1 The texture validation process

Require: vertex tree, texture graph
1: **for all** subtris in the vertex tree **do**
2: find all possible variations for the current subtri
3: **for all** possible triangle variations **do**
4: project the triangle to the the current texture rectangle according to the projection matrix
5: check the normalized device coordinates
6: **if** one of the triangle corners cross the texture border **then**
7: find new texture options
8: register the new textures to the current node structure
9: **end if**
10: **end for**
11: **end for**

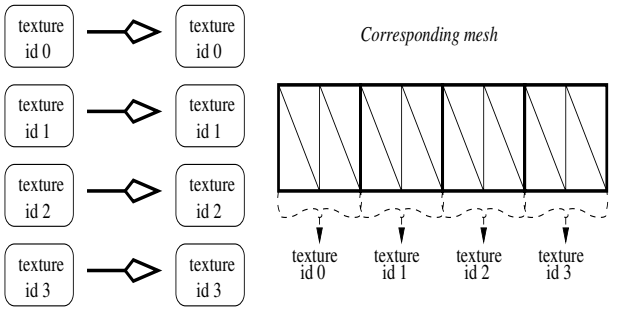
the vertex tree is traversed in a top down order and active nodes are determined according to some screen error metric. Subtris of active nodes comprise the triangles selected to render. In the framework of Luebke and Erikson [10] triangles are rendered directly whereas in our approach we implement several significant extensions to the common framework, which will be explained in the following Sections. Furthermore we utilize a *texture proxy map* to ensure correct mapping of textures during the simplification.

Texture Proxy Map The *texture proxy map* is a direct mapping from the original textures to the textures which are used for the current refinement. At the beginning the texture proxy map is the identity (Figure 6a). Obviously if this map is consistently updated, then we can guarantee correct texture mapping during simplification (Figure 6b).

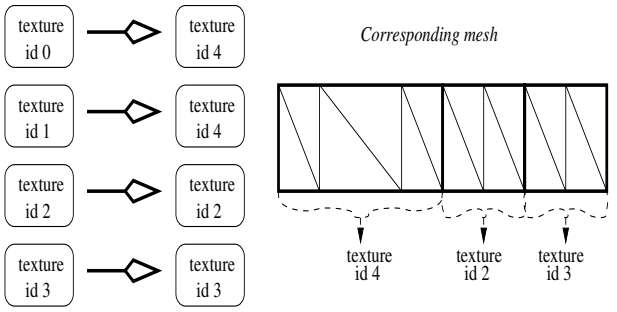
Texture selection In Section 3.2.1 we mentioned that a set of valid textures is inserted into the nodes of the vertex tree. Now, at runtime we need to select the best possible texture for the current viewpoint in this texture set. Therefore the viewing direction needs to be compared with the normal vector of the texture rectangle, to select the most suitable texture. On the other hand, when the set of suitable textures in the processed node is empty, the original texture is valid and used for the texture mapping at runtime.

After the most convenient texture is determined, we perform a breadth-first search in the texture graph to acquire the common texture ancestor of the new texture and the texture which is already used in the texture proxy map. This common texture is further used to update the texture proxy map.

Algorithm 2 provides the pseudo-code for the runtime management.



(a) Initial texture proxy map



(b) Updated texture proxy map. A triangle linked with texture 0 would be rendered with texture 4 in this case.

Figure 6: The texture proxy map that is maintained at runtime. This mapping assigns actual textures to original texture ids.

View Frustum Culling View-dependent refinement assigns higher resolution to the parts of the mesh inside the viewing frustum and the lowest possible resolution to invisible portions. If all triangles are considered for texture selection, invisible but coarse parts of the mesh force coarser textures to be selected. Thus a view frustum culling has to be done before selecting suitable textures. Otherwise, the texture becomes coarser if a user zooms closely to some mesh detail. This situation is illustrated in Figure 7, where all active triangles are considered for texture selection. Without view frustum culling the light shaded texture is erroneously selected instead of the dark shaded ones.

Therefore texture selection must only be based on visible triangles instead of all active triangles. This reduction is achieved by culling active triangles against the viewing frustum. Determining visible triangles can be accelerated by exploiting the spatial hierarchy induced by the vertex tree: if the bounding sphere of a node is entirely inside or outside the viewing frustum, the visibility status of all subtris is known immediately. Otherwise the subtris must be checked for visibility individually.

Algorithm 2 Texture selection during runtime

Require: Active triangles
1: **for all** active triangles **do**
2: perform the view frustum culling
3: **end for**
4: **for all** visible triangles **do**
5: **if** registered texture set is empty **then**
6: use original texture
7: **else**
8: find the most suitable texture
9: update texture proxy map
10: **end if**
11: **end for**

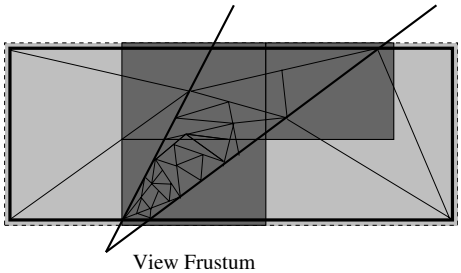


Figure 7: View frustum culling. Without view frustum culling before the texture selection, the light shaded texture is used instead of the dark shaded ones.

Rendering After the texture selection and the view frustum culling the actual mesh with the proper texture is rendered. Initially the list of active triangles is sorted with respect to the associated proxy texture to minimize graphics state changes. Whenever a set of triangles with a common texture needs to be rendered, the appropriate coefficients for automatic texture coordinate generation stored in the proxy are passed to the graphics library.

4 Results

The majority of the material we will present is focused on testing our approach on artificial and real data-sets. Furthermore we will analyze the performance of our method. Note, that all results reported were obtained on a 666Mhz Intel processor with 512MB main memory and a GeForce2 GTS with 32MB graphics memory.

Three data-sets (artificial and real) were tested to span several model categories (Figure 8 to Figure 10) and to cover a large range of texture counts.

Model	Triangles	Original Textures	Complete Texture Set	Estimated Space (MB)
Grid	32768	16	39	5.1
Sphere	5120	20	72	3.0
Alley	24576	12	29	13.8

Table 1: The names, complexity and the texture memory consumption of the tested models.

One of the artificial data-sets consists a *Grid* of texture patches (see Figure 8). Typical terrain datasets are topologically similar to such regular grids. As expected, texture mapping of such meshes produces no texture distortions or other rendering artifacts, thus we obtain a smooth transition between the representations (Figure 8a-d).

The correct texture mapping for complex 3D models is shown on a *Sphere*. Figure 9 illustrates several approximations applied on a sphere. We obtain a high visual quality for the simplified sphere, compared to the original mesh.

Also the approach has been used for texture mapping in urban environments, particularly in virtual city models. Figure 10 shows two mesh approximations of an typical *Alley*. The face count of the simplified mesh is relatively low, with respect to the original mesh of nearly 15,000 faces. In contrast to the face count the visual quality of the simplified mesh is slightly reduced.

It should be also mentioned that the underlying geometry for the original model is synthetically subdivided using a common subdivision algorithm.

4.1 Performance

Table 2 summarizes the validation performance of our approach on all these models. As expected, the validation performance depends mainly on the triangle count of the models.

Model	Triangles	Complete Texture Set	Validation Time (ms)
Grid	32768	39	496.938
Sphere	5120	72	75.697
Alley	24576	29	409.546

Table 2: Texture validation performance for the test models.

Figure 11 shows a comparison between the overall rendering time and the texture selection time for a defined path applied on the alley. It can be clearly observed that the texture selection requires only 10 to 15% of the overall rendering time. We should also mention that the texture selection is performed for a complete texture set of 29 textures. But for thousand of textures the time spent in texture selection will be probably more significant and should be therefore accelerated.

5 Conclusion and Future Work

In this work we presented the integration of general texture mapping into a view-dependent rendering framework. Textures for the actual displayed mesh are chosen from a pool of possible candidates covering various patches of the mesh at different resolutions. Criteria for runtime texture selection incorporate geometric validity and visual fidelity for the current viewing parameters. To our best knowledge this approach is the only framework that handles richly textured multiresolution representations with several associated texture images and texture coordinate discontinuities.

Our approach represents a smooth transition between highly detailed objects in the near field and coarse impostor-based geometry in the far field. Therefore it can be seen as a natural generalization of traditional level of detail and impostor approaches with only slightly increased memory and performance costs.

Future work will address several aspects:

- The visual quality should be enhanced, e.g. by utilization of more sophisticated metrics to rank texture merges [2].
- The numbers of generated textures should be minimized to save memory bandwidth and to allow faster streaming over a network.
- Runtime texture selection must be accelerated to handle very large scenes with possibly thousands of textures. For these scenes the time spent in texture selection will be probably significant. An incremental approach similar to the method proposed in [18] to refine the selected textures can be beneficial.

6 Acknowledgements

This work has been done in the VRVis research center, Graz and Vienna/Austria (http://www.vrvis.at), which is partly funded by the Austrian government research program Kplus.

References

[1] Daniel G. Aliaga and Anselmo Lastra. Automatic image placement to provide a guaranteed frame rate. In *Proceedings of SIGGRAPH '99*, pages 307–316, 1999.

[2] Alexander Bornik, Peter Cech, Andrej Ferko, and Roland Perko. Beyond image quality comparison. Technical report, Technical University Graz, 2003.

[3] Jonathan Cohen, Dinesh Manocha, and Marc Olano. Simplifying polygonal models using successive mappings. In *Proceedings of IEEE Visualization '97*, pages 395–402, 1997.

[4] J. El-Sana and A. Varshney. Generalized view-dependent simplification. In *Proceedings of EUROGRAPHICS '99*, pages 83–94, 1999.

[5] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of IEEE Visualization '98*, pages 263–270, 1998.

[6] Hugues Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96*, volume 30, pages 99–108, 1996.

[7] Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proceeding of SIGGRAPH '97*, volume 31, pages 189–198, 1997.

[8] Hugues H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings IEEE Visualization '98*, pages 35–42, 1998.

[9] Stefan Jeschke and Michael Wimmer. Textured depth meshes for real-time rendering of arbitrary scenes. In *Eurographics Workshop on Rendering '02*, 2002.

[10] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97*, volume 31, pages 199–208, 1997.

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

[12] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *Proceedings of SIGGRAPH '01*, pages 409–416, 2001.

[13] Gernot Schaufler and Wolfgang Stürzlinger. A three dimensional image cache for virtual reality. In *Proceedings of EUROGRAPHICS '96*, pages 227–236, 1996.

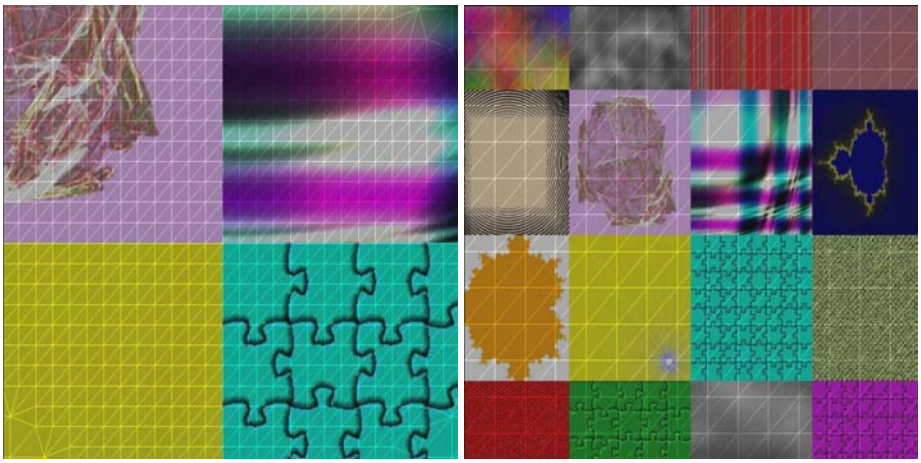
[14] François Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. In *Proceedings of EUROGRAPHICS '97*, pages 207–218, 1997.

[15] Michael Wimmer, Peter Wonka, and Francois Sillion. Point-based impostors for real-time visualization. In *Proceedings of the Eurographic Workshop on Rendering '01*, pages 163–176, 2001.

[16] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96*, pages 335–344, 1996.

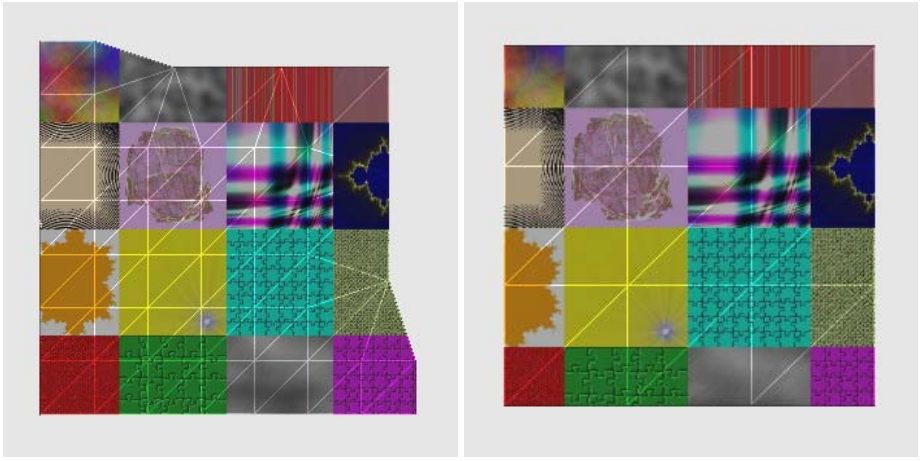
[17] Christopher Zach and Joachim Bauer. Automatic texture hierarchy generation from orthographic facade textures. In *Workshop of the Austrian Association for Pattern Recognition '02*, 2002.

[18] Christopher Zach and Konrad Karner. Fast event-driven refinement of dynamic levels of detail. In *Proceedings of the Spring Conference on Computer Graphics '03*, 2003.



(a) Close up of the base mesh (2178 triangles)

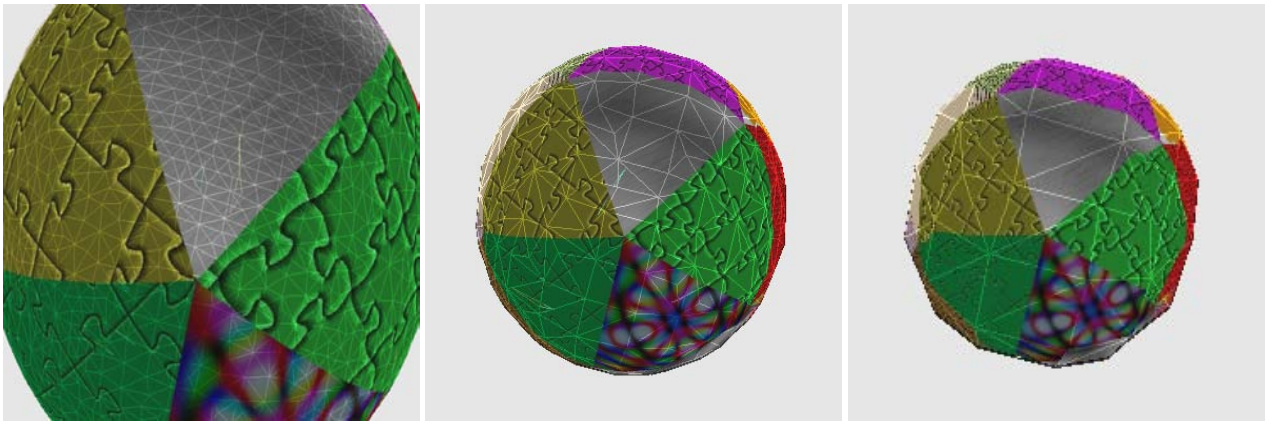
(b) Simplified mesh (350 triangles)



(c) Intermediate mesh (74 triangles)

(d) The coarsest mesh (18 triangles)

Figure 8: Images obtained from an interactive flight over the artificial grid dataset. Correct texture mapping of all multiresolution representations can be guaranteed.

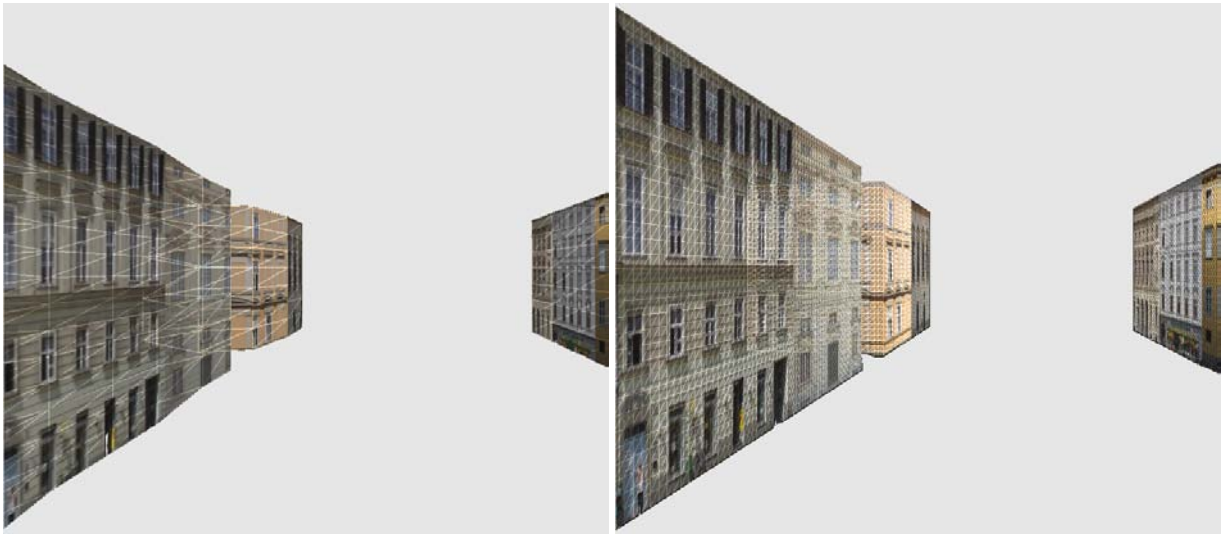


(a) High resolution mesh (3976 triangles)

(b) Simplified mesh (523 triangles)

(c) Coarse mesh (108 triangles)

Figure 9: Snapshots from three simplification steps applied on a sphere. In spite of a drastic geometric simplification a smooth transition in the visual quality between (a) and (c) can be obtained.



(a) A scene from an simplified alley in an urban environment (613 triangles)

(b) Same scene as (a) but rendered with the highest possible resolution (14343 triangles)

Figure 10: These two snapshots illustrate an example rendering of facade textures which are normally used in urban environments. Despite the drastic simplification of the geometric resolution in (a) and (b), the visual quality is only slightly different.

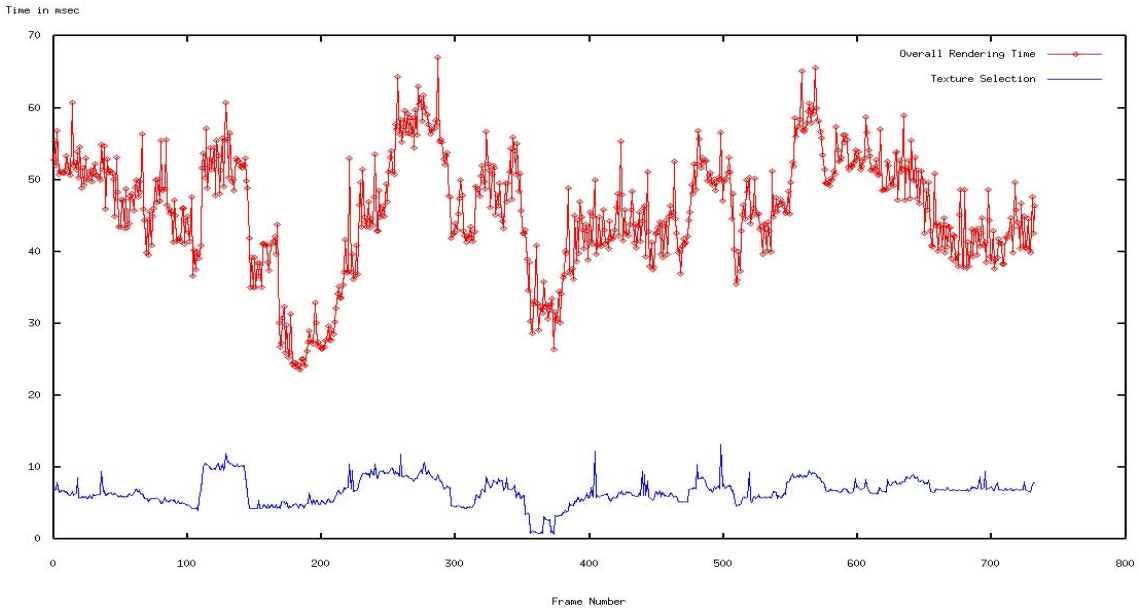


Figure 11: Overall rendering time and texture selection time for the alley model applying a defined path. The complete texture set consists of 29 textures.